Generating and Auto-Tuning Parallel Stencil Codes

Inauguraldissertation

zur Erlangung der Würde eines Doktors der Philosophie
vorgelegt der Philosophisch-Naturwissenschaftlichen Fakultät
der Universität Basel

von

Matthias-Michael Christen

aus Affoltern BE, Schweiz

Basel, 2011

creative commons

Attribution – NonCommercial – NoDerivs 2.5 Switzerland

You are free:

to Share — to copy, distribute and transmit the work

Under the following conditions:

Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Noncommercial — You may not use this work for commercial purposes.

No Derivative Works —You may not alter, transform, or build upon this work.

With the understanding that:

Waiver — Any of the above conditions can be *waived* if you get permission from the copyright holder.

Public Domain — Where the work or any of its elements is in the *public domain* under applicable law, that status is in no way affected by the license.

Other Rights — In no way are any of the following rights affected by the license:

- Your fair dealing or *fair use* rights, or other applicable copyright exceptions and limitations;
- The author's *moral* rights;
- Rights other persons may have either in the work itself or in how the work is used, such as *publicity* or privacy rights.

Notice — For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to the web page `http://creativecommons.org/licenses/by-nc-nd/2.5/ch`.

Disclaimer — The Commons Deed is not a license. It is simply a handy reference for understanding the Legal Code (the full license) – it is a human-readable expression of some of its key terms. Think of it as the user-friendly interface to the Legal Code beneath. This Deed itself has no legal value, and its contents do not appear in the actual license. Creative Commons is not a law firm and does not provide legal services. Distributing of, displaying of, or linking to this Commons Deed does not create an attorney-client relationship.

Genehmigt von der Philosophisch-Naturwissenschaftlichen Fakultät auf Antrag von

Prof. Dr. Helmar Burkhart
Prof. Dr. Rudolf Eigenmann

Basel, den 20. September 2011

Prof. Dr. Martin Spiess,
Dekan

Abstract

In this thesis, we present a software framework, PATUS, which generates high performance stencil codes for different types of hardware platforms, including current multicore CPU and graphics processing unit architectures. The ultimate goals of the framework are productivity, portability (of both the code and performance), and achieving a high performance on the target platform.

A stencil computation updates every grid point in a structured grid based on the values of its neighboring points. This class of computations occurs frequently in scientific and general purpose computing (e.g., in partial differential equation solvers or in image processing), justifying the focus on this kind of computation.

The proposed key ingredients to achieve the goals of *productivity*, *portability*, and *performance* are domain specific languages (DSLs) and the auto-tuning methodology.

The PATUS stencil specification DSL allows the programmer to express a stencil computation in a concise way independently of hardware architecture-specific details. Thus, it increases the programmer *productivity* by disburdening her or him of low level programming model issues and of manually applying hardware platform-specific code optimization techniques. The use of domain specific languages also implies code reusability: once implemented, the same stencil specification can be reused on different hardware platforms, i.e., the specification code is *portable* across hardware architectures. Constructing the language to be geared towards a special purpose makes it amenable to more aggressive optimizations and therefore to potentially higher *performance*.

Auto-tuning provides *performance* and performance *portability* by automated adaptation of implementation-specific parameters to the characteristics of the hardware on which the code will run. By automating

the process of parameter tuning — which essentially amounts to solving an integer programming problem in which the objective function is the number representing the code's performance as a function of the parameter configuration, — the system can also be used more *productively* than if the programmer had to fine-tune the code manually.

We show performance results for a variety of stencils, for which PATUS was used to generate the corresponding implementations. The selection includes stencils taken from two real-world applications: a simulation of the temperature within the human body during hyperthermia cancer treatment and a seismic application. These examples demonstrate the framework's flexibility and ability to produce high performance code.

Contents

Acknowledgments

I would like to thank Prof. Dr. Helmar Burkhart and PD Dr. Olaf Schenk for having been given the opportunity to start this project and for their research guidance, their support, advice, and confidence.

I would also like to thank Prof. Dr. Rudolf Eigenmann for kindly agreeing to act as co-referee in the thesis committee and for reading the thesis.

I am grateful to the other members of research group, Robert Frank, Martin Guggisberg, Florian Müller, Phuong Nguyen, Max Rietmann, Sven Rizzotti, Madan Sathe, and Jürg Senn for contributing to the enjoyable working environment and for stimulating discussions.

I wish to thank the people at the Lawrence Berkeley National Laboratory for welcoming me – twice – as an intern in their research group and for the good cooperation; specifically my thanks go to Lenny Oliker, Kaushik Datta, Noel Keen, Terry Ligocki, John Shalf, Sam Williams, Brian Van Straalen, Erich Strohmaier, and Horst Simon.

Finally, I would like to express my gratitude towards my parents for their support.

This project was funded by the Swiss National Science Foundation (grant No. 20021-117745) and the Swiss National Supercomputing Centre (CSCS) within the *Petaquake* project of the Swiss Platform for High-Performance and High-Productivity Computing (HP2C).

Chapter 1

Introduction

The advent of the multi- and manycore era has led to a software crisis. In the preceding era of frequency scaling, performance improvement of software came for free with newer processor generations. The current paradigm shift in hardware architectures towards more and simpler "throughput optimized" cores, which essentially is motivated by the power concern, implies that, if software performance is to go along with the advances in hardware architectures, parallelism has to be embraced in software. Traditionally, this has been done for a couple of decades in high performance computing. The new trend, however, is towards multi-level parallelism with gradated granularities, which has led to mixing programming models and thereby increasing code complexity, exacerbating code maintenance, and reducing programmer productivity.

Hardware architectures have also grown immensely complex, and consequently high performance codes, which aim at eliciting the machine's full compute power, require meticulous architecture-specific tuning. Not only does this obviously require deeper understanding of the architecture, but also is both a time consuming and error-prone process.

The main contribution of this thesis is a software framework, PATUS, for a specific class of computations — namely nearest neighbor, or *stencil* computations — which emphasizes productivity, portability, and performance. PATUS stands for **P**arallel **A**uto-**Tu**ned **S**tencils.

A stencil computation updates every grid point in a structured grid based on the values of its neighboring points. This class of computations is an important class occurring frequently in scientific and general purpose computing (e.g., in PDE solvers or in image processing), justifying the focus on this kind of computation. It was classified as the core computation of one of the currently 13 computing patterns — or *motifs* — in the often-cited *Landscape of Parallel Computing Research: A View from Berkeley* [9].

The proposed key ingredients to achieve the goals of productivity, portability, and performance are domain specific languages and the auto-tuning methodology. The domain specific language approach enables the programmer to express a stencil computation in a concise way independently of hardware architecture-specific details such as a low level programming model and hardware platform-specific code optimization techniques, thus increasing productivity. In our framework, we furthermore raise productivity by separating the specification of the stencil from the algorithmic implementation, which is orthogonal to the definition of the stencil.

The use of domain specific languages also implies code reusability: the same stencil specifications can be reused on different hardware platforms, making them portable across hardware architectures. Thus, the combined use of domain specific languages and auto-tuning make the approach performance-portable, meaning that no performance is sacrificed for generality. This requires, of course, that an architecture-aware back-end exists, which provides the domain-specific and architecture-specific optimizations. Creating such a back-end, however, has to be done only once.

We show that our framework is applicable to a broad variety of stencils and that it provides its user with a valuable performance-oriented tool.

This thesis is organized in five parts. The first part is a survey of the current challenges and trends in high performance computing, from both the hardware and the software perspective. In the second part, our code generation and auto-tuning framework PATUS for stencil computations is introduced. It covers the specification of stencil kernels and provides some background on algorithms for saving bandwidth and synchronization overhead in stencil computations, and presents ideas how to implement them within the PATUS framework. The part is concluded with a deliberation on auto-tuners and search methods. In the third part, performance experiments with PATUS-generated codes are conducted, both for synthetic stencil benchmarks and for stencils taken from real-world applications. Implementation details on PATUS are discussed in part four, and part five contains concluding remarks and ideas how to proceed in the future.

PATUS is licensed under the GNU Lesser General Public License. A copy of the software can be obtained at `http://code.google.com/p/patus/`.

Part I

High-Performance Computing Challenges

Chapter 2

Hardware Challenges

> To return to the executive faculties of this engine: the question must arise in every mind, are they *really* even able to *follow* analysis in its whole extent? No reply, entirely satisfactory to all minds, can be given to this query, excepting the actual existence of the engine, and actual experience of its practical results.
>
> — Ada Lovelace (1815–1852)

In advancing supercomputing technology towards the exa-scale range, which is projected to be implemented by the end of the decade, power is both the greatest challenge and the driving force. Today, the established worldwide standard in supercomputer performance is in the PFlop/s range, i.e., in the range of 10^{15} floating point operations per second. Realizing that imminent scientific questions can be answered by models and simulations, the scientific world also has come to realize that accurate simulations have a demand for still higher performance, hence the exigence for ever increasing performance. Hence, the next major milestone in supercomputing is reaching one EFlop/s — 10^{18} operations per second — subject to a serious constraint: a tight energy budget.

A number of studies [21, 94, 168] have addressed the question how a future exa-scale system may look like. There are three main design areas that have to be addressed: the compute units themselves, memory, and interconnects.

Until 2004, performance scaling of microprocessors came at no effort for programmers: each advance in the semiconductor fabrication process reduces the gate length of a transistor on an integrated circuit. The transistors in Intel's first microprocessor in 1971, the Intel 4004 4-bit microprocessor, had a gate length of 10μm [103]. Currently, transistor gate lengths have shrunken to 32 nm (e.g., in Intel's Sandy Bridge architecture). The development of *technology nodes*, as a fabrication process in a certain gate length is referred to, is visualized in Fig. 2.1. The blue line visualizes the steady exponential decrease of gate lengths since 1971 (note the logarithmic scale of the vertical axis).

Overly simplified, a reduction by a factor of 2 in transistor gate lengths used to have the following consequences: To keep the electric field constant, the voltage V was cut in half along with the gate length. By reducing the length of the gates, the capacitance C was cut in half. Energy therefore, obeying the law $E = CV^2$, was divided by 8. Because of the reduced traveling distances of the electrons, the processor's clock frequency could be doubled. Thus, the power consumption of a transistor in the new fabrication process is $P_{\text{new}} = f_{\text{new}} E_{\text{new}} = 2 f_{\text{old}} \cdot E_{\text{old}}/8 = P_{\text{old}}/4$. As integrated circuits are produced on 2D silicon wafers, 4 times more transistors could be packaged on the same area, and consequently the (dynamic) power consumption of a chip with constant area remained constant. In particular, doubling the clock frequency led to twice the compute performance at the same power consumption.

The empirical observation that the transistor count on a cost-effective integrated circuit doubles every 18–24 month (for instance, as a consequence of the reduction of transistor gate lengths) is called *Moore's Law* [115]. Although stated in 1965, today, almost half a century later, the observation still holds. The red line in Fig. 2.1, interpolating the transistor counts of processors symbolized by red dots, visualizes the exponential trend. The number of transistors is shown on the right vertical logarithmic axis.

A widespread misconception about Moore's Law is that *compute performance* doubles every 18–24 months. The justification for this is that, indeed, as a result of transistor gate length reduction, both clock frequency and packaging density could be increased — and compute performance is proportional to clock frequency.

However, reducing the voltage has a serious undesired consequence.

The leakage power in semiconductors is increased dramatically relative to the dynamic power, which is the power used to actually switch the gates. Also, semiconductors require a certain threshold voltage to function. Hence, eventually, the voltage could not be decreased any further. Keeping the voltage constant in the above reasoning about the consequences of gate length scaling has the effect that, since the energy is proportional to the *square* of the voltage, the power is now increased by a factor of 4. The power that can be reasonably handled in consumer chips (e.g., due to cooling constraints), is around 80 W to 120 W, which is the reason for choosing not to scale processor frequencies any further. The green line in Fig. 2.1 visualizes the exponential increase in clock frequency until 2004, after which the curve visibly flattens out. Current processor clock rates stagnate at clock frequencies of around 2–3 GHz. The processor with the highest clock frequency ever sold commercially, IBM's z196 found in the zEnterprise System, runs at 5.2 GHz. Intel's cancellation of the Tejas and Jayhawk architectures [61] in 2004 is often quoted as the end of the frequency scaling era.

After clock frequencies stopped scaling, in a couple of years transistor gate length scaling in silicon-based semiconductors will necessarily come to a halt as well. Silicon has a lattice constant of 0.543 nm, and it will not be possible to go much further beyond the 11 nm technology node depicted in Fig. 2.1 — which is predicted for 2022 by the International Roadmap for Semiconductors [63] and even as early as for 2015 by Intel [88] — since transistors must be at least a few atoms wide.

Yet, Moore's Law is still alive thanks to technological advances. Fabrication process shrinks have benefited from advances in semiconductor engineering such as Intel's Hafnium-based high-κ metal gate silicon technology applied in Intel's 45 nm and 32 nm fabrication processes [84]. Technological advances, such as Intel's three-dimensional *Tri-Gate* transistors [35], which will be used in the 22 nm technology node, are a way to secure the continuation of the tradition of Moore's Law. Other ideas in semiconductor research include graphene-based transistors [102, 146], which have a cut-off frequency around $3\times$ higher than the cut-off frequency of silicon-based transistors; or replacing transistors by novel components such as memristors [151]; or eventually moving away from using electrons towards using photons.

The era of the frequency scaling has allowed sequential processors to become increasingly faster, and the additionally available transistors were used to implement sophisticated hardware logic such as out-of-

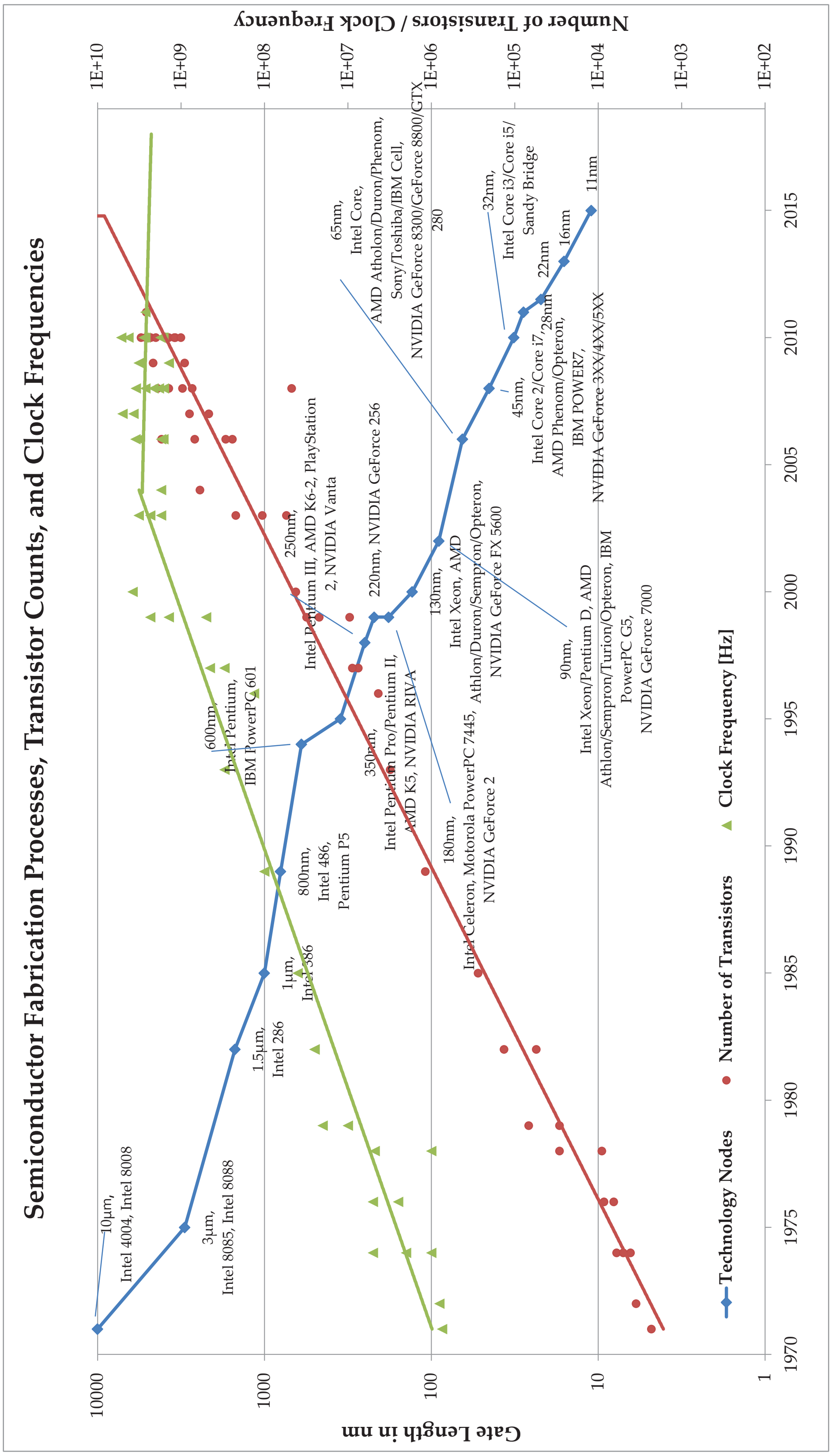


Figure 2.1: *Development of technology nodes, transistor counts, and clock frequencies. Note the logarithmic scales. Data sources: [87, 174, 175].*

order execution, Hyper Threading, and branch prediction. Hardware optimizations for a sequential programming interface were exploited to a maximum. To sustain the exponential performance growth today and in the future, the development has to be away from these complex designs, in which overhead of control hardware outweighs the actual compute engines. Instead, the available transistors have to be used for compute cores working in parallel. Indeed, the end of frequency scaling simultaneously was the beginning of the multicore era. Parallelism is no longer only hidden by the hardware, such as in instruction level parallelism, but is now exposing explicitly to the software interface. Current industry trends strive towards the manycore paradigm, i.e., towards integrating many relatively simple and small cores on one die.

The end of frequency scaling has also brought back co-processors or accelerators. Graphics processing units (GPUs), which are massively parallel compute engines and, in fact, manycore processors, have become popular for general-purpose computing. Intel's recent Many Integrated Core (MIC) architecture [35] follows this trend, as do the designs of many microprocessor vendors such as adapteva, Clearspeed, Convey, tilera, Tensilica, etc.

There are a number of reasons why many smaller cores are favored over less and bigger ones [9]. Obviously, a parallel program is required to take advantage of the increased explicit parallelism, but assuming that a parallel code already exists, the performance-per-chip area ratio is increased. Addressing the power consumption concern, many small cores allow more flexibility in dynamic voltage scaling due to the finer granularity. The finer granularity also makes it easier to add redundant cores which can either take over when others fail, or redundant cores can be utilized as a means to maximize silicon waver yield: if two cores out of eight are not functional due to fabrication failures, the die still can be sold as a six-core chip: e.g., the Cell processor in Sony's PlayStation3 is in fact a nine-core chip, but one (possibly not functional) core is disabled to reduce production costs. Lastly, smaller cores are also easier to design and verify.

Today, throughput-optimized manycore processors are implemented as external accelerators (GPUs, Intel's MIC), but eventually the designs will be merged together into a single heterogeneous chip including traditional "heavy" latency-optimized cores and many light-weight throughput-optimized cores. A recent example for such a design was the Cell Broadband Engine Architecture [76] jointly developed by Sony, Toshiba, and

IBM. The main motivation for these heterogeneous designs are their energy efficiency. Going further toward energy efficient designs, special-purpose cores might be included, which are tailored to specific class of algorithms (signal processing, cryptography, etc.) or which can be reconfigured at runtime, much like Convey's hybrid-core computers, into which algorithm classes ("Personalities") are loaded at runtime and emerge in hardware.

Memory remains the major concern in moving towards an exa-scale system. While microprocessor compute performance used to double every 18–24 months, memory technology evolved, too, but could not keep up at this pace. The consequence is what is often called the *memory gap*: the latency between a main memory request and the request being served has grown into an order of hundreds of processor cycles. Equally, memory bandwidth has not increased proportionally to compute performance. For a balanced computation, several tens of operations on one datum are required. The consequence is that many important scientific compute kernels (including stencil computations, sparse equation system solvers and algorithms on sparse graphs) have become severely bandwidth limited. A hierarchy of caches, i.e., a hierarchy of successively smaller, but faster memories, mitigate this problem to some extent, assuming that data can be reused after bringing them to the processing elements. The memory gap also has a new interpretation in the light of energy efficiency. Data movement is expensive in terms of energy, and more so the farther away the data has to be transferred from. Data transfers have become more expensive than floating point operations. Therefore, data locality not only has to be taken seriously because of the impact on performance, but also as an energy concern.

Improvements in the near future include DDR4 modules, which offer higher bandwidth, yet have lower power consumption, higher memory density and a resilient interface to prevent errors. More interestingly, *Hybrid Memory Cubes* are a major advance in memory technology, i.e., stacked 3D memory cubes with yet higher bandwidth, lower power consumption, and higher memory density. Other ideas are in the area of bridging the gap between DRAM and hard disk drives by means of flash-type non-volatile memories, thereby addressing application fault tolerance; check-pointing to non-volatile semiconductor-based memory will be a lot faster than check-pointing to hard disks, and therefore could substantially speed up scientific applications, which depend on fault tolerance.

The third pillar in high performance computing hardware are interconnects. The increasing demand for bandwidth has led to increasing data rates. Electrical transmission suffers from frequency-dependent attenuation (the attenuation increases as the frequency is raised), limiting both the frequency and the cable length. Thus, electrical cables are gradually being replaced by optical interconnects (e.g., active optical cables, which convert electrical signals to optical ones for transmission and back to electrical ones so that they can be used as a seamless replacements for copper cables [178]). Not only can optical interconnects tolerate higher data rates, but they are also around one order of magnitude more power efficient [24].

As thousands of nodes need to be connected, it is not practical to use central switches. Instead, hierarchical structures can be used. On the other hand, finding good network topologies is a concern, as a substructure of the network is used for a parallel application running on a part of a supercomputer, and thus a good mapping between the application's communication requirements and the actual hardware interconnect has to be set up so as to avert performance deterioration. The current top 1 supercomputer (as of June 2011), the K computer installed at the RIKEN Advanced Institute for Computational Science in Japan [161], employs the "Tofu" interconnect, a 6D mesh topology with 10 links per node, into which 3D tori can be embedded. In fact, whenever a job is allocated on the machine, it is offered a 3D torus topology [4].

The consequence of substantially increasing explicit on-chip parallelism is profound. Inevitably, it needs to be embraced in order for applications to benefit from the increased total performance the hardware has to offer. Simultaneously, both network and memory bandwidth per Flop will drop and the memory capacity per compute unit will drop. This means that data can no longer be scaled up exponentially in size, and the work per compute unit decreases as the explicit parallelism increases exponentially. An exa-scale machine is expected to have a total number of cores in the order of 10^8 to 10^9, and, therefore, likely a thousand-way on-chip parallelism.

These issues must be somehow addressed in software. Most urgently, the question must be answered how this amount of parallelism can be handled efficiently. This is the challenge of programmability; other challenges are minimizing communication and increasing data locality — which in the long run means that a way must be found of expressing data locality in a parallel programming language, either implicitly or explicitly — and, lastly, fault tolerance and resilience.

Chapter 3

Software Challenges

> It must be evident how multifarious and
> how mutually complicated are the
> considerations which the working of such an
> engine involve. There are frequently several
> distinct sets of effects going on
> simultaneously; all in a manner independent
> of each other, and yet to a greater or less
> degree exercising a mutual influence.
>
> — Ada Lovelace (1815–1852)

As described in the previous chapter, there are currently two trends in the evolution of hardware architectures: The hardware industry has embraced the manycore paradigm, which means that explicit parallelism is increasing constantly. On the other hand, systems will become more heterogeneous: processing elements specialized for a specific task are far more power efficient than general purpose processors with the same performance.

These trends necessarily need to be reflected in software. High performance computing has been dealing with parallelism almost from the start. Yet, in a way, parallelism was simpler when it "only" had to deal with homogeneous unicore processors. Having entered the multi- and manycore era, not only do we have to address parallelism in desktop computing, but this new kind of on-chip parallelism also needs to be reflected in how we program supercomputers: now not only inter-node parallelism has to be taken care of, but also the many-way explicit finer-grained intra-node parallelism has to be exploited. Also, the specialization of processor components obviously entails specialization at the software level.

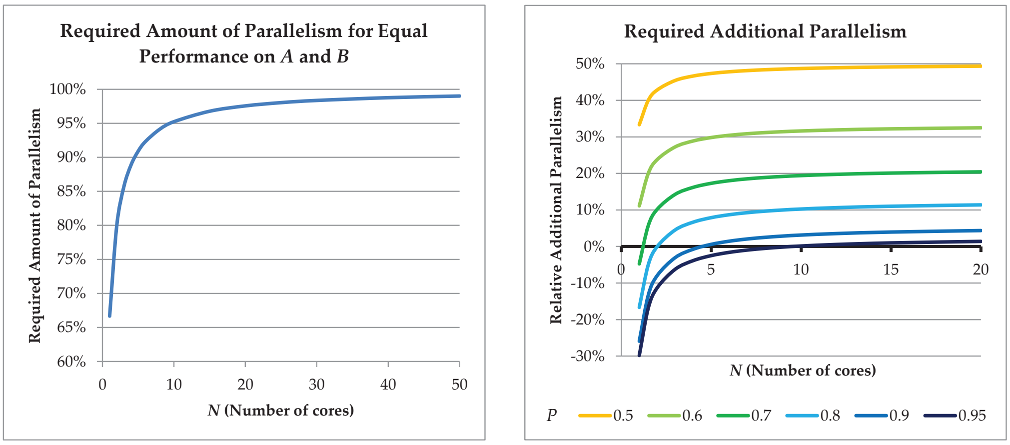
With the massive parallelism promised in the near future, several issues become critical, which must be addressed in software. They will eventually also influence programming models. Synchronization needs to be controllable in a fine-grained manner; frequent global synchronizations become inhibitingly expensive. Data locality becomes increasingly important, calling for control over the memory hierarchy and for communication-reducing and communication-avoiding algorithms. In large parallel systems, statistically a larger absolute number of failures will occur, which must be addressed by fault-tolerant and resilient algorithms.

3.1 The Laws of Amdahl and Gustafson

In the following we give a theoretical example of the impact on a fixed problem when the amount of explicit parallelism is increased. Assume we are given two hypothetical processors, A and B. Let A be a processor with medium-sized cores and B a processor with small cores. We assume that both processors consume the same amount of power; let A have N cores running at a clock frequency of f, and let B have $4N$ cores running at half the frequency, $f/2$, and assume that the reduction in control logic reduces the power consumption of each small core by a factor of 2 compared to the medium-sized cores of processor A.

As the industry trend moves towards processors of type B rather than A as outlined in Chapter 2, now we can ask: what are the implications for software when replacing processor A by processor B? How much parallelism does a program need so that it runs equally fast on both processors? Given an amount of parallelism in the program, do we need to increase it so that we can take advantage of the increased parallelism offered by processor B running at a slower speed?

Let P denote the percentage of parallelism in the program under consideration, $0 < P \leqslant 1$. In view of Amdahl's law [6], which states that, for a fixed problem, the speedup on a parallel machine with N equal com-



(a) (b)

Figure 3.1: *Required amount of parallelism and additionally required parallelism when switching from faster, heavier cores to more cores, which are more light weight and slower.*

pute entities is limited by

$$S_{\text{strong}}(P, N) = \frac{1}{\frac{P}{N} + (1 - P)},$$

in order to achieve identical speedups S_A and S_B for both processors A and B in our hypothetical setup, we have

$$S_A = \frac{2}{\frac{P}{N} + (1 - P)} = \frac{1}{\frac{P}{4N} + (1 - P)} = S_B,$$

or, after solving for P,

$$P(N) = \frac{2N}{2N + 1}.$$

The graph of this function is shown in Fig. 3.1 (a). If, for some fixed N, the amount of parallelism P is smaller than indicated by the curve in Fig. 3.1 (a), the program will run slower on processor B than on processor A. Note that in our setup already for N as small as 5, 90% of the program needs to be parallelized. Fig. 3.1 (b) shows the amount of parallelism that has to be added relative to the already existing amount, which is given by

$$q(P, N) = \frac{2N(P - 1) + P}{(1 - 4N)P}.$$

In general, if the clock frequency of A is ρ times faster than the one of B, but B has c times more cores than A, The amount of parallelism

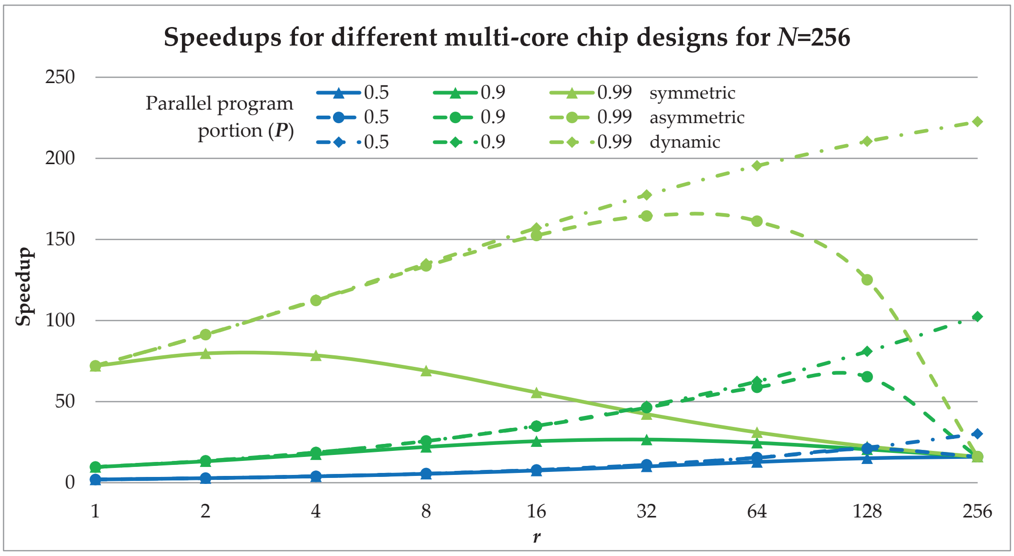


Figure 3.2: *Amdahl's law for symmetric (solid lines), asymmetric (dashed lines), and dynamic (dashed-dotted lines) multicore chips with* 256 *base core equivalents due to Hill and Marty [81] for three different quantities of program parallelism.*

required of a program so that it runs equally fast on both A and B is given by

$$P(N, c, \rho) = \frac{1}{1 + \frac{c-\rho}{(\rho-1)cN}},$$

and the relative amount of parallelism by which a program has to be increased to run equally fast is given by

$$q(P, N, c, \rho) = \frac{(\rho - 1)cN(P - 1) + (c - \rho)P}{\rho(1 - cN)P}.$$

Assuming that more powerful cores can be built given the necessary resources, we can ask how the ideal multi- or manycore chip should look like, given the parallel portion of a program. Hill and Marty take this viewing angle [81], again assuming the simple model of Amdahl's law. The chip designer is given N *base core equivalents*, each with the normalized performance 1. We also assume that r base core equivalents can be fused into a $\Phi(r)$-times more powerful core, i.e., a core which speeds a sequential workload up by a factor of $\Phi(r)$. Φ is assumed to be a sublinear function; if it were linear or super-linear, combining would always be beneficial.

In the simplest symmetric setting, all of the N base core equivalents are combined into equally large cores of r base core equivalents each, thus resulting in a device of $\frac{N}{r}$ cores. Then, as all cores are $\Phi(r)$-times more efficient than a base core equivalent, the speedup is given by

$$S_{\text{symmetric}}(P, N, r) = \frac{1}{\frac{P}{\Phi(r)\frac{N}{r}} + \frac{1-P}{\Phi(r)}}.$$

In an asymmetric — or heterogeneous — setting, Hill and Marty assume that there are small cores, one base core equivalent each, and one larger $\Phi(r)$-times more powerful core of r base core equivalents. Then, assuming the sequential part is executed by the larger core and the parallel portion by both the larger and all the $N - r$ small cores, the speedup becomes

$$S_{\text{asymmetric}}(P, N, r) = \frac{1}{\frac{P}{1 \cdot \Phi(r) + (N-r) \cdot 1} + \frac{1-P}{\Phi(r)}}.$$

Furthermore, if the cores could be dynamically reconfigured to become one larger core of r base core equivalents for sequential execution or N small cores for parallel execution, the speedup in this dynamic setting is

$$S_{\text{dynamic}}(P, N, r) = \frac{1}{\frac{P}{N} + \frac{1-P}{\Phi(r)}}.$$

Fig. 3.2 shows speedup curves for both the symmetric, asymmetric, and dynamic designs for $N = 256$ base core equivalents. As in [81], the graphs show the behavior for $\Phi(r) := \sqrt{r}$, which is sub-linear for $r \geqslant 1$. The colors in the figure encode the parallelism quantity: blue for $P = 0.5$, dark green for $P = 0.9$, and light green for $P = 0.99$. The speedup is plotted on the vertical axis; on the horizontal axis the number of base core equivalents r is varied. Thus, in the symmetric case, to the left (for $r = 1$) all cores are small, and to the right (for $r = 256$) the number for one large core consisting of 256 base core equivalents is given. Similarly, in the asymmetric and dynamic cases, the size of the one larger core increases from left to right.

The most striking results, which can be inferred from the graphs, apart from the fact that in any case the amount of parallelism is crucial, is that too many small cores are sub-optimal; that the asymmetric configuration leads to greater speedups than the symmetric configuration in any case; and that the dynamic configuration is most beneficial. The sub-optimality of many small cores is also highlighted by Fig. 3.3. It shows

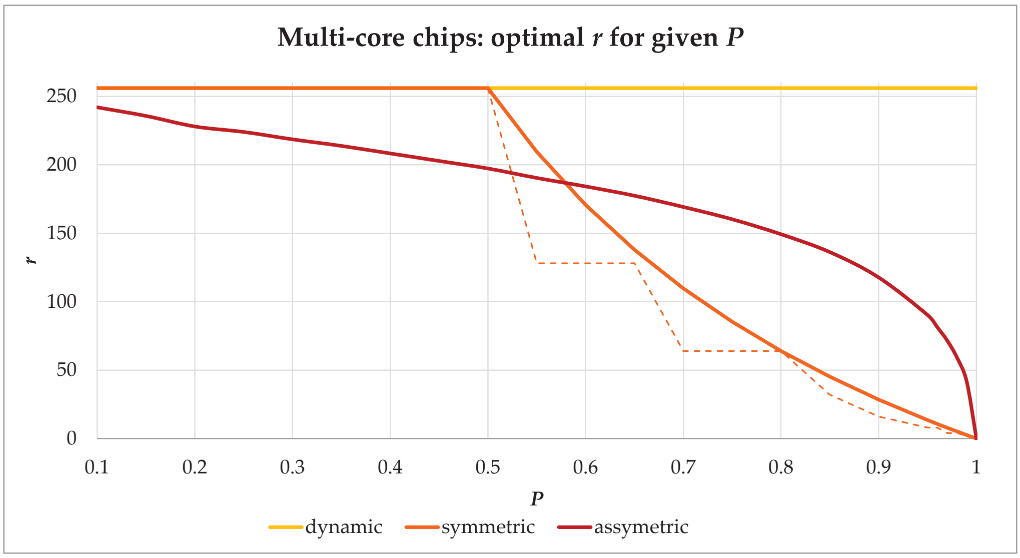


Figure 3.3: *Optimal choices for the number of base core equivalents (r) for program parallelism percentages P in Hill's and Marty's model.*

the optimal number of base core equivalents r to be coalesced into the larger cores in the symmetric, asymmetric, and dynamic scenarios for a given parallelism percentage P. In the symmetric setting (the orange curve in Fig. 3.3), if less than 50% of the program is (perfectly) parallelized, the model favors one large core ($r = 256$). The dashed orange line shows the number of base core equivalents per core that partition the resources without reminder. According to this line, up to $P = 65\%$ 2 cores, up to $P = 80\%$ 4 cores, etc. are required. In the asymmetric setting, the required size of the one large core decreases only slowly as P increases, and only drops sharply, favoring small cores, if P is close to 1. The dynamic configuration is designed to adapt to sequential and parallel regions; thus, clearly, the maximum speedup is reached in a parallel region when there are many small cores and in a sequential region when there is one core which is as powerful as possible, i.e., for maximal r, as conveyed by Figs. 3.2 and 3.3.

Gustafson put the pessimistic prospects of Amdahl's law into perspective [78]. He argued that rather than fixing the problem instance, the time to solve the problem should be fixed: in practice, when scaling a parallel program up on a large parallel machine with many nodes, the problem size is scaled up simultaneously. This scenario is commonly referred to as *weak scaling*, whereas scaling out a fixed problem on a parallel machine is referred to as *strong scaling*. In the weak scaling setting, let the time required to solve a problem with a percentage P of parallelism be 1 on a parallel machine with N compute entities. Then if the same program is executed sequentially, the time required is $N \cdot P + (1 - P)$, as the parallel part takes N times longer to execute. Thus, the speedup is

$$S_{\text{weak}}(P, N) = \frac{NP + (1 - P)}{1} = N + (1 - N)(1 - P),$$

which is commonly called the law of Gustafson-Barsis.

Traditionally, weak scaling was indeed what was applied in practice: larger machines enabled solving larger problems. However, with the dramatic increase of explicit on-chip parallelism and the consequential decrease of per-core memory we are facing today and in the near future, we are inevitably forced to leave the area of weak scaling and gradually forced into the realm of strong scaling governed by Amdahl's law.

With his law, Amdahl made a case against massive parallelism; Gustafson could relativize it by assuming that the problem size grew along with the available parallelism. Today we are challenged by having to embrace massive parallelism, but we are no longer in the position in which we can make use of Gustafson's loophole.

Amdahl's law is a model in which simplifying assumptions are made. It may give a speedup estimate for an upper bound of the speedup, yet the relative amount of parallelism is hardly quantifiable precisely in practice, so Amdahl's law gives rather a qualitative than a quantitative assessment. Furthermore, it might have to be applied to code section rather to a whole program, as parallelism can change dynamically. Platforms become more and more heterogeneous, which implies that there can be many types of explicit parallelism. For instance, a trend today is to *hybridize* MPI codes by inserting OpenMP pragmas so that the software takes advantage of the hardware's architectural features. Thus, there is a level of coarse-grained parallelism from the MPI parallelization, as well as the more fine-grained OpenMP concurrency layer, which is used, e.g., to explicitly extract loop-level parallelism — a typical application of OpenMP. Another trend is to deliberately enter the regime of heterogeneity by using hardware accelerators: for instance, general purpose GPU computing enjoys a constantly gaining popularity. In fact, 3 of the current world's fastest computers are equipped with GPU accelerators [161]. In this case there are even more levels of explicit parallelism.

Thus, the question remains how the additionally required parallelism can be extracted from a program. It can be accepted for a fact that compilers have failed at auto-parallelization. Certainly, compilers *can* vectorize or automatically parallelize certain types of loops. In fact, automatic loop parallelizers such are Cetus [157] or PLuTo [23] are able to automatically insert an OpenMP pragma at the right place into the loop nest of a stencil computation. But a compiler typically is not able to extract additional parallelism or reduce the bandwidth usage or the synchronization overhead, e.g., by automatically applying one of the algorithms, which will be discussed in Chapter 6, given a naïve implementation of a stencil calculation.

Mostly, the inhibiting factor for parallelization lies in fact within the algorithmic design. Coming from a long tradition of sequential computing, algorithms still have sequential semantics, and therefore intuitively are implemented inherently sequentially. Obviously, a compiler can typically not, or only to a very limited extent, make such an algorithm “more parallel” by applying loop transformations. Undoubtedly, most of the compute time is spent in loops, which is a reason why a lot of research has focused on understanding loops. Unfortunately, loop-level parallelism is not sufficient: not enough parallelism might be exposed or it might have the wrong granularity, thus, e.g., incurring a high synchronization overhead and ultimately result in a slow-down instead of a speedup. Typically, a parallel version of an algorithm is in fact a radically different algorithm; we will give a concrete example in Chapter 4.

History teaches us that we must embrace parallelism rather than fight it, even more so as, having forcibly left the frequency scaling era, parallelism has started to permeate “consumer computing”: not seldom today, desktop and laptop computers have CPUs with four or even more cores, and general purpose-programmable GPUs are omnipresent. This inevitably leads to the question how to program these devices.

3.2 Current De-Facto Standards for Parallel Programming Models

Historically, high performance computing has been concerned with parallel processing since the 1960s. In contrast, desktop computing was traditionally typically sequential (in the sense that algorithms were implemented sequentially), at least until the beginning of the multicore commodity CPU era in 2004, the year of Intel’s cancellation of their Tejas and Jayhawk architectures, which is often quoted as the end of the frequency scaling era and therefore the rise of the multicores.

Surprisingly, despite the long history of parallel computing in high performance computing, the programming languages used in both areas are not much different in style. Notably, C/C++ as examples for languages used in both areas, they are inherently sequential languages: there are no language constructs for parallel execution[*]. Parallelism is only offered through external libraries, typically the Message Passing Interface (MPI) and threading libraries such as pthreads (POSIX threads) or the Microsoft Windows threading API. In fact, the notion of parallelism has been known in desktop computing for a while; multi-threading is used to carry out (sequential) compute-intensive operations in the background, which is the natural application of the task parallel model offered by these threading libraries. The tight interaction of threads required when decomposing an algorithm into parallel parts as required in typical high performance computing tasks, however, can only be done at a high programming effort. For instance, data-level parallelism or loop-level parallelism is cumbersome to implement with pthreads, Java threads, or even Java's concurrency library, since the programmer has to take care of subdividing the data or the iteration space manually. Unless additional libraries, such as Intel's Threading Building Blocks, are used on top of the threading libraries, the programmer is responsible for assigning the work to threads and for doing the load balancing. Decomposing data volumes, for instance, involves easy-to-miss translations of thread IDs to index spaces. Furthermore, parallel programming requires constructs for communication and synchronization, which are often not self-explanatory to the parallel programming novice (such as a *future* in Java). This leads to a steep learning curve and is a reason why parallel programming is said to be difficult and why parallel programming has not been taught in programming classes. In view of today's hardware, sequential programming should really be considered a special case of parallel programming instead of the other way around.

Both message passing libraries and threading libraries are typically programmed in a way that make many instances of the same program execute the same instructions, but on different sets of data. This programming model is called the *single program multiple data* (SPMD) model. It is currently the most common programming model; almost all MPI

---

[*] Fortran offers a data-level parallel notation in the form of vectors, which is one form of parallelism. In Java, threads are part of the runtime system, and Java offers the `synchronized` keyword. These concepts target another form of parallelism: task-level parallelism.

codes are written in this fashion, and it extends to more modern parallel languages, such as the ones in the PGAS family (cf. Chapter 3.3). The greatest shortcoming of the SPMD model is that it obfuscates the structure of an algorithm by splitting it into program fragments and forcing the programmer to add the required management overhead. The model also makes it hard to express nested parallelism, which might occur naturally, e.g., in divide-and-conquer algorithms.

SPMD is a simple model in the sense that there are few primitives for communication and synchronization and it offers high execution model transparency. Although MPI offers an abundance of communication styles and functions, they could be emulated using *sends* and *receives*. As for the execution model transparency, the programmer knows exactly by design how the data is distributed and where code is executed. The simplicity also extends to the compilation and deployment process: a standard sequential compiler is sufficient. However, the simplicity of the model comes at a high overhead for the programmer, placing the burdens of fragmented style programming and the error-prone bookkeeping overhead that is associated with it on her or him.

So far, MPI [111] is most prevalent programming paradigm for programming distributed memory architectures. It has become the de-facto standard for these types of architectures. MPI was conceived in the early 1990s by a consortium from academia and industry. The first MPI standard was released in 1994; currently the MPI forum is working on the third version of the standard. The fact that many reliable and freely available implementations for a multitude of platforms and many language bindings (C/C++, Fortran, Java, Python, Perl, R) exist and the fact that it is a well specified standard have significantly contributed to the success of MPI.

A typical MPI code runs as many program instances simultaneously as there are hardware resources (in concordance with the SPMD model); thus parallelism is expressed at program granularity. Communication and synchronization are done by calls to the MPI library. Communication is typically two-sided, i.e., one process issues a *send* to the receiver process, which must call the *recv* function, lest the program hangs up in a dead lock.

Today, OpenMP is a popular choice to utilize thread-level parallelism. For shared memory computer systems and for intra-node concurrency, the OpenMP API [127] has become the de-facto standard. OpenMP is a set of compiler directives rather than providing language level constructs, or providing support for concurrency by means of a library. As such, while not being tied to a programming language, it depends on the compiler supporting it. The programming paradigm is quite successful, and it has been adapted in many C/C++ and Fortran compilers. Because OpenMP instructions are directives or *pragmas*, an OpenMP-instrumented program can still be compiled to a correct *sequential* program, even if a compiler does not support OpenMP. It therefore also allows the programmer to parallelize a code incrementally. The beauty of OpenMP is that, in contrast to the fragmenting SPMD style, it gives the programmer a *global view* of a parallel algorithm. OpenMP can be used to express both loop-level and task-level parallelism. Its major shortcoming is that it is designed for a shared memory environment; as commonly known, shared memory platforms do not scale above a certain number of processors. It does not offer as fine-grained (and therefore, low-level) control over threads as could be done with pthreads or operating system-specific functions, such as setting thread priorities, and it offers no sophisticated synchronization constructs such as semaphores or barriers involving subgroups of threads. Instead, the thread-level details are taken care of by the compiler and the OpenMP runtime system, and therefore also relieve the programmer of tedious bookkeeping tasks.

OpenMP has been developed since 1997 by a broad architecture review board with members from industry (hardware and compiler vendors) and academia. Its shortcomings of being restricted to the shared memory domain have been addressed in research as source-to-source translation from OpenMP to distributed memory and accelerator programming paradigms, specifically to SHMEM [96], MPI [16], and CUDA [97]. (CUDA C [123] is the model for programming NVIDIA GPUs; also a programming model in the SMPD style.)

In high performance computing programming, the heterogeneity of the hardware platforms is reflected in the current trend to mix programming models. The different levels of explicit parallelism (inter-node, intra-node, or even accelerators) are addressed by using hybrid programming paradigms, most commonly in the form of MPI+OpenMP or, with accelerators such as GPUs becoming more popular, e.g., MPI+CUDA or even MPI+OpenMP+CUDA. The different programming models are used to address the different levels or types of parallelism. Typically, MPI is used for the coarse-grained inter-node parallelism and OpenMP for the fine-grained intra-node and on-chip parallelism, such as to express loop-level parallelism.

3.3 Beyond MPI and OpenMP

One way to make parallel programming accessible to a broader public is by promoting new languages explicitly targeted at parallel programming. The holy grail of parallel programming is no longer to find a good parallel implementation for a given good sequential implementation of a program automatically, but to provide a language and a compiler that bridges the unavoidable tensions between generality, programmability, productivity, and performance. Obviously, a language targeted at parallel programming has to allow a natural expression of various flavors of parallelism, preferably in a non-obscuring global view sort of way, thus allowing concise formulations of algorithms. Parallel programmability means to provide useful abstractions of parallel concepts, i.e., the way concurrency is expressed, data is distributed, and synchronization and communication are programmed. Furthermore, abstraction should not come at the cost of an opaque execution model, giving the programmer no fine-grained control of the mapping between the software and the hardware, and thus ultimately limiting the performance by potentially ignoring native architectural features a hardware platform has to offer. Ideally, a language should also be portable across multiple flavors of hardware architectures. For instance, OpenCL [95] tries to bridge the chasm between latency-optimized multicore designs (CPUs) and throughput-optimized manycore architectures (GPU-like devices). While it works in principle, in practice, a programmer striving for good performance (i.e., efficient use of the hardware), still has to carry out hardware-specific optimizations manually.

A number of approaches towards productivity in parallel computing has been proposed. One idea is to reuse sequential program components and to orchestrate data movement between them on a high abstraction level using a coordination language such as ALWAN [28, 79], which was developed at the University of Basel.

A recent trend aiming at productivity are languages in the *partitioned global address space* (PGAS) family. The latest attempt at creating new parallel productivity languages are the projects developed and funded within DARPA's[†] High Productivity Computing Systems (HPCS) program. The parallel languages which have emerged from there are Chapel

---

[†] DARPA is the Defense Advanced Research Projects Agency, an agency of the United States Department of Defense responsible for the development of new technology for military use.

(by Cray), X10 (by IBM), and Fortress (by Sun/Oracle). However, time will tell whether these new languages are adopted by the community and could even eventually replace traditional and established languages such as C/C++, Fortran, and Java, which have matured over the years; many parallel languages have come and silently vanished again.

The underlying idea of PGAS languages, including Unified Parallel C (UPC), Co-Array Fortran, and Titanium is to model the memory as one global address space, which is physically subdivided into portions local to one process. Any portion of the data can be accessed by any process, but an access time penalty is incurred if non-local data is read or written. This concept obviously increases productivity, relieving the programmer of the need of explicit communication calls; yet, both UPC, Co-Array Fortran, and Titanium are still languages in the SPMD model, thus providing no global view. The data is (more or less, depending on the language) partitioned implicitly, while the algorithm still has to be decomposed by the programmer in an explicit way.

Co-Array Fortran [122] is an extension to Fortran which introduces a new, special array dimension for referencing an array across multiple instances of an SPMD program. Such an array is called a *co-array*. The array size in the local dimensions typically depends on the number of SPMD instances, hence obstructing the global view.

UPC [45] is a C-like language, which supports the PGAS idea by the `shared` keyword, which causes arrays to be distributed automatically in a cyclic or block-cyclic way over the SPMD instances. The programmer is given some control over the data distribution by the possibility to specify block sizes. UPC provides a more global view via a *for-all* construct with affinity control, i.e., a mechanism to distribute iterations among program instances. The idea of the affinity control is to minimize the (implicit) communication by matching the distributed iteration space to the data subdivision. Data locality is managed by distinct pointer specifiers identifying global or local memory portions as a part of UPC's type system, which enables the compiler to reason about locality statically. Except for the *for-all* construct, UPC is a language in the SPMD model, which makes it a mix between SPMD and a global view approach.

Finally, Titanium [187] is a Java-like language developed at Berkeley with extensions for the SPMD model (affinity control, and keywords for synchronization and communication). Its main goals are performance, safety, and expressiveness. There is no Java Virtual Machine; instead, Titanium compiles to C code and therefore only a subset of Java's runtime and utility classes, and also only a subset of Java's language constructs are supported.

The two languages developed within the DARPA HPCS program, Chapel [32] and X10 [83], are explicitly targeted at productivity. (Fortress was dropped from the HPCS project in November 2006.) Productivity is not quantifiable (certainly the number of lines of a program is not a productivity measure), but it encompasses the notions of performance, programmability, portability, and robustness. The grand challenge for these undertakings was "deliver[ing] 10× improvement in HPC application development productivity over today's systems by 2010, while delivering acceptable performance on large-scale systems" [145].

Chapel's two main design goals were to address productivity through a global view model by supporting general parallelism. Traditionally, there have been languages and libraries geared towards data parallelism (High Performance Fortran, ZPL [33], Intel's Ct, which is now Intel's Array Building Blocks, etc.) and others geared towards task parallelism (Charm++, Cilk, Intel Threading Building Blocks, etc.). One of the goals of Chapel is to support both types of parallelism naturally.

Chapel's data parallel features were most influenced by High Performance Fortran, the Cray MTA™ and XMT™ language extensions, and by ZPL, an array-based parallel language, supporting global view computation on distributed arrays, and therefore limited to data parallel computations.

In Chapel, data parallelism uses *domains*, which define the size and shape of the data arrays upon which a computation operates, and supports the parallel iteration by a *for-all* construct iterating over a domain. Task parallelism is introduced as *anonymous threads* by the *cobegin* language construct rather than an explicit fork-join.

Parallelism is understood as *may* parallelism rather than *must* parallelism. In this way arbitrary nesting is supported without swamping the system with too many threads, and sophisticated mechanisms for load balancing such as work stealing can be applied.

The global view model allows algorithms and data structures to be expressed in their entirety and thereby allows them to be expressed in a shorter and conciser form. Thus, they become them less error-prone to program more maintainable, as overhead cluttering the actual computation is rendered unnecessary. The program starts as one logical thread of execution and concurrency is spawned on demand by means of data or task parallel language constructs, i.e., *for-all* loops or *cobegin* constructs.

As a modern programming language, Chapel supports object orientation, generic programming, and managed memory. Its syntax is deliberately distinct from programming languages which have previously existed, but it is inspired by C, Java, Fortran, Modula, and Ada.

X10 is a purely object oriented language (also primitive data types are treated as classes, but X10 distinguishes reference and value classes) and follows the PGAS paradigm, extending it to a globally asynchronous, locally synchronous model. The syntax is reminiscent of C++, Java, and Scala, also allowing constructs known from functional programming languages. As in Chapel, threads are spawned anonymously — in X10 by the `async` keyword, — which runs a lightweight *activity* at a certain *place*. Places are an abstraction for a shared-memory computational context [34]. They can be therefore thought of as abstractions of shared-memory multi-processors. The global view for data parallelism is supported via *distributions*: mappings from regions, i.e., index sets, to subsets of places. Arrays are mappings from distributions to a base type, thereby implementing a general multi-dimensional array concept, inspired by ZPL [33]. Fine-grained synchronization constructs are provided by *clocks* and atomic sections. Also, reading from remote locations has to be done asynchronously within an activity; such an activity is left a *future*, which will receive the result. X10 forces the programmer to manage data distribution explicitly in the belief that data locality and distribution cannot be hidden in favor of a transparent execution model and good performance.

The "traditional" PGAS languages as well as Chapel and X10 address *horizontal* data locality. As mentioned in Chapter 2, *vertical* data locality, i.e., the data locality of a datum within the memory hierarchy will become more and more important as the memory gap continues to diverge. Currently, there is a lot of active research in communication minimizing and avoiding algorithms, particularly for linear algebra algorithms [12, 13, 56, 75, 114]. Data transfers are expensive both in time and in energy. High memory latencies and relatively low bandwidths impact performance, and transferring data over long stretches impact performance per Watt.

However, none of the languages presented above addresses this issue. Eventually there will be a need for explicit data placement in both the horizontal and the vertical dimension. A priori, this puts yet another burden on the programmer, unless a model can be found that is both expressive enough to facilitate the implementation of (at least a certain class of) algorithms and provide a data transfer-minimizing mapping to the memory hierarchy. Sequoia [60, 140] takes steps in this direction: it provides language mechanisms to describe data movement through the memory hierarchy and to localize computation.

3.4 Optimizing Compilers

Traditionally, compilers were the magic instrument to achieve both good code performance and code portability. An optimizing compiler was supposed to take care of architecture-specific details and carry out the corresponding code optimizations, and possibly also automatically parallelize the code at the same time.

General purpose languages, such as C/C++ or Fortran, which still are the most popular languages in the high performance computing area, have an abstraction level which is high enough so that programs can be compiled to and ported across various ISAs of diverse microprocessor architectures. Code portability is provided by leaving the decision to the compiler which optimizations to perform that work well for the target architecture. (Of course, portability is broken if architecture-specific or compiler-specific optimizations are carried out by the programmer.)

However, as hardware architectures or architecture subsystems become more specialized, traditional languages (C/C++, Fortran) become insufficient or inadequate for expressing mappings of algorithms to the hardware, lest the compiler is able to perform aggressive algorithmic transformations. A recent example are GPUs, which are designed to be massively (data-) parallel architectures. Despite NVIDIA, as an example, proposes C/C++ as base language, a *C for CUDA* program translated from a C program, differs significantly from the original. Also, Fortran for CUDA requires architecture-specific modification of the original code.

Although we came to recognize that more than an optimizing compiler is needed to be successful in parallel computing, the value of an optimizing compiler and the advances in compiler technology must not be underestimated.

In this chapter we discuss some types of compiler optimizations and concentrate on loop transformations, which are in fact the basis for the stencil code generation framework PATUS proposed in part II.

Depending on the requirements of the environment, there are a number of possible objectives to optimize for. In the area of high performance computing, a popular requirement is to minimize the time to solution, so the goal of the optimization is to maximize program performance. This means: maximizing the use of computational resources, minimizing number of operations (or clock cycles spent doing the operations), and minimizing data movement, i.e., making the most efficient use of the memory bandwidth between parts of the memory hierarchy. A variant which becomes more and more relevant in high performance computing, and has always been highly relevant in embedded and mobile systems, is to optimize for performance per Watt. Embedded systems with tight memory constraints might also ask for minimizing memory usage (minimizing program size, avoid duplicating data, etc.).

An optimizing compiler has three tasks: It needs to *decide* which part of the program is subject to optimization and which transformations are applicable and contribute toward the optimization objective; it needs to *verify* that the transformation is legal, i.e., conserves the semantics of the original program; and it finally has to actually *transform* the code portion [10]. Furthermore, the scope of the optimization can be a single statement ("peephole optimization"), a code block, a loop, a perfect loop nest (i.e., a loop nest in which each loop contains only the nested loop, except for the inner-most loop), an imperfect loop nest, a function, or the compiler can apply interprocedural analysis and optimization, which is most expensive, but yields the most gain. Current state-of-the-art compilers (e.g., Intel's C/C++ compiler as of version 8 [86], GNU gcc since version 4.1 [173], the PGI compilers [112], Microsoft's C compiler [70]) support interprocedural optimization and can deliver substantial performance gains if this option is turned on.

A typical compiler optimization does not just have a benefit associated with it, but it is rather a tradeoff between two ends. An optimization might reduce computation or increase instruction level parallelism at the cost of register usage, or remove subroutine call or control overhead at the risk of incurring instruction level cache capacity misses, or increase data locality at the price of increased control overhead, or remove access conflicts by allowing increased memory usage. Also, compiler optimizations are typically dependent on each other or optimize towards opposite ends. Thus, applying optimizations independently does not guarantee that the global optimum is reached. While applying an optimization might increase the performance of a program part, applying it globally might be harmful. Pan and Eigenmann have proposed a framework, PEAK [129], to break a program into parts for which the best compiler optimization flags are determined individually by using an auto-tuning approach: the performance is measured based on a benchmark executable created from a code section and the best combination of compiler optimization flags is determined iteratively with a search method. Indeed, we believe that the auto-tuning methodology will play an increasingly important role in compiler technology.

In the following, we give a survey over common compiler optimizations. For a more detailed overview, the reader is referred to [10] or [93].

Types of Compiler Optimizations

There is range of **basic optimizations** which are always beneficial, or at least do not de-optimize the code. Such optimizations include dead and useless code removal and dead variable elimination. **Partial evaluation** type of optimizations (constant propagation, constant folding, algebraic simplification, and strength reduction) are also beneficial in any case. They can also act as enablers for other types of optimizations.

Many algorithms in practice tend to be limited in performance by the bandwidth to the memory subsystem. An optimizing compiler can apply a certain number of **memory access transformations**, which try to mitigate the problem. One of the most essential tasks in this category is register allocation, i.e., assigning variables to (the limited set of) registers such that loads and stores are minimized. Register allocation can be modeled as a graph coloring problem. Graph coloring is NP-complete, but good heuristics exist. Another aspect of memory access transformations is removing cache set conflicts, bank conflicts, or false sharing of cache lines. In current hardware architectures, e.g., shared memory of GPUs is organized in banks, and when threads try to write to the same memory bank concurrently, the accesses are serialized, resulting in a severe performance penalty. Array padding, i.e., adding "holes" to an array at the end in unit stride direction, can eliminate or at least mitigate such problems.

As most of the execution time is typically spent in loops, **loop transformations** are a more valuable type of optimizations, which can result in considerable performance gains. This is certainly true for stencil computations. In fact, our stencil code generation framework, which will be presented in Part II, is concerned with applying kinds of loop transformation techniques, which a current state-of-the-art compiler does not perform, as they are specific to stencil computations.

There are a number of goals in terms of performance improvements that one tries to achieve by transforming loops involving computations of or access to array elements. General goals of loop transformations are:

- **Increasing data locality.** One transformation doing this is *loop tiling*. The idea is to decompose the iteration space of a loop nest into small *tiles*. The goal is then to choose the tile size such that the data involved in iterating over a tile fits into cache memory and thus avoid non-compulsory transfers to and from memory further down in the memory hierarchy, thus optimizing cache usage.
  Another transformation increasing data locality is *loop interchange*: sometimes loops could be interchanged so that the innermost loop becomes the loop iterating over the unit stride dimension of an array. This maximizes cache line reuse and reduces potential TLB misses.

- **Increasing instruction level parallelism.** By replicating the loop body, *loop unrolling* potentially increases instruction level parallelism and decreases loop control overhead, but, on the downside, it increases register pressure. Loop unrolling introduces a new parameter, the number of loop body replications, which has to be chosen (in an optimizing compiler at code generation time) such that instruction level parallelism is maximized under the constraint that not too many variables are spilled into slower memory and thereby degrade the performance.
  Software pipelining helps removing pipeline stalls. The operations in one loop iteration are broken into stages $S_1(i), \dots, S_k(k)$, where i is the index in the original loop. In the pipelined loop, one iteration performs the stages $S_1(j), S_2(j-1), \dots, S_k(j-k)$. Software pipelining comes with some overhead; it requires prologue and epilogue loops (filling and draining pipeline), each of which requires $k-1$ iterations.

- **Decreasing loop control overhead.** *Loop fusion*, i.e., combining multiple loops into one, reduces loop control overhead (and potential pipeline stalls due to branch misprediction), but increases register pressure.
  Loop unrolling, as mentioned before, also decreases loop control overhead.

- **Decreasing computation overhead.** *Loop invariant code motion* moves subexpressions which are invariant with respect to the loops' index variables out of the loop so that they are computed only once. Especially array address calculations (created by either the programmer or the compiler) benefit from this optimization. However, the optimization increases register pressure.

- **Decreasing register pressure.** *Loop fission*, i.e., splitting a loop into two or more loops, decreases register pressure and can thereby avoid spilling of variables into slower memory, but the loop control overhead is increased.

- **Parallelization.** Loop parallelization comes in two flavors: vectorization, exploiting data level parallelism; and a task parallelism-based idea of parallelization, actually assigning different iterates to different units of execution.

- **Exposing parallelism.** The original loop nest might not be parallelizable. For instance, vectorization may only become applicable after a *loop interchange* transformation.
  Loop skewing is a technique that reshapes the iteration space (by introducing new loop indices depending on the original ones) so that loop carried data dependences are removed.

- **Enabling other types of loop transformations.** Given a loop nest, some loop transformation may not be legal a priori, i.e., if applied, it would change the semantics due to loop carried data dependences. Such techniques include the ones mentioned in the bullet above, as well as loop splitting and loop peeling (which is a special case of loop splitting), which break a loop into many to avoid a special first iteration.

Dependences

The basic tool for loop analysis are data dependences. Determining the legality of a loop transformation, i.e., under which circumstances, with respect to dependences, a loop transformation does not change the meaning of a program, and, as a consequence, developing tests that could recognize dependences or prove independence was an active research area from the 1970s to the 1990s [5, 14, 26, 71, 100, 101, 105, 138, 182].

Two statements S_1, S_2 are *data dependent* if they contain a variable such that they cannot be executed simultaneously due to a conflicting use of the variable[‡]. In particular, if variable written by S_1 is read by S_2 there is a *flow dependence*, if a variable in S_1 is read and written in S_2 there is an *anti-dependence*, and if both S_1 and S_2 write the same variable there is an *output dependence*.

Note that anti-dependences and output dependences are not as restrictive as flow dependences. The variable in S_2 could be replaced by another variable, which would remove the dependence.

For loop analysis, both inter- and intra-iteration dependences have to be captured. Dependences between iterations are called *loop-carried dependences*. Consider a general perfect loop nest which reads from and writes to a multi-dimensional array `a`:

Listing 3.1: *Dependences in a perfect loop nest.*

```
for $i_1$ = $l_1$ .. $u_1$
  ...
    for $i_k$ = $l_k$ .. $u_k$
$S_1$ :     a[$f_1(i_1,\dots,i_k),\dots,f_\ell(i_1,\dots,i_k)$] = ...
$S_2$ :     ... = a[$g_1(i_1,\dots,i_k),\dots,g_\ell(i_1,\dots,i_k)$]
```

Obviously, if S_2 of the iteration $(i_1,\dots,i_k) = (J_1,\dots,J_k)$ depends on S_1 in another iteration $(i_1,\dots,i_k) = (I_1,\dots,I_k)$, it must hold that

$$f_j(I_1,\dots,I_k) = g_j(J_1,\dots,J_k) \qquad \forall j = 1,\dots,\ell. \tag{3.1}$$

If it can be proven that Eqn. 3.1 has an integer solution $(\hat{i}_1,\dots,\hat{i}_k)$ such that $l_j \leqslant \hat{i}_j \leqslant u_j$ for $j = 1,\dots,k$, then there is a loop carried dependence. Conversely, if no integer solution exists, the loop carries no dependences. Hence, deciding whether there are loop carried dependences amounts to solving a system of Diophantine equations, which is an NP-complete problem. Optimizing compilers resort to restricting f and g to affine functions; if they are not, compilers err on the conservative side, and all possible dependences are assumed. Numerous number-theoretical results have been applied to prove or disprove the existence of a solution, including simple observations as: "If f_j and g_j are different constant functions for some j, then Eqn. 3.1 cannot have a solution" (in the literature this is referred to as the "ZIV test," ZIV being the acronym

---

[‡] There is also the notion of *control dependence*: two statements S_1, S_2 are *control dependent* if S_1 determines whether S_2 is executed. This type of dependence will not be discussed in this overview.

for “zero index variable”), or more sophisticated GCD, Banerjee, λ, and Omega tests [93]. The problem can also be formulated as a linear integer programming problem, or Fourier-Motzkin elimination can be used (the complexity of which is double exponential in the number of inequalities). If simple tests handling special cases fail, the optimizing compiler can gradually proceed to more complex and more general tests.

A solution to Eqn. 3.1 — should there be one — gives rise to the notions of *distance vectors* and *direction vectors*. For a solution $J = J(I) \in \mathbb{Z}^k$, the distance vector is defined as $d := J - I$. A direction vector is a symbolic representation of the signs of d.

Example 3.1: *Calculating distance vectors.*

Consider the loop nest

```
for i1 = 2 ..  n
    for i2 = 1 ..  n-1
S1 :      a[i1,i2] = a[i1-1,i2] + a[i1-1,i2+1]
```

The statement S_1 has a loop carried dependence on itself: We have the following index function for the right and left hand side, respectively:

$$f(i_1, i_2) = (i_1, i_2)$$
$$g_1(i_1, i_2) = (i_1 - 1, i_2), \quad g_2(i_1, i_2) = (i_1 - 1, i_2 + 1)$$

Solving $f(I_1, I_2) = g_1(J_1, J_2)$ gives $I_1 = J_1 - 1, I_2 = J_2$, i.e.,

$$J(I) = (I_1 + 1, I_2),$$

and hence the first distance vector is

$$d_1 = ((I_1 + 1) - I_1, I_2 - I_2) = (1, 0).$$

Similarly, solving $f(I_1, I_2) = g_2(J_1, J_2)$ yields the second distance vector

$$d_2 = (1, -1).$$

The distance vectors of the transformed loop nest give information about whether the transformation is legal: a legal distance vector d must be *lexicographically positive*, denoted $d \succ 0$, meaning that the first non-zero entry in d must be positive. (A negative component in d corresponds to a dependence on an iteration of the corresponding loop with higher iteration count. If the first non-zero component was negative, this would mean that there is a dependence on a statement in a future iteration.)

There are two general models for loop transformations. The *polyhedral model* [74], applicable to loops with affine index functions and affine loop bounds, interprets the iteration space as a polyhedron, and loop transformations correspond to operations on or affine transformations of that polyhedron. For instance, loop tiling decomposes the polyhedron into smaller sub-polyhedra. The *unimodular model* associates unimodular matrices, i.e., matrices in $GL_n(\mathbb{Z})$, with loop transformations. For instance, interchanging the two loops in a doubly nested loop would be described by the matrix $\left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$.

The beauty of the latter framework is that chaining loop transformations corresponds to simple multiplication of the matrices. The matrices also describe how distance vectors are altered under the transformation, namely simply by multiplying the distance vector by the matrix describing the transformation. Thus, legality of a transformation can be easily checked by testing the corresponding resulting vector for lexicographical positivity. However, it is less general than the polyhedral model. For instance, loop tiling cannot be described by a unimodular matrix.

Example 3.2: *Determining interchangeability of loops.*

Interchanging the loops of the loop nest in Example 3.1 is not legal, because the second distance vector d_2 would not be lexicographically positive:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ -1 \end{pmatrix} = \begin{pmatrix} -1 \\ 1 \end{pmatrix} \not\succ 0$$

Parallelization

The following theorem answers the question under which conditions a loop within a loop nest can be parallelized. The theorem is equivalent to Theorem 3.1 in [180] and [179], and a proof can be found there.

Theorem 3.1. *The j-th loop in a loop nest is parallelizable if and only if for every distance vector $d = (d_1, \ldots, d_k)$ either $d_j = 0$ or there exists $\ell < j$ such that $d_\ell > 0$.*

Example 3.3: *Determining which loops can be parallelized.*

In the loop nest in example 3.1, the outer cannot be parallelized because the first entry of both distance vectors is 1. The inner loop, however, can be parallelized because the second entry of d_1 is 0, and the second condition holds for d_2 setting $\ell = 1$.

Loop Skewing

Wolfe and Lam propose skewing as a powerful enabling transformation for parallelization [180, 179]. In fact, Wolf shows in [179] that any loop nest can be made parallelizable after applying a skewing transformation. Different hardware architectures have different requirements for the granularity of parallelism. For instance, multicore CPU systems favor coarse grained parallelism (parallel loops should be as far out as possible in a loop nest), while GPUs prefer finer grained parallelism. Skewing can transform a loop nest such that the desired granularity of parallelism is exposed. Skewing is an affine transformation of the iteration space, in the unimodular framework for a doubly nested loop described by the matrix

$$\begin{pmatrix} 1 & 0 \\ f & 1 \end{pmatrix}.$$

f is the *skewing factor* which has to be chosen such that the distance vectors of the loops to parallelize fulfill the parallelization condition in Thm. 3.1. Skewing is always legal, but the disadvantage is that the loop bounds become more complicated, as, e.g., originally rectangular iteration spaces are mapped to trapezoidal iteration spaces.

Vectorization

Current CPU architectures support a *Single Instruction Multiple Data* (SIMD) mode, which carries out an arithmetic instruction on a vector of data rather than only on a single scalar value. In contrast to vector machines, examples for which include the Cray-1 (1976) and the Japanese "Earth Simulator" supercomputer, which was number one on the TOP500 list from June 2002 until June 2004 [161], current SIMD implementations in microprocessors feature short and fixed vector lengths: current Intel and AMD architectures implement the Streaming SIMD Extensions (SSE) instruction set; SSE registers are 128 bytes wide, so that 2 double precision or 4 single precision fixed-length floating point vector operands can be operated on simultaneously. Similarly, AltiVec [113] implemented in the PowerPC processor family, has 128 byte wide SIMD registers. The newer AVX [85] instruction set, which has 256 byte wide SIMD vectors, is supported as of Intel's *Sandy Bridge* and *Ivy Bridge* processor microarchitectures released in 2011 and 2012, and in AMD's *Bulldozer* released in 2011. Manycore architectures, such as GPUs or Intel's new MIC ("Many Integrated Core") architecture tend to have wider SIMD vectors.

Vectorization is a special form of parallelization. It modifies a fixed sequence of operations which are previously performed sequentially on a set of individual data elements into a mode in which the sequence of operations is performed simultaneously on multiple data elements. An optimizing compiler tries to extract data level parallelism from a loop to leverage SIMD processing.

Loop Tiling

Loop tiling is a loop transformation that breaks the original iteration space into small *tiles*. The transformation converts a loop or a loop nest into a loop nest with doubly as many nested loops so that the iteration space is subdivided into smaller iteration subspaces (*tiles*) over which the the inner loop nest iterates, while the outer loop nest controls the iteration over the tiles. Tiling is done to improve spatial data locality for one level of the memory hierarchy, or for multiple levels if applied recursively. Tiling of a pair of loops in a loop nest is legal whenever they can be interchanged legally.

Example 3.4: *Rectangular Loop Tiling*

This example shows the result of a tiling transformation of the rectangular iteration space [`min_i`, `max_i`] × [`min_j`, `max_j`] × [`min_k`, `max_k`].

```
                       for ti=min_i..max_i by tile_i
for i=min_i..max_i       for tj=min_j..max_j by tile_j
  for j=min_j..max_j       for tk=min_k..max_k by tile_k
    for k=min_k..max_k       for i=ti..min(max_i, ti + tile_i-1)
      ...                      for j=tj..min(max_j, tj + tile_j-1)
                                 for k=tk..min(max_k, tk + tile_k-1)
                                   ...
```

Compilers try to find the best optimizations based on static reasoning.

Their real challenge is to do so without actually executing the code. There is a current research trend away from statical optimization towards automatically evaluating the effect of optimizations based on actual program execution, thus determining ideal parameterizations based on benchmarks. This automatic tuning or *auto-tuning* methodology will be discussed further in Chapter 8.

3.5 Domain Specific Languages

Another approach aiming at code and performance portability across architectures are domain specific languages (DSLs). As opposed to general purpose languages such as C/C++, Fortran, Java, or the new parallel languages such as Chapel and X10, domain specific languages are designed to express solutions in a very specific problem domain. Well-known examples for DSLs are the database query language SQL; Matlab, a language used for rapid prototyping of numerical algorithms; HTML to create websites. Akin to the current trend of specialization at the hardware level, DSLs propose specialization at the software/language level.

The effectiveness of a programming language is measured in terms of generality, productivity, and performance [108]. Generality means providing constructs rich enough so that the language can ideally be used to solve an arbitrary type of problem. Productivity means supporting idioms that are easy to use and understand and provide a high level of abstraction, e.g., through rich data types. Performance means that the language can be compiled to machine code that matches the hardware architecture and makes good use of the hardware resources.

Obviously, it is desirable to have a language that is highly effective, meeting all of these three criteria. However, such a language does not exist, as the three criteria are conflicting. Programming languages sacrifice one goal to some extent for the two others. C/C++, Fortran, Java, or even more platform-specific languages such as OpenCL are general purpose languages which let the programmer achieve good performance, but require a verbose programming style. The languages allow to express platform-specific optimizations at the cost of maintainability, reusability, portability, error-proneness, and time spent to implement the optimizations. On the other hand, scripting languages such as Python or Ruby allow the programmer to be productive by providing expressive syntax and good library support, but in general, scripts written in such languages cannot optimized for high performance. DSLs can take the role of languages sacrificing generality for productivity and performance [31].

Restricting a language to a narrow, well defined domain has several advantages [165]:

- DSLs are designed to be *expressive* and *concise*. They enable the domain specialist to express an algorithm or a problem in an idiom that is natural to the domain, typically on a high level of abstraction. This facilitates the *maintainability* of the code. It also conveys *safety*: it is easier to produce correct code. Sources of error might be prevented by the design of the language or by an automated analysis of the code.

- Running DSL programs on a new platform requires that the backend infrastructure is ported *once*. Porting programs to a new platform therefore does not require re-writing of the entire application, emphasizing *portability* and *reusability* of the code. By relying on expert knowledge of the new platform, platform-specific optimizations have to be done once, resulting also in performance portability of the application.
  As an example in particular, switching from a CPU-based system to GPUs, requires re-engineering the entire application or even re-engineering algorithms for a better match of the new hardware platform, which can be prevented by using a DSL.

- Due to the fact that the expressiveness of a DSL is restricted to a precisely defined problem domain, domain-specific knowledge enables aggressive optimizations, which can lead to a substantial increase in performance.

On the downside, the user is required to learn a new language, and the developer pays the cost of implementing and maintaining the system, which might include creating a compiler and a programming environment (IDE, debugging facilities, profilers, etc.).

There are different approaches to mitigate these disadvantages. If the DSL is compiled to a lower-level, possibly hardware-specific programming language there is no need for a dedicated compiler, and the general purpose compiler can be relied on to perform additional general optimizations. Instead of doing a source-to-source translation, another approach is to *embed* a DSL into a host language. Then, the constructs of the host language, the optimization features, and the programming environment of the host language can be reused. Also, the programmer does not need to learn an entirely new syntax. Furthermore, in the embedded DSL approach, multiple DSLs can interoperate through the host language, which enables seamless combinability of DSLs embedded into the same host language. However, the host language might not provide a direct mapping to specialized hardware. For instance, if a DSL was embedded in Java or a language that compiled to Java Bytecode, the program could not be run on a GPU.

Chafi et al. counter this problem by a concept they call *language virtualization* [31]. They call a language *virtualizable* if, among other criteria, performance on the target architecture can be guaranteed, which, as one approach, entails that the embedded program is liftable to an internal representation within the host language, i.e., the embedded program can be represented using an abstract syntax tree within the host language. From this internal representation, a performance layer code in a language supported by the target architecture can be synthesized. The language of their choice, which allows to do this, is Scala [125]. The proposed framework, Delite [31, 126], aims at simplifying DSL development targeting parallel execution. Delite compilation stages comprise virtualization, which lifts the user program into an internal representation in which domain-specific optimizations (written by DSL developer) are performed and represents operations as Delite nodes, thereby mapping them to parallel patterns such as *map*, *reduce*, *zip*, and *scan*; the Delite compilation handles Delite-specific implementation details and performs generic optimizations; and finally the domain-specific operations are mapped to the target hardware.

Liszt [31] is a DSL approach for finite element-based simulations — and therefore loosely related to the framework presented in Part II — built on top of Delite with the goal to express simulation code at a high level of abstraction, independently of the parallel target machine. The framework generates code for lower level programming models (MPI and C for CUDA).

Other approaches to bridge the gap between productivity and performance have been proposed. SEJITS [30] is an approach, which does a just-in-time compilation of an algorithm or a part of an algorithm specified in a high productivity language (such as a scripting language like Python) to an *efficiency-level* language, which maps more directly to the underlying hardware. PetaBricks [7] is a framework that tries to compose an algorithm from sub-algorithms selected by an auto-tuner and thereby guaranteeing that the final implementation is hardware-efficient and maps well to the selected hardware architecture.

3.6 Motifs

So far, we have discussed separate efforts in hardware and software/language design. Motifs provide a basis for bringing the research areas together.

Motifs were conceived in a joint effort by interdisciplinary scientists from the Lawrence Berkeley Lab and UC Berkeley [9]. They originated from an idea presented in 2004 by Phillip Colella [43], who identified *seven dwarves*, methods he believed would be important in the numerics related to physical sciences for at least the next decade. The seven dwarves are perceived as "well-defined targets from [the] algorithmic, software, and architecture standpoint" [43]. They were subsequently complemented by more dwarves in discussions with experts from various fields of computer and computational sciences. The resulting 13 motifs try to capture and categorize any type of algorithm or numerical method in computer science and computational sciences.

The main motivation for motifs is to provide a fundament for innovation in parallel computing. They can be viewed as a high-level abstraction unified framework, spanning algorithms and methods applied in the entire range from embedded computing to high performance computing, which are relevant now and in the long term. Realizing that methods may change over time, more motifs may be added as needed. The high level of abstraction allows reasoning about requirements of parallel applications without fixing and thereby limiting oneself to specific code instances solving a certain problem. Rather, freedom to explore new parallel methods and algorithms is encouraged, while knowledge from lessons learned in the past can be incorporated, e.g., which methods work well on large-scale parallel systems and which do not. Ongoing efforts aim at formalizing and systematizing the motif approach [91].

Traditionally, selecting or designing a hardware infrastructure for a software application was guided by benchmarks such as SPEC [48]. One of the hopes in the development of motifs is to facilitate the decomposition of an application into basic modules describing computation and communication aspects of the application, which is somewhat obscured in traditional benchmark suites. Conversely, the original paper [9] also maps benchmarks from embedded and general purpose computing and application domain-specific problems to motifs.

Motifs are classes of algorithms and methods characterized by their compute and data movement patterns. Often, similar parallelization techniques apply to any algorithm or method within a motif. Motif boundaries are highlighted by the fact that in some cases there are libraries encapsulating subsets of algorithms within a motif, e.g., dense linear algebra operations or fast Fourier transforms. However, a motif may also contain significantly different algorithmic approaches to a method. For instance, there are multiple algorithmic approaches to particle simulations.

Since motifs are designed to capture the algorithmic essence of an application and since they are not implementation-specific, they can be used as a measure of success for both hardware architects and (parallel) programming language or programming model designers. By identifying the motifs within an application and by evaluating motif-based benchmarks on some hardware architecture, it can be judged whether that hardware architecture is a good match for that application. In fact, in this way they can be viewed as a tool for the hardware-software co-design methodology. We also view them as basic building blocks. The work proposed in this thesis tries to provide a software infrastructure capturing a subset of the structured grid motif: the class of stencil computations.

In the following, we briefly describe the 13 motifs. The first 7 are the original seven dwarves. We try to describe the quintessential operations within a motif, characterize computation, data movement, and parallelizability.

- **Dense Linear Algebra.** This motif covers linear algebra operations (products of vectors and matrices and basic vector and matrix operations such as transposition or calculating norms as encapsulated by BLAS ("Basic Linear Algebra Subroutines") packages and more sophisticated LAPACK routines such as matrix decompositions (LU, QR, Cholesky, SVD, etc.), and solvers (triangular solves, eigenvalue solvers, etc.). "Dense" refers to the storage format of vectors and matrices: the *entire* mathematical objects are stored in memory in a contiguous data layout. This implies that dense linear algebra motif instances have very regular data access patterns and furthermore lend themselves well to vectorization and usually also well to parallelization (e.g., a matrix-matrix multiply consists of many independent scalar products).

This motif encapsulates a mathematical domain (linear algebra) and works on objects stored in one specific format.

- **Sparse Linear Algebra.** The operations are the same as in the dense linear algebra motif. However, objects are stored in a *sparse* format in which zero elements are omitted. This means that indices referencing non-zero elements must be stored as well. There are a number of established sparse matrix storage formats. The richness of the motif comes from the ways sparse matrices are stored, which leads to fundamentally different algorithmic instances of linear algebra operations depending on the storage format. Due to the irregularity and unpredictability of the sparsity pattern it is difficult to exploit the memory hierarchy, and finding a good representation of the matrix data is essential for finding a good mapping to the architecture [188, 17]. The irregularity is also a major challenge for parallelization (most notably for finding a good load balance).

  Again, this motif deals with one mathematical domain, but, as opposed to the dense linear algebra motif, is generalized to arbitrary data storage formats.

- **Spectral Methods.** The archetype of the spectral method motif is the discrete Fourier transform. The most popular algorithm for computing discrete Fourier transforms is the The Cooley-Tukey algorithm [47], which reduces the time complexity of a transform of length n from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$ by using properties of the n-th roots of unity $e^{-\frac{2\pi i}{n}\ell}$ in a recursive scheme. Due to the natural recursive structure of the algorithm, it can be decomposed into subproblems that can be solved independently, hence, the algorithm's structure can be exploited for parallelism. Furthermore, multidimensional transforms can be viewed as many independent one-dimensional transforms. One of the challenges of the Cooley-Tukey algorithm is the array holding the input data is accessed with varying (yet pre-determinable) strides, reducing spatial data locality. There are ways around this, e.g., by reordering data elements by what corresponds to a transpose.

  Generally, this motif encapsulates mathematical integral transforms.

- **N-Body Methods.** N-Body methods summarize particle simulations in which any two particles within the system interact. Given N particles, the naïve approach therefore has $\mathcal{O}(N^2)$ operations, rendering the computation processor bound as N becomes large. This approach exhibits natural parallelism, since all particle-particle interactions occur simultaneously and can be computed independently. More sophisticated approaches (e.g., the Barnes-Hut algorithm [15] or the Fast Multipole Method [73]) reduce the compute complexity to $\mathcal{O}(N \log N)$ or even to linear complexity. In these approaches, tree structures are needed to manage the computation, since the space is broken recursively into boxes, and thus the overall structure of the computation becomes related to the graph traversal motif. Load balancing becomes an issue in parallelization, which has to be done dynamically at runtime.

  This motif is specific to one particular technique of physical simulations.

- **Structured Grids.** The structured grid motif comprises computations updating each grid point of the structured grid with values from neighboring grid points. Typically the neighborhood structure is fixed, in which case it is called a *stencil*. Hence, this motif is often identified with stencil computations, which, from the point of view of the computation and data movement structure, are the main interest in this thesis.

  The structured grid motif naturally exposes a high degree of parallelism. (At least natural parallelism is exposed if distinct grids are read and written; this approach is called Jacobi iteration. The other approach is to update the grid in place (Gauss-Seidel iteration), which initially is an inherently sequential approach, but remedies exist, e.g., by skewing the iteration space (cf. section 3.4) or applying a colored iteration scheme.) The parallelism exposed is also data level parallelism, thus there is also opportunity for vectorization. However, instances of the structured grid motif are memory bound as typically only a very limited amount of computation is performed per grid point. Yet, the data access patterns are regular and statically determinable.

  Motif instances are relatively easy to implement in software (when performance is not an issue) and map well to the hardware architectures of the current trend (GPUs, Intel's MIC architectures). A more complex version of the motif is found in adaptive mesh refinement [19].

This motif is related to the sparse linear algebra motif as stencil operators are mostly considered to be linear and therefore the updates can be expressed as a multiplication of a highly regular matrix encoding the neighborhood structure (which obviously contains many redundancies due to its regularities) by a vector containing all the grid point values.

This motif encapsulates a specific class of computations defined by its regular and localized structure on data stored also in a regular fashion.

- **Unstructured Grids.** Similarly to the structured grid motif, in the unstructured grid motif grid points are updated with values on the neighboring grid points. In contrast to the structured grid motif, the grids operated on are not given, e.g., as Cartesian grids, but, e.g., as a list of triangles from some triangulation of the model geometry. This obviously destroys the regularity and determinability of the data access pattern. The double indirection used to access grid nodes also destroys data locality. But still the amount of parallelism is the same as in the structured grid motif except that it does not lend itself for immediate vectorization. Depending on the desired granularity of parallelization, often graph partitioning techniques are applied for load balancing.

  This motif is used whenever the model geometry is represented irregularly, such as by a triangulation. It occurs in finite element solvers, for instance. It is related to the sparse linear algebra motif as the updates — provided that they are linear — can be expressed as a multiplication of a vector containing all the grid point values by a matrix encoding the neighborhood structure.

  As the sparse linear algebra motif is a generalization of the dense linear algebra motif in terms of data storage, this motif is a generalization of the structured grid motif.

- **MapReduce.** The MapReduce motif summarizes parallel patterns, most notably the embarrassingly parallel *map*, i.e., the application of some function to every data element — a name borrowed from functional-style programming languages, — and *reduce*, i.e., combining a set of data elements by some reduction operator. Map and reduce can be generalized to subsets and are arbitrarily composable. The name was coined by Google, proposing the homonymous algorithm [54].

- **Combinational Logic.** The archetype of this motif is bitwise logic applied to a huge amount of data, used, e.g., when calculating CRC sums, or for encryption. It can exploit bit-level parallelism. This motif encapsulates operations on bit streams.

- **Graph Traversal.** Irregular memory accesses and typically only a low amount of computation characterize this motif. Its performance is limited by memory latency.

- **Dynamic Programming.** The idea of dynamic programming is to decompose a problem recursively into smaller (and therefore more easily solvable) subproblems. Instances of this motif include the Viterbi algorithm, methods for solving the Traveling Salesperson Problem, or the Knapsack Problem. The motif encapsulates a specific algorithmic technique typically related to optimization.

- **Backtrack and Branch-and-Bound.** Branch-and-bound algorithms are used to find global optima in certain types of combinatorial optimization problems such as Integer Linear Programming problems, Boolean Satisfiability, or the Traveling Salesperson Problem by using a divide-and-conquer strategy, which dynamically generates parallelism in a natural way. Again, this motif encapsulates a specific algorithmic technique related to optimization.

- **Graphical Models.** This motif is related and very similar to graph traversal, although the intention of this motif is to highlight probabilistic aspects found in Bayesian networks, hidden Markov models, or neural networks. Applications include speech and image recognition. So, in contrast to the more general graph traversal motif, this motif highlights the intended application domain.

- **Finite State Machines.** Finite state machines can model a system with a finite set of states and a set of rules of under which circumstances to transition between these states. It has been conjectured that finite state machines might be embarrassingly sequential [9]. Finite state machines have applications in electronic design automation, design of communication protocols, and parsing (and representing languages in general). This motif encapsulates a general discrete modeling technique.

From the previous presentation we can see that motifs are a mélange of mathematical domains, of (physical) simulation, modeling, and algorithmic techniques and are differentiated by data storage patterns.

In certain cases (for specific algorithms or applications) it might not be entirely clear how to categorize the candidate in question. For instance, sorting, one of the fundamental algorithmic problems in computer science, is mentioned in [9], but it is not explicitly classified. Maybe surprisingly, Quicksort is mentioned as an example of graph traversal. Clearly, Quicksort (and other sort algorithms) are divide-and-conquer algorithms, for which no explicit motif was defined. They could be subsumed under the quite general motif of graph traversal, mapping the recursive nature of the algorithms to trees, or they could be captured by the dynamic programming motif. In the traditional sense, an algorithm is said to be a dynamic programming algorithm if subproblems overlap, and a divide-and-conquer algorithm if the subproblems are independent, which then could be interpreted as a special case of dynamic programming.

It is noteworthy that, while established parallelization styles exist for specific motifs, the concept of parallel patterns (including the high-level structural parallel pattern, the parallelism type (task parallelism, data parallelism), parallel data structures, and implementation of parallel primitives (barriers, locks, collectives)) is orthogonal to motifs [176].

Software libraries exist for several of the motifs, at least for the initial seven dwarves, the computational science motifs. Also, the auto-tuning methodology has been applied successfully to some of the motifs. The most prominent auto-tuning frameworks within motif boundaries are:

- ATLAS [171, 172] for a subset of the dense linear algebra motif (BLAS) — one of the first frameworks to embrace the auto-tuning methodology for a scientific computing library;

- MAGMA [3, 160], bringing LAPACK and auto-tuning to multicore architectures and GPUs;

- OSKI [166] for sparse linear algebra kernels;

- FFTW [65], which is an auto-tuned framework for Fast Fourier Transforms;
- SPIRAL [139], which does auto-tuning in a more generalized setting than FFTW for many types of signal processing transforms.

Besides supporting software infrastructures for computations in the structured grid motif such as POOMA [2] and FreePOOMA [77] — which provide mechanisms to partition and parallelize over regular grids and automatically takes care of ghost zones, — or supporting software for actual stencil kernels, such as a set of C++ classes to guide efficient implementations of stencil-based computations using one particular parallelization scheme [154], and besides toolkits for sophisticated finite difference solvers such as the adaptive mesh refinement package CHOMBO [44], there are ongoing projects dedicated to stencil computations, which also support some level of tuning [92, 99, 163, 155]. Most notably, the auto-tuning methodology is pursued by Berkeley scientists in [92]; the other projects are compiler infrastructures, further discussed in the related work section in Chapter 5.6. In this thesis, we propose a software framework for the core computations of the structured grid motif embracing the auto-tuning methodology with the idea of bringing DSLs into play for user productivity and performance. The auto-tuning process is done on the basis of *Strategies*, hardware-architecture- and stencil-independent descriptions of parallelization and optimization methods to be applied.

Chapter 4

Algorithmic Challenges

> The Analytical Engine has no pretensions whatever to *originate* anything. It can do whatever we *know how to order* it to perform. It can *follow* analysis; but it has no power of *anticipating* any analytical relations or truths.
>
> — Ada Lovelace (1815–1852)

Besides the hardware and software pillars, algorithmics is the third pillar of computer science. The increasing complexity and diversity of hardware platforms and the explicit parallelism in particular have severe implications for algorithm design. Traditionally, the field of algorithmics concerns itself with theoretical development of algorithms, for which provable worst-case or average-case performance guarantees are deducted. These predictions are typically based on overly simplified, thus non-realistic machine models such as (P)RAM[*] and on an abstract description of the algorithm, which does not require an actual implementation of the algorithm.

The ultimate goal of the relatively new discipline of algorithm engineering — the community even speaks of the new paradigm — consists in bridging the gap between the algorithmic theory and practice. This

---

[*] (P)RAM, which stands for (parallel) random access machine, is a model of a register machine with an unbound memory, which is also shared addressable and uniformly accessible in the parallel model [72].

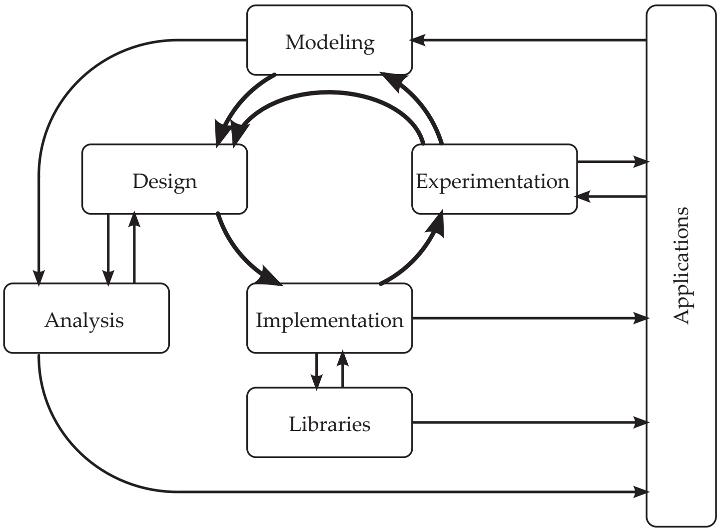


Figure 4.1: *The process of algorithm engineering*

gap was caused by separating design and analysis from implementation and experimentation [117]. Algorithm engineering promotes tighter integration of algorithm design and analysis with the practical aspects of implementation and experimentation as well as inductive reasoning, and is often driven by an application. In the algorithm engineering cycle illustrated in Fig. 4.1, design and analysis, implementation, experimentation and modeling of realistic hardware architectures are equally important concepts, and the feedback loop suggests that findings from experimentation potentially give rise to alterations and improvements of the model and the algorithm design for the next iteration. Thus, algorithm engineering, like software engineering, is not a linear process.

Emphasizing practical concerns in algorithm engineering has several consequences:

- Algorithms have to be designed based on a model capturing the actual hardware characteristics including deep memory hierarchies and, more importantly, parallelism. Creating the hardware model itself is one of the challenges of algorithm engineering as it has to reflect and adapt to the technological advances in hardware design.

- Simpler algorithms and simpler data structures are favored over more complex ones. Immensely complex algorithms and data structures have emerged from classical algorithmics as one of its goals is to improve asymptotic run time complexity of algorithms. As a matter of fact, algorithm were designed which are so complex that they have never been implemented.

- The actual running time matters. Oftentimes, worst case complexities are too pessimistic for input data from a realistic application. Also, the big O notation can hide huge constant factors — notably in the case of many complex algorithms developed for a "good" asymptotic running time complexity, — thus rendering an algorithm inefficient for practical purposes.

- Space efficiency can often be traded for time efficiency. Preprocessing can help to make a more time efficient algorithm applicable given that auxiliary data can be stored. Of course, preprocessing is only viable if the time used for preprocessing can be compensated.

- The movement towards data-driven computation entails the necessity to process exponentially increasing volumes of data. A response to the problem is the active research in one-pass space-efficient, (sub-) linear time, and approximation algorithms.

- Implementations of algorithms have to be engineered to be robust, efficient, as generally applicable as possible, and therefore reusable.

- Raising experimentation to a first class citizen implies that the experiments be reproducible. As in the natural sciences, experiments should be conducted in a rigorous manner, including the application of sound statistical methods for result evaluation.

Addressing the first point of the above list, which is probably the most important in the light of this thesis, we state that a parallel version of an algorithm typically is in fact a radically different algorithm, which although shares the same preconditions and postconditions as the sequential one. We exemplify this on the basis of the *single source shortest path* problem. Let $G = (V, E, w)$ be a weighted undirected graph with vertex set V, edge set $E \subseteq V \times V$, and a weight function $w : E \longrightarrow \mathbb{R}^+$. The sequential textbook algorithm to find the shortest path from a given source vertex $s \in V$ to all other vertices in V is Dijkstra's algorithm [58] shown in Algorithm 4.1.

Algorithm 4.1: *Dijkstra's Single Source Shortest Path Algorithm.*

```
**Input:** Graph $G = (V, E \subseteq V \times V, w : E \longrightarrow \mathbb{R}^+)$, source vertex $s \in V$
**Output:** Path lengths $\{\ell_v : v \in V\}$
$\ell_s \leftarrow 0$
$\ell_v \leftarrow \infty$ for $v \in V \setminus \{s\}$
Priority queue $Q \leftarrow \{(v, \ell_v) : v \in V\}$
while $Q \neq \emptyset$ do
    $u \leftarrow Q$.pop
    for all $v \in \mathrm{Adj}(u)$ do
        if $v \in Q$ and $\ell_u + w(u,v) < \ell_v$ then
            $\ell_v \leftarrow \ell_u + w(u,v)$                ▷ update length
        end if
    end for
end while
```

The algorithm requires an efficient implementation of a priority queue Q; the time complexity depends on how "fast" the minimum element can be extracted from the queue. For instance, when choosing a binary min heap for Q, elements can be extracted in $\mathcal{O}(\log |Q|)$ time. Then, the algorithm's time complexity is $\mathcal{O}(|E| \cdot \log(|V|))$.

The *for-all* loop in Algorithm 4.1 suggests that the algorithm can be easily parallelized. Note, however, that the parallelism is limited to the number of vertices adjacent to u — on average $|E|/|V|$. Also, an efficient concurrent implementation of a priority queue is required. By observing that the priority queue does not distinguish nodes with equal $\ell_\bullet$, the parallelism can be increased by extracting and processing all the vertices with equal $\ell_\bullet$ at once. However, as the adjacency sets might overlap in this case, the queries and updates then have to be done atomically. These and some further ideas on parallelization are discussed in [72], yet from these brief observations it can be recognized that, in spite of the suggestive *for all* loop, this algorithm cannot be parallelized effectively.

Finding efficient algorithms for the shortest path problem is still an active area of research: The ninth DIMACS challenge [62] held in 2006 was dedicated to this problem; one of the recent algorithms, PHAST [55] was published in 2011. However, only two [59, 104] of the twelve DIMACS challenge papers deal with parallel algorithms.

Shortest path algorithms are not only of academic interest: they are key ingredients in routing (typical examples including web-based map services and navigation systems) as well as in (social) network analysis (e.g., the *betweenness centrality* measure, which indicates how well a vertex is connected to the network, is based on shortest paths).

The ideas and implementations presented in one of the parallel DIMACS papers [59] are based on Dijkstra's algorithm, while in the other [104] the Δ-stepping algorithm is used, which offers more opportunity for parallelization and uses a simpler and more efficient data structure than a priority queue. Vertices are filled into buckets, which group vertices with tentative path lengths in a certain range. The path lengths — and therefore the buckets — are updated as the algorithm progresses.

PHAST on the other hand leverages preprocessing (which has to be done only once per graph) to introduce a hierarchy of *shortcut paths*, which are utilized in the actual algorithm. Not only has the sequential version of the algorithm been reported to be an order of magnitude faster than Dijkstra's (for a particular graph), but it is also amenable to parallelization and offers better data locality [55]. However, it only works as efficiently for a certain type of graphs such as road networks.

Part II

The PATUS Approach

Chapter 5

Introduction To Patus

> We may say most aptly, that the Analytical Engine weaves algebraical patterns just as the Jacquard-loom weaves flowers and leaves.
>
> — Ada Lovelace (1815–1852)

PATUS is a code generation and auto-tuning framework for the core computations of the structured grid motif: the class of stencil computations [38, 37, 39]. PATUS stands for **P**arallel **A**uto-**Tu**ned **S**tencils.

Its ultimate goals are to provide a means towards productivity and performance on current and future multi- and manycore platforms. The conceptional tool to reach these goals are separation and composability: PATUS separates the point-wise stencil evaluation from the implementation of stencil sweeps, i.e., the way the grid is traversed and the computation is parallelized. The goal is to be able to use any of the (non-recursive) stencil-specific bandwidth- and synchronization-saving algorithms, which will be described in Chapter 6, for any concrete stencil instance. The actual point-wise stencil computation is described by the *stencil specification*, and the grid traversal algorithm and parallelization scheme is described by what we call a *Strategy*. Also, both stencil specification and Strategy are (ideally) independent of a *hardware architecture description*, which, together with back-end code generators, forms the basis for the support of various hardware platforms and programming models.

These three concepts are orthogonal and (ideally) composable. The actual stencil computation is typically dictated by the application and therefore has to be implemented as a stencil specification by the PATUS user. A small domain specific language was designed for this purpose. Choosing a Strategy is independent of the specification, and also selecting the hardware platform is naturally independent of the stencil and also of the Strategy.

Thus, PATUS is an attempt at leveraging DSLs as a method for high-level abstraction specifications, implemented exemplarily for the core computations of one of the motifs, and thereby, sacrificing generality, aiming at productivity and performance.

The idea of leveraging the auto-tuning methodology is to find the best Strategy, i.e., the Strategy that yields the best performance (GFlop/s, GFlop/s per Watt, or any other performance metric which can be measured in the benchmarking code), for a fixed, application-specific stencil and a fixed hardware platform — the one the code will run on[*]. While Strategies are designed to be independent of the hardware architecture, they obviously have an impact on the performance as they, by their nature, define the mapping to the hardware architecture, and they contain some architecture-related optimizations. Yet, by leveraging DSLs once again, Strategies are independent of the programming model used to program the platform, and thus they also provide a basis for experimentation with parallelization schemes and optimization techniques. For the user who only wants to generate and auto-tune a code for an application-specific stencil, PATUS comes with a selection of Strategies, which have proven successful in practice.

Unlike in a co-design approach, we take the hardware platform as given and try to maximize the performance under these constraints. Currently traditional CPU architectures (using OpenMP for parallelization) and NVIDIA CUDA-programmable GPUs are supported as target platforms. However, separating the stencil from the grid traversal algorithm and from platform-specific optimizations, can make the tool also valuable in a hardware-software co-tuning environment.

---

[*] In the current state, the auto-tuner does not yet search over Strategies. Instead, Strategies are typically parametrized, and the auto-tuner tries to find the parameter configuration which yields the best performance for a given stencil and a given hardware platform.

5.1 Stencils and the Structured Grid Motif

Stencil computations are the core operation of the structured grid motif. A stencil computation is characterized by updating each point in a structured grid by an expression depending on the values on a fixed geometrical structure of neighboring grid points.

Applications of stencil computations range from simple finite difference-type partial differential equation (PDE) solvers to complex adaptive mesh refinement methods and multigrid methods. As in simple PDE solvers, finite difference adaptive mesh refinement methods use stencils to discretize differential operators. In multigrid methods, the interpolation operators (restriction, smoothing, and prolongation) are stencil operations. Stencil computations also occur in multimedia applications, for instance as filters in image processing such as edge detection, blurring or sharpening.

A structured grid can be viewed as a graph, but unlike the graphs considered in the unstructured grid motif or the two graph motifs, the only admissible type of graphs in the structured grid motif is such that each *interior* vertex has the same neighborhood structure, and, in particular, the same number of edges.

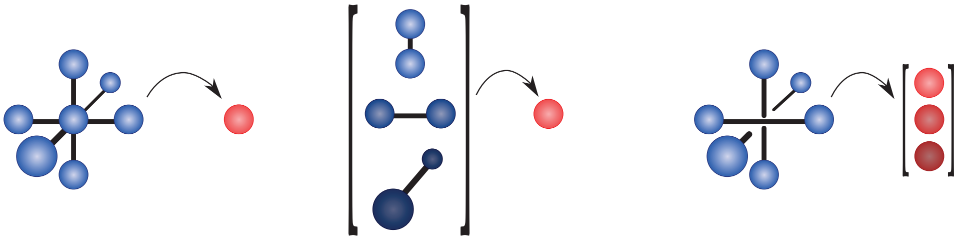
Although, in theory, the node valence, i.e., the number of a vertex's of outgoing edges, can be arbitrary, the most common case are rectilinear grids with $2d$ edges per interior node, where d is the dimensionality of the grid. The reason for this choice is the resulting simplicity of the underlying mathematics (e.g., the discretization of differential operators) and that the method then can be easily mapped to a programming model and programming language, and, ultimately, to the hardware. However, it is instructive to note that, while certain topological structures — planar surfaces, cubes, or cylinders and tori by gluing appropriate boundaries — can be readily discretized with rectilinear grids, others cannot. For instance, a rectilinear discretization of the surface of a sphere would lead to problems at the poles; an iteratively refined icosahedral mesh would be more suitable in that it would provide a uniform discretization of the sphere. A nice overview is given in [176]. In this work, we restrict ourselves to rectilinear grids.

5.1.1 Stencil Structure Examples

The geometrical structure of the actual stencil is interrelated to the grid topology. Although, once the grid topology is fixed, thereby dictating the neighborhood structure, there is still freedom to chose the neighbors contributing to the stencil. Table 5.1 shows some examples of typical stencil structures defined on rectilinear grids. These stencil structures all occur in practice when discretizing basic differential operators (upper row) or have been taken from operators found within applications.

The images show other aspects in which stencil computations can differ: although all the stencils shown are defined on a rectilinear grid, the values defined over the grid nodes can have different data structures. In simple finite difference-type PDE solvers, stencils represent discretized versions of differential operators, i.e., a stencil is a representation of a differencing operator. Table 5.1 (a) – (c) show representations of finite difference discretizations of three basic differential operators, the Laplacian, the divergence and the gradient operators. Typical finite difference approximations of these operators in Cartesian coordinates on a uniform grid with mesh size h are shown in table 5.2. Obviously, the structure of these operators differ: the Laplacian maps functions to functions, the divergence operator maps a vector field to a function and, vice versa, the gradient operator maps a function to a vector field. Therefore, the stencil representations of these operators have structurally different inputs and outputs: The discrete Laplacian operates on a scalar-valued grid, and the result is written back to a scalar-valued grid (the same or a different grid, depending on the grid traversal strategy). The divergence operator takes a vector-valued input grid and writes to a scalar-valued output grid. Conversely, the gradient has a scalar-valued grid as input and a vector-valued grid as output.

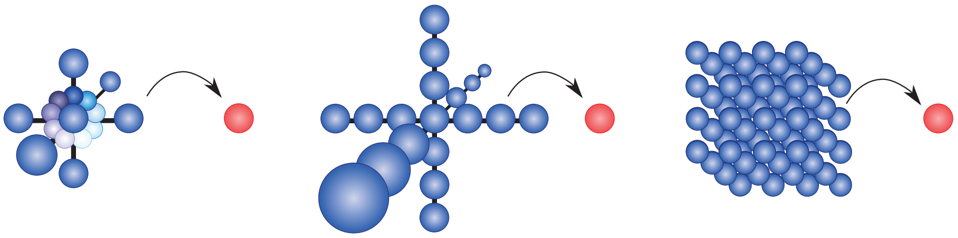
Table 5.1 (d) visualizes the stencil of the hyperthermia application (cf. Chapter 11.1), which requires many additional coefficients; (e) and (f) are higher-order stencils, which depend on more than the immediate neighboring grid points. The stencils are inspired by stencils occurring in COSMO [64], a weather prediction code. The *Wave* stencil in (g) visualizes the stencil of an explicit finite difference solver for the classical wave equation, which will be used in Chapter 5.2 as an illustrative example for the usage of PATUS. Unlike the other stencils in Table 5.1 it depends on two previous time steps instead of just one. Sub-figure (h) finally visualizes the 2D stencil from the edge detection and the Game of



(a) **Laplacian**
3D 7-point stencil,
scalar → scalar

(b) **Divergence**
3D 6-point stencil,
vector → scalar

(c) **Gradient**
3D 6-point stencil,
scalar → vector



(d) **Hyperthermia**
3D 7-point stencil,
scalar + 9 coefficients
→ scalar

(e) **6th order Laplacian**
3D 19-point stencil,
scalar → scalar

(f) **Tricubic interpolation**
3D 64-point stencil,
scalar → scalar

(g) **Wave**
3D 13-point stencil,
scalar → scalar
depending on 2 time steps

(h) **Edge detection / Game of Life**
2D 9-point stencil,
scalar → scalar

Table 5.1: *Some examples of typical stencil structures.*

Operator	Notation & Cart. representation	A finite difference approximation
Gradient	$\nabla = \left(\dots, \frac{\partial}{\partial x_i}, \dots\right)^\mathsf{T}$	$\frac{1}{2h}\left(\dots, u(\mathbf{x}+h\mathbf{e}_i) - u(\mathbf{x}-h\mathbf{e}_i), \dots\right)^\mathsf{T}$
Divergence	$\nabla\cdot = \left(\dots, \frac{\partial}{\partial x_i}, \dots\right)$	$\frac{1}{2h}\sum_i \left(u_i(\mathbf{x}+h\mathbf{e}_i) - u_i(\mathbf{x}-h\mathbf{e}_i)\right)$
Laplacian	$\Delta := \nabla\cdot\nabla = \sum_i \frac{\partial^2}{\partial x_i^2}$	$\frac{1}{h^2}\sum_i \left(u(\mathbf{x}-h\mathbf{e}_i) - 2u(\mathbf{x}) + u(\mathbf{x}+h\mathbf{e}_i)\right)$

Table 5.2: *Basic differential operators, their Cartesian representation, and a finite difference approximation evaluated on a function u, which is scalar-valued for the gradient and the Laplacian, and vector-valued for the divergence operator.*

Life kernels.

Image Processing

In image processing applications, stencil computations occur in *kernels* based on convolution matrices, which assign each pixel in the result image the weighted sum of the surrounding pixels of the corresponding pixel in the input image. The matrix contains the weights, which stay constant within a sweep. Fig. 5.1 shows two examples of image processing kernels and the corresponding convolution matrices; a blur kernel based on the Gaussian blur

$$G(x,y) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right)$$

with $\sigma = 5$ (but cut off to fit into a 5×5 matrix) and an edge detection kernel [25].

Cellular Automata

Another application of stencil operations are cellular automata. Conceptually, a cellular automaton is an infinite discrete grid of cells, each of which takes one of finitely many states at any given time within a discrete time line. In each time step, a set of rules is applied to each cell, depending on its state and the state of its neighbors. Thus, the global state of the cellular automaton is evolved over time from an initial configuration. The term was coined in the 1960s by von Neumann [144] who was

$$[1] \qquad \frac{1}{1000}\begin{bmatrix} 36 & 39 & 40 & 39 & 36 \\ 39 & 42 & 43 & 42 & 39 \\ 40 & 43 & 44 & 43 & 40 \\ 39 & 42 & 43 & 42 & 39 \\ 36 & 39 & 40 & 39 & 36 \end{bmatrix} \qquad \begin{bmatrix} 1 & 2 & 1 \\ 2 & -12 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Original image Blur filter Edge detection

Figure 5.1: *Two examples of image processing filters and the corresponding convolution matrices.*

working on models for self-replicating biological systems. Cellular automata have been applied by physicists and biologists as a discrete modeling technique. Cellular automata were broadly popularized by Gardner who described Conway's *Game of Life*, a cellular automaton devised in the 1970s [69].

In the Game of Life, each cell — also called an *organism* in this context — can be either *live* or *dead*. Starting from an initial configuration the following "genetic" rules are applied:

- **Survival**. A *live* cell with two or three neighbors remains *live*.
- **Death**. A *live* cell *dies* from overpopulation if there are more than three *live* neighbors or from isolation if there are fewer than two *live* neighbors.
- **Birth**. A *dead* cell becomes *live* if it has exactly three *live* neighbors.

Obviously, the rules are nearest neighbor operations, and, in 2D, they can be formulated as a 9-point stencil. Currently, PATUS does not support conditionals, but for the Game of Life they can be converted to an arithmetic expression, which lets PATUS emulate the cellular automaton, albeit inefficiently: If *live* cells are represented by the value 1 and *dead* cells by a value close to 0, then the number of *live* neighboring cells at

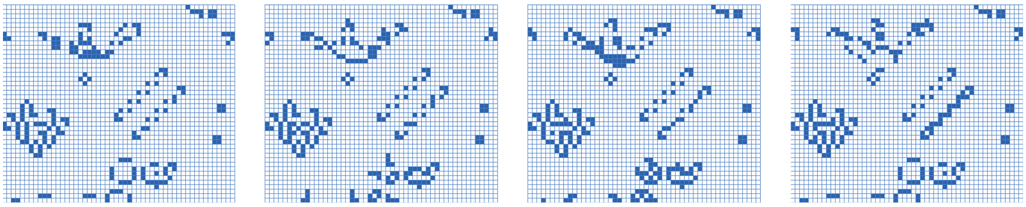


Figure 5.2: *Four iterations of the Game of Life.*

time step t is

$$\begin{aligned} L := & u[x-1,y-1;t] + u[x,y-1;t] + u[x+1,y-1;t] + \\ & u[x-1,y;t] + u[x+1,y;t] + \\ & u[x-1,y+1;t] + u[x,y+1;t] + u[x+1,y+1;t] \end{aligned}$$

by making use of the fact that on a computer $1+\alpha = 1$ for a sufficiently small α.

The rules could then be encoded as the arithmetic stencil expression

$$u[x,y;t+1] := \frac{1}{1+(u[x,y;t]+L-3)(L-3)C},$$

where $C \gg 1$ is some large number. The observation is that the polynomial $(u[x,y;t]+L-3)(L-3)$ has zeros for $L = 3$ (i.e., if the number of live neighbors is 3) and, since $u[x,y;t] \in \{0,1\}$, for $(u[x,y;t],L) \in \{(0,3),(1,2)\}$. The function $\xi \mapsto \frac{1}{1+C\xi}$ maps $\xi = 0$ to 1 and any other $\xi \in \mathbb{Z}$ to a number close to 0. Hence, in summary, if the number of *live* neighbors is 3 (i.e., $L = 3$), $u[x,y;t+1]$ will become 1, regardless of the value of $u[x,y;t]$, in accordance with the second part of the survival rule and the birth rule. If $u[x,y;t] = 1$ and the number of *live* neighbors is 2, $L = 2$, $u[x,y;t+1]$ will also become 1 in accordance with the first part of the survival rule. In all other cases the organism dies ($u[x,y;t+1] \approx 0$).

5.1.2 Stencil Sweeps

We call the application of a stencil computation to all the interior points in a grid a *stencil sweep*, or simply, a *sweep*.

Oftentimes, when conducting scientific simulations modeled by PDEs, it is of interest how the observables evolve over time. Such time-dependent problems are discretized in time, and advancing the states by one time step corresponds to carrying out one stencil sweep. Examples for such simulations are the applications discussed in Chapter 11. The

Hyperthermia cancer treatment planning application is a simulation that tries to predict the temperature distribution within the human body during the treatment, which involves applying heat to the body at the tumor location. The second application is an explicit finite difference solver to conduct earthquake simulations.

Another characteristic of the stencil motif is the way the grid is traversed. If there are distinct input and output sets — sets of nodes on which data is only read from and other sets of nodes to which data is only written to within one sweep — the order in which the nodes are processed is irrelevant. This type of procedure is called the *Jacobi iteration*. This is the problem structure which we limit ourselves to in this work.

However, if, while sweeping through the grid, the same node set is used for both read and write accesses, the order in which the nodes are visited becomes relevant, as the method then prescribes which part of the stencil uses the new, updated values for the evaluation and which part still operates on the old values. Such a method is called a *Gauss-Seidel iteration*. Such a method is hard to parallelize. One common variant which lends itself to parallelization is a red-black Gauss-Seidel: every other node is colored in the same color and in the first sweep all the red nodes are updated while the black nodes are only read, and in the second sweep the roles of the nodes are exchanged. This is akin to the homonymous methods in numerical linear algebra [143] to solve sparse systems of linear equations.

5.1.3 Boundary Conditions

Another concern are handling nodes at boundaries (i.e., nodes with a neighborhood structure differing from the "standard" structure). Because of the different structure, the stencil might not be applicable to boundary nodes; neighboring nodes might be missing. Also, the continuous mathematical model for the computation might require to handle boundary nodes differently and apply problem-specific boundary conditions. For instance, the simplest case (Dirichlet boundary conditions) is to keep the value on the boundary constant. Other applications might require that the flux through the boundary is conserved (Neumann boundary conditions), or, when modeling waves, a common requirement is that waves are absorbed instead of reflected at the domain boundary (absorbing boundary conditions, e.g., perfectly matched layers [18], since typically only a small subdomain of the otherwise infinite domain is modeled). In this work we choose not to treat boundaries specially for the time being. Instead, since the boundary can be viewed as a sub-manifold, we suggests that a special, separate stencil is defined, implementing the boundary handling.

5.1.4 Stencil Code Examples

In code, a simple, basic stencil computation is shown in Algorithm 5.1. The stencil, a 3D 7-point Laplacian, which depends on all the nearest neighbors parallel to the axes, is evaluated for all the interior grid points indexed by integer coordinates $[1, X] \times [1, Y] \times [1, Z] \cap \mathbb{Z}^3$. The center point is denoted by $u[x, y, z; t]$. x, y, z are the spatial coordinates, t is the temporal coordinate.

Algorithm 5.1: *A Laplacian stencil sweeping over* $[1, X] \times [1, Y] \times [1, Z]$.

```
procedure STENCIL-LAPLACIAN
    initialize $u[x, y, z; 0]$ and boundaries $u[\{0, X+1\}, \{0, Y+1\}, \{0, Z+1\}; t]$
    for $t = 0..T-1$ do                                  ▷ iterate over time domain
        for $z = 1..Z$ do                                ▷ iterate over spatial domain
            for $y = 1..Y$ do
                for $x = 1..X$ do
                    $u[x, y, z; t+1] \leftarrow \alpha \cdot u[x, y, z; t] + \beta \cdot ($   ▷ evaluate
                        $u[x-1, y, z; t] + u[x+1, y, z; t] +$                    ▷ stencil
                        $u[x, y-1, z; t] + u[x, y+1, z; t] +$
                        $u[x, y, z-1; t] + u[x, y, z+1; t])$
                end for
            end for
        end for
    end for
end procedure
```

In practice, however, it is hardly necessary to save every time step. Hence, we can use only two grids in the computation, `old` and `new`, as shown in Algorithm 5.2. After each sweep, the roles of `old` and `new` are exchanged (line 14), and the values computed in the previous sweep are again accessed from the `old` grid in the current sweep.

Algorithm 5.2: *Modified Laplacian stencil.*

```
procedure STENCIL-LAPLACIAN
    initialize old[$x, y, z$] (including boundary points)
    for $t = 0..T - 1$ do                                  ▷ iterate over time domain
        for $z = 1..Z$ do                                  ▷ iterate over spatial domain
            for $y = 1..Y$ do
                for $x = 1..X$ do
                    new[$x, y, z$] ← $\alpha \cdot$ old[$x, y, z$] + $\beta \cdot$ (      ▷ evaluate
                        old[$x - 1, y, z$] + old[$x + 1, y, z$]+      ▷ stencil
                        old[$x, y - 1, z$] + old[$x, y + 1, z$]+
                        old[$x, y, z - 1$] + old[$x, y, z + 1$])
                end for
            end for
        end for
        swap (new, old)                                    ▷ swap pointers to new and old
    end for
end procedure
```

5.1.5 Arithmetic Intensity

The arithmetic intensity [131] is a metric which is often used for analyzing and predicting the performance of algorithms and compute kernels.

Definition 5.1. *The* arithmetic intensity *of an algorithm or a compute kernel is defined as the number of floating point operations used for that algorithm divided by the number of data elements that have to be transferred to the compute unit(s) in order to carry out the computation.*

Example 5.1: *Arithmetic intensity of a matrix-matrix multiplication.*

Multiplying two $n \times n$ matrices equates to evaluating n^2 scalar products, each of which has $\mathcal{O}(n)$ floating point operations. The data that need to be brought into memory are the two matrix factors: a data volume of $\mathcal{O}(n^2)$ data elements. Thus, the arithmetic intensity is $\frac{\mathcal{O}(n^3)}{\mathcal{O}(n^2)} = \mathcal{O}(n)$.

From a hardware architecture point of view, stencil computations are attractive because of the regularity of the data access pattern. This allows streaming in the data from the main memory to the compute elements, unlike in the unstructured grid motif where logically related data can be far apart in memory. Yet, due to their typical low arithmetic intensity, the performance of typical structured grid motif kernels is limited by the available memory bandwidth: The number of floating point operations per grid point is constant and is typically low compared to the — also constant — number of memory references. Thus, stencil computations have a constant arithmetic intensity with respect to the problem size, unlike, e.g., a matrix-matrix multiplication, whose arithmetic intensity scales with the problem size as shown in Example 5.1.

The upper bounds for the arithmetic intensities of the stencils shown in Table 5.1 are summarized in Table 5.3. The numbers given in Table 5.3 are actually upper bounds on the *ideal* arithmetic intensities, which are calculated only from compulsory data transfers. Viewed globally, we need to bring the data entire d-dimensional domain required for the computation to the compute elements, and write the result back. In addition to the computed data, the data volume that is read has to include the halo layer. Thus, for instance, for the Laplacian, on a domain in which N grid points are computed, we need to bring $N + \mathcal{O}\left(dN^{(d-1)/d}\right)$ points[†] (including the halo points) to the compute elements (or into fast memory) and write back N points. Equivalently, $\frac{1}{N}\left(\left(N + \mathcal{O}\left(dN^{(d-1)/d}\right)\right) + N\right) = 2 + \mathcal{O}\left(dN^{(d-2)/d}\right)$ data elements per computed element need to be transferred. In Table 5.3 we neglect the extra term, thus ignoring the extra boundary data that has to be brought in. Therefore the values shown as arithmetic intensities are really upper bounds which cannot be reached in practice. To compute the divergence operator, we have to read 3 grids instead of just one, and write back one grid, hence the "4" in the "number of data elements" column in Table 5.3. Conversely, to compute the gradient, one grid has to be read and 3 grids written back, etc.

There is also a hardware-related local-view justification for the as-

---

[†] A face of a hypercube-shaped (rectilinear) discrete d-dimensional domain containing N points has approximately $N^{(d-1)/d}$ points, and a d-dimensional hypercube has $2d$ faces, which accounts for the factor d. The latter can be seen inductively from the construction of a $(d+1)$-dimensional hypercube from two d-dimensional ones: each of the two d-dimensional hypercubes are one face of the new $(d+1)$-dimensional one, and each of the $2d$ faces (by induction hypothesis) gives rise to a face in the $(d+1)$-dimensional hypercube by connecting the $(d-1)$-dimensional faces of each of the copies of the d-dimensional hypercubes to d-dimensional sub-hypercubes: the faces in the new $(d+1)$-dimensional hypercube.

Kernel	Flops	# Data Elements	Arithmetic Intensity
Laplacian	8	2	4.0
Divergence	8	4	2.0
Gradient	6	4	1.5
Hyperthermia	16	11	1.4
6th order Laplacian	22	2	11.0
Tricubic	318	5	63.6
Wave	19	2	9.5
Blur	31	2	15.5
Edge Detection	10	2	5.0
Game of Life	13	2	6.5

Table 5.3: *Flop counts, compulsory numbers of data elements to transfer, and arithmetic intensities (Flops per transferred data element) for the stencils shown as examples in Table 5.1.*

sumption that by loading the center point of a stencil the neighboring points are made available automatically: In hardware, data is loaded by blocks rather than by individual elements. For instance, by reading one element on a cache-based CPU architecture, in fact an entire cache line of data is read. Also the typical shared memory loading pattern on CUDA-programmed GPUs follows this pattern.

The arithmetic intensity is a kernel-specific metric. Its analogue for a hardware platform is sometimes called the *compute balance*. The compute balance is defined as the number of (floating point) operations the machine can do per transferred data element. To measure realistic compute balances, we could measure the sustained performance of a machine, e.g., by running the LINPACK benchmark or benchmarking a matrix-matrix multiply with two large matrices (or some other compute bound kernel), and divide the number by the available bandwidth, measured, e.g., using the STREAM benchmark [106]. The compute balances for the machines used for our benchmarks in Part III are shown in Table 5.4[‡].

[‡] The performance numbers were measured by benchmarking a single precision matrix-matrix multiply (SGEMM) of two large square matrices (8192 × 8192) using the MKL on the Intel platform, the ACML on the AMD and CUBLAS on the GPU, respectively, on all available cores. The performance of a double precision matrix-matrix multiply (DGEMM) is half of that of SGEMM. Note that the compute balance in Flops per data element does not change when the data type is altered, as the bandwidth in number of data elements per second is halved when switching from single to double precision. (A single precision number has a length of 4 Bytes, a double precision requires 8 Bytes.) The bandwidths of the Intel and AMD CPU systems were measured with the STREAM benchmark [106], and the bandwidth of the NVIDIA Fermi C2050 GPU was measured

Architecture	Sust. Performance	Sust. Bandwidth	Compute Balance
Intel Xeon E7540 *(Nehalem)*	155 GFlop/s	35.0 GB/s	17.7 Flop/datum
AMD Opteron 6172 *(Magny Cours)*	309 GFlop/s	53.1 GB/s	23.3 Flop/datum
NVIDIA Tesla C2050 *(Fermi)*	618 GFlop/s	84.4 GB/s	29.3 Flop/datum

Table 5.4: *Sustained single precision peak performances, sustained DRAM bandwidths, and compute balances for the hardware platforms used in the benchmarks in Part III.*

Intuitively, one might argue that if the arithmetic intensity of a kernel is less than the compute balance of a machine, the kernel is memory bound, and if the arithmetic intensity is greater than the compute balance, the kernel is compute bound. Asymptotically, the statement is true, but today's complex architectures introduce subtleties which call for sophistication of the model. In [176], Williams develops the *roofline model*, which, based on the arithmetic intensity, provides a means to reflect the impact of certain hardware-specific code optimizations. The models allow to determine which memory-specific or compute-specific optimizations (NUMA awareness, software prefetching, vectorization, balancing multiply and add instructions, etc.) have to be carried out so that the kernel, given its arithmetic intensity, runs most efficiently on a given hardware platform. Note that the notion of the compute balance should even be differentiated with respect to the memory hierarchy since the bandwidths to the different levels in the memory hierarchy differ — by orders of magnitude. Hence, in reality, there are DRAM, L3, L2, L1, and register compute balances. In Table 5.4 only the DRAM compute balance is shown.

5.2 A Patus Walkthrough Example

In this section, we give an example, how a PATUS stencil specification can be derived from a mathematical problem description. From this stencil specification, the PATUS code generator generates a C code which implements the compute kernel along with a benchmarking harness that can be

---

using NVIDIA's bandwidth benchmark with ECC turned on.

used to measure the performance of the generated code and also shows how the kernel is called from within a user code.

5.2.1 From a Model to a Stencil

Consider the classical wave equation on $\Omega = [-1, 1]^3$ with Dirichlet boundary conditions and some initial condition:

$$\begin{aligned} \frac{\partial^2 u}{\partial t^2} - c^2 \Delta u &= 0 && \text{in } \Omega, \\ u &= g && \text{on } \partial\Omega, \\ u|_{t=0} &= f. \end{aligned} \tag{5.1}$$

We use an explicit finite difference method to discretize the equation both in space and time. For the discretization in time we use a second-order scheme with time step δt. For the discretization in space we choose a fourth-order discretization of the Laplacian Δ on the structured uniformly discretized grid Ω_h with step size h. This discretization gives us

$$\frac{u^{(t+\delta t)} - 2u^{(t)} + u^{(t-\delta t)}}{\delta t} - c^2 \Delta_h u^{(t)} = 0, \tag{5.2}$$

where Δ_h is the discretized version of the Laplacian:

$$\begin{aligned} \Delta_h u^{(t)}(x,y,z) = {} & \tfrac{-15}{2h^2} u^{(t)}(x,y,z) + \\ & \tfrac{-1}{12h^2} \Big(u^{(t)}(x-2h,y,z) + u^{(t)}(x,y-2h,z) + u^{(t)}(x,y,z-2h) \Big) + \\ & \tfrac{4}{3h^2} \Big(u^{(t)}(x-h,y,z) + u^{(t)}(x,y-h,z) + u^{(t)}(x,y,z-h) \Big) + \\ & \tfrac{4}{3h^2} \Big(u^{(t)}(x+h,y,z) + u^{(t)}(x,y+h,z) + u^{(t)}(x,y,z+h) \Big) + \\ & \tfrac{-1}{12h^2} \Big(u^{(t)}(x+2h,y,z) + u^{(t)}(x,y+2h,z) + u^{(t)}(x,y,z+2h) \Big). \end{aligned} \tag{5.3}$$

Substituting Eqn. 5.3 into Eqn. 5.2, solving Eqn. 5.2 for $u^{(t+\delta t)}$, and interpreting u as a grid in space and time with mesh size h and time step δt, we arrive at the stencil expression

$$\begin{aligned} u[x,y,z;t+1] = 2u[x,y,z;t] - u[x,y,z;t-1] + c^2 \tfrac{\delta t}{h^2} \Big(& \tfrac{-15}{2} u[x,y,z;t] + \\ & \tfrac{-1}{12} \left(u[x-2,y,z;t] + u[x,y-2,z;t] + u[x,y,z-2;t] \right) + \\ & \tfrac{4}{3} \left(u[x-1,y,z;t] + u[x,y-1,z;t] + u[x,y,z-1;t] \right) + \\ & \tfrac{4}{3} \left(u[x+1,y,z;t] + u[x,y+1,z;t] + u[x,y,z+1;t] \right) + \\ & \tfrac{-1}{12} \left(u[x+2,y,z;t] + u[x,y+2,z;t] + u[x,y,z+2;t] \right) \Big), \end{aligned}$$

which is visualized in Fig. 5.1 (g).

To actually solve the discretized equation, we need to specify the mesh size h and the time step δt or, equivalently, the number of grid points N and the number of time steps $t_{\max}$. Choosing a concrete number for the number of time steps, we can transform the above equation almost trivially into a PATUS stencil specification:

Example 5.2: *A PATUS stencil specification.*

The listing below shows the PATUS stencil specification for the classical wave equation discretized by 4th order finite differences in space and by 2nd order finite differences in time. Note that the maximum N in the domain size specification is inclusive.

```
stencil wave
{
    domainsize = (1 .. N, 1 .. N, 1 .. N);
    t_max = 100;

    operation (float grid u, float param c2dt_h2)
    {
        u[x,y,z; t+1] = 2 * u[x,y,z; t] - u[x,y,z; t-1]+
            c2dt_h2 * (
                -15/2 * u[x, y, z; t] +
                4/3 * (
                    u[x+1, y, z; t] + u[x-1, y, z; t] +
                    u[x, y+1, z; t] + u[x, y-1, z; t] +
                    u[x, y, z+1; t] + u[x, y, z-1; t]
                )
                -1/12 * (
                    u[x+2, y, z; t] + u[x-2, y, z; t] +
                    u[x, y+2, z; t] + u[x, y-2, z; t] +
                    u[x, y, z+2; t] + u[x, y, z-2; t]
                )
            );
    }
}
```

5.2.2 Generating The Code

We feed this stencil specification as an input to PATUS, which will turn it to C code. PATUS expects two other input files: a template defining how the code will be parallelized and how code optimizations will be applied, e.g., how loop tiling/cache blocking is applied. In PATUS lingo, this is called a *Strategy*. The other input is a description of the hardware architecture. It defines which code generation back-end to use (e.g., the OpenMP paradigm for shared memory CPU system, or NVIDIA C for CUDA for NVIDIA GPUs), and how arithmetic operations and data types are mapped to the corresponding vector intrinsics and vector data types, for instance.

Example 5.3: *Generating the C code from a stencil specification.*

```
java -jar patus.jar codegen --stencil=wave.stc
    --strategy=cacheblocking.stg
    --architecture="arch/architectures.xml,Intel x86_64 SSE"
    --outdir=output
```

The command in Example 5.3 will create an implementation for the stencil kernel specified in the file `wave.stc` and a benchmarking harness for that kernel and file it in the directory `output`. The Strategy chosen is a cache blocking strategy defined in the file `cacheblocking.stg`, which comes with the PATUS software. The hardware architecture for which the code will be generated is specified by the identifier `Intel x86_64 SSE`, the definition of which can be found in the architecture definition file, `arch/architectures.xml`.

After running the PATUS code generation, the directory `output` will contain the following files:

- `kernel.c` — The implementation of the stencil kernel defined in `wave.stc`.

- `driver.c` — The benchmarking harness invoking the stencil kernel and measuring the time for the stencil call. It allocates and initializes data with arbitrary values and (by default) validates the result returned by the stencil kernel by comparing it to a naïve sequential implementation.

- `timer.c` — Functions related to timing and calculating the performance.

- `patusrt.h` — Header file for the timing functions.

- `cycle.h` — Functions for counting clock cycles to do the timing. (This code was borrowed from FFTW [65].)

- `Makefile` — A GNUmake Makefile to build the benchmarking harness.

The benchmarking harness then can be built by typing `make` on the command line. This will compile and link the generated code and produce the benchmarking executable, by default called `bench`.

5.2.3 Running and Tuning

The benchmark executable requires a number of parameters to run:

Example 5.4: *Starting the benchmark executable.*

```
chrmat@palu1:wave> ./bench
Wrong number of parameters.  Syntax:
./bench <N> <cb_x> <cb_y> <cb_z> <chunk> <_unroll_p3>
```

`N` corresponds to the N used in the stencil specification for the definition of the domain size. If additional identifiers would have been used to define the domain size, they would appear as parameters to the executable as well. The `cb_x`, `cb_y`, `cb_z`, and `chunk` arguments come from the Strategy. They specify the sizes of the cache blocks in x, y, and z directions and the number of consecutive cache blocks assigned to one thread. PATUS unrolls the inner-most loop nest containing the stencil evaluation. `_unroll_p3` selects one of the unroll configuration code variants: by default, PATUS creates code variants with the loop nest unrolled once (i.e., no unrolling is done) or twice in each direction. Since the example wave stencil is defined in 3 dimensions, there are $2^3 = 8$ code variants with different unrolling configurations.

In Example 5.5, the benchmark executable was run with a domain size of 200^3 grid points and an arbitrary cache block size of $16 \times 16 \times 16$ grid points per block, one block in a packet per thread, and the 0^{th} loop unrolling configuration.

Example 5.5: *Running the benchmark executable with arbitrary parameters.*

```
chrmat@palu1:wave> ./bench 200 16 16 16 1 0
Flops / stencil call:  19
Stencil computations:  40000000
Bytes transferred:     509379840
Total Flops:           760000000
Seconds elapsed:       0.230204
```

Example 5.5: *Running the benchmark executable with arbitrary parameters.* (cont.)

```
Performance:           3.301418 GFlop/s
Bandwidth utilization: 2.212731 GB/s
506450156.000000
Validation OK.
```

The benchmark executable prints the number of floating point operations per stencil evaluation, the total number of stencil evaluations that were performed for the time measurement, the number of transferred bytes and the total number of floating point operations, the time it took to complete the sweeps and the calculated performance in GFlop/s and the bandwidth utilization. The number below that is a representation of the time spent in the compute kernel, on which the auto-tuner bases the search. (It tries to minimize that number.) The string "Validation OK" says that the validation test (against the naïve, sequential implementation) was passed, i.e., the relative errors did not exceed certain bounds.

The idea is that the user chooses the problem size, N, but all the other parameters, which do not change the problem definition, but rather implementation details which potentially affect the performance, are to be chosen by the auto-tuner. In the current state of the software, the user still needs some knowledge how to find the performance-specific parameters (in this case `cb_x`, `cb_y`, `cb_z`, `chunk`, `_unroll_p3`). An idea for future work is to encapsulate this knowledge in the Strategy, which defines the parameters, so that an auto-tuner configuration script can be generated in order to fully automate the auto-tuning process.

We choose $N = 200$. Experience tells us that in cases with relatively small domain size (such as $N = 200$), a good choice for `cb_x` is N. There are several reasons why it is a bad idea to cut the domain in the x direction — which is the unit stride direction, — i.e., choosing `cb_x` $< N$. Reading data in a streaming fashion from DRAM is faster than jumping between addresses. Utilizing full cache lines maximizes data locality. And the hardware prefetcher is most effective when the constant-stride data streams are as long as possible. In Example 5.6 we let the auto-tuner choose `cb_y`, `cb_z`, `chunk`, and `_unroll_p3` by letting `cb_y` and `cb_z` from 4 to $N = 200$ in increments of 4 (`4:4:200`), and we want the auto-tuner to try all the powers of 2 between 1 and 16 (`1:*2:16!`) for the fifth command line argument, `chunk`, and try all the values between 0 and 7 (`0:7!`) for `_unroll_p3`, by which all the generated loop unrolling variants are referenced. The exclamation mark specifies that the corresponding parameter is searched exhaustively, i.e., the auto-tuner is instructed to visit all of the values in the range specified.

Example 5.6: *Running the* PATUS *auto-tuner.*

```
java -jar patus.jar autotune ./bench 200 200 4:4:200 4:4:200
   1:*2:16!  0:7!
   "C((\$1+\$3-1)/\$3)*((\$1+\$4-1)/\$4)>=$OMP_NUM_THREADS"
```

The auto-tuner also allows the user to add constraints, such as the expression in the last argument in the call in Example 5.6. Constraints are preceded by a `C` and can be followed by any comparison expression involving arithmetic. Parameter values are referenced by a number preceded by a dollar sign \$; the numbering starts with 1. In Example 5.6, the condition reads $(\$1 + \$3 - 1)/\$3 \cdot (\$1 + \$4 - 1)/\$4 \geqslant T$. The subexpression $(\$1 + \$3 - 1)/\$3$ is the number of blocks in the y-direction, using the integer division (which rounds towards zero) to express the missing ceiling function[§]). Similarly, $(\$1 + \$4 - 1)/\$4$ is the number of blocks in z-direction. Thus, the condition demands that the total number of blocks must be greater or equal than the number of threads executing the program (assuming the environment variable `$OMP_NUM_THREADS` is set and controls how many OpenMP threads the program uses). Adding constraints is optional, but they can reduce the number of searches as they restrict the search space. In this case, we exclude the configurations in the search space of which we know they result in bad performance, but constraints can be a helpful tool to suppress invalid configurations. For instance, the numbers of threads per block on a GPU must not exceed a particular number lest the program fails. (The numbers are 512 threads per block on older graphics cards, and 1024 threads per block on Fermi GPUs).

Example 5.7 shows an excerpt of the auto-tuner output. The program is run for many parameter configurations, and at the end of the search, the auto-tuner displays the parameter configuration and output of the run for which the best performance was achieved.

[§] For positive integers a, b and the integer division $\div$ (defined by $a \div b := \lfloor \frac{a}{b} \rfloor$), it holds $\lceil \frac{a}{b} \rceil = (a + b - 1) \div b$.

Example 5.7: *Output of the auto-tuner.*

```
./bench 200 200 4:4:200 4:4:200 1:*2:16!  0:7!
Parameter set { 200 }
Parameter set { 200 }
Parameter set { 4, 8, ...}
Parameter set { 4, 8, ...}
Parameter set , Exhaustive { 1, 2, 4, 8, 16 }
Parameter set , Exhaustive { 0, 1, 2, 3, 4, 5, 6, 7 }
Using optimizer:  Powell search method
Executing [./bench, 200, 200, 4, 4, 1, 0]...
...

200 200 36 160 1 3
1.5052755E8

Program output of the optimal run:
Flops / stencil call:  19
Stencil computations:  40000000
Bytes transferred:     509379840
Total Flops:           760000000
Seconds elapsed:       0.068421
Performance:           11.107680 GFlop/s
Bandwidth utilization: 7.444774 GB/s
150527550.000000
Validation OK.
```

5.3 Integrating into User Code

By default, PATUS creates a C source file named `kernel.c` (The default setting can be overridden with the `--kernel-file` command line option, cf. Appendix A.) This kernel file contains all the generated code variants of the stencil kernel, a function selecting one of the code variants, and an initialization function, `initialize`, which does the NUMA-aware data initialization and should preferably be called directly after allocating the data (cf. Chapter 6.4.2).

Example 5.8: *The generated stencil kernel code for the example wave stencil.*

The generated stencil kernel and the data initialization function have the following signatures:

```
void wave (float** u_0_1_out,
  float* u_0_m1, float* u_0_0, float* u_0_1,
```

Example 5.8: *The generated stencil kernel code for the example wave stencil.* (cont.)

```
  float c2dt_h2,
  int N,
  int cb_x, int cb_y, int cb_z, int chunk,
  int _unroll_p3);

void initialize (
  float* u_0_m1, float* u_0_0, float* u_0_1,
  float dt_dx_sq,
  int N,
  int cb_x, int cb_y, int cb_z, int chunk);
```

The selector function, which is named exactly as the stencil in the stencil specification (`wave` in Example 5.8), is the function, which should be called in the user code. Its parameters are:

- pointers to the grid arrays, which will contain the results on exit; these are double pointers and marked with the `_out` suffix;
- input grid arrays, one for each time index required to carry out the stencil computation; e.g., in the wave equation example, three time indices are required: the result time step $t+1$, which depends on the input time steps t and $t-1$; the time index is appended to the grid identifier as last suffix; the `m` stands for "minus";
- any parameters defined in the stencil specification, e.g., `c2dt_h2` in the wave equation example;
- all the variables used to specify the size of the problem domain; only `N` in our example;
- Strategy- and optimization-related parameters, the best values of which were determined by the auto-tuner and can be substituted into the stencil kernel function call in the user code; the strategy used in the example has one cache block size parameter, (`cb_x`, `cb_y`, `cb_z`) and a chunk size `chunk`; furthermore, `_unroll_p3` determines the loop unrolling configuration.

The output pointers are required because PATUS rotates the time step grids internally, i.e., the input grids change roles after each spatial sweep. Thus, the user does typically not know which of the grid arrays contains the solution. A pointer to the solution grid is therefore assigned to the output pointer upon exit of the kernel function.

The initialization function can be modified to reflect the initial condition required by the problem. However, the generated loop structure should not be altered. Alternatively, a custom initialization can be done after calling the initialization function generated by PATUS.

5.4 Alternate Entry Points to Patus

As the auto-tuner module is decoupled from the code generating system, it can be used as a stand-alone auto-tuner for other codes besides the ones created by PATUS. If the user has a hand-crafted existing parametrized code for which the best configurations need to be determined, the PATUS auto-tuner is a perfectly valid option to find the best set of parameters. The only requirements are that the tunable parameters must be exposed as command line parameters and that the program must print the timing information to `stdout` as last line of the program output in an arbitrary time measurement unit. Note that the auto-tuner tries to *minimize* this number, i.e., it must be a time measurement rather than a performance number.

5.5 Current Limitations

In the current state, there are several limitations to the PATUS framework:

- Only shared memory architectures are supported (specifically: shared memory CPU systems and single-GPU setups).
- It is assumed that the evaluation order of the stencils within one spatial sweep is irrelevant. Also, always all points within the domain are traversed per sweep. One grid array is read and another array is written to. Such a grid traversal is called a Jacobi iteration. In particular, this rules out schemes with special traversal rules such as red-black Gauss-Seidel iterations.
- No extra boundary handling is applied. The stencil is applied to every interior grid point, but not to boundary points. I.e., the boundary values are kept constant. This corresponds to Dirichlet boundary conditions. To implement boundary conditions, they could be

factored into the stencil operation by means of extra coefficient grids. In the same way, non-rectilinear domain shapes and non-uniform grids can be emulated by providing the shape and/or geometry information encoded as coefficients in additional grids. Alternatively, special $(d-1)$-dimensional stencil kernels could be specified for the boundary treatment, which are invoked after the d-dimensional stencil kernel operating on the interior. This approach, however, will not allow to use temporal blocking schemes.

- The index calculation assumes that the stencil computation is carried out on a flat grid (or a grid which is homotopic to a flat grid). In particular, currently no cylindrical or torical geometries are implemented, which require modulo index calculations.

- There is no support for temporally blocked schemes yet.

5.6 Related Work

There are a number of research projects related to PATUS.

The Berkeley stencil auto-tuner [92] aims at converting Fortran 95 stencil code automatically into a tuned parallel implementation. It generalizes prior work by Williams [176] and Datta [53] who identified potential optimizations for stencil computations and implemented these optimizations manually or with simple code generation scripts specific for a selected stencil. Auto-tuning is applied to select the best blocking, loop unrolling, and software prefetching method configuration. The emphasis of the work of Kamil et al. in [92] is on fully automated translation and tuning process, including the replacement of the original code by the tuned version. In the user program, stencil loops are annotated, and thereby marked as such. The stencil expressions are then parsed and transformed into an abstract syntax tree (AST) on which the optimizations are performed as manipulations of the abstract syntax tree. From the abstract syntax tree, C, Fortran, and CUDA code can be generated. Currently, similarly to PATUS, the Berkeley auto-tuner supports compiler-type optimizations: loop unrolling, cache blocking, arithmetic simplification and constant propagation. In the future, better utilization of SIMD, common subexpression elimination, cache bypass, software prefetching will be integrated. The parallelization is based on domain decomposition (blocking), and optimizations relevant on parallel machines (in particular, NUMA-awareness) are addressed.

Auto-tuning involves building of a hardware-specific benchmarking harness for the extracted loops. In [92] only exhaustive search is mentioned as search method, which is applied after the search space is pruned based on hardware-specific heuristics. For instance, only block sizes are chosen such that the maximum number of threads is used. Speedups of around 2× over naïve implementations on traditional CPU systems for selected stencil kernels (gradient, divergence, and Laplacian) are reported.

Pochoir [155] is a stencil compiler developed at the MIT. The theoretic foundation is derived from a generalization of Frigo's and Strumpen's cache oblivious stencil algorithm [66, 67], cf. Chapter 6. This implies that Pochoir is designed for iterative stencil computations which execute many sweeps atomically. The aforementioned generalization consists in allowing *hyperspace cuts*, i.e., cutting multiple spatial dimensions at once while the original papers [66, 67] only did one space cut. This increases parallelism, yet does not impair the cache efficiency of the original algorithm.

The stencils are specified in a DSL embedded into C++, which offers the advantages that existing tool chains and IDEs can be used. Additionally, it allows Pochoir also to offer a non-optimized template-based implementation of the stencil which can be compiled without the Pochoir compiler. This feature simultaneously provides the mechanism for error checking and debugging. Similarly to PATUS, the stencil operation is specified in a per-point Pochoir *kernel function*. Since the Pochoir DSL is embedded into C++, the kernel function can contain arbitrary C++ code. Pochoir also provides a clean mechanism for handling boundary conditions. However, the stencil shape needs to be specified explicitly. The Pochoir compiler is a source-to-source translator, which applies the cache oblivious algorithm and parallelization (threading) using Cilk++ to the stencil specified in the Pochoir EDSL. Optionally, auto-tuning can be used to find the best trapezoid base size, i.e., the size of the trapezoids for which the recursion stops. Currently only one specific algorithm is supported (which requires multiple time steps so that it becomes beneficial), but the Pochoir EDSL was designed in a way flexible enough that it could be exchanged with other back-ends.

The authors report large speedups ($> 10\times$) compared to naïvely implemented code for a 2D 5-point stencil on large data sets (5000 × 5000 grid points) and an appropriately large number of time steps (5000).

Other frameworks for dynamically time-blocked stencil computations include *CORALS* [153] by Strzodka et al., and newer work by the team [152], and the stencil C++ class framework described in [154]. CORALS and other temporal blocking algorithms such as wave front parallelization [170] are described in more detail in Chapter 6.

Mint [163] targets NVIDIA GPUs as hardware platforms and translates traditional, but annotated, C code to CUDA C. The goal is to increase programmer productivity by introducing pragmas similar to those of OpenMP, but keeping them to a limited set of intuitive annotations (*parallel* for data parallel execution of the annotated code section, *for* for parallelizing a loop nest and doing the data decomposition, *barrier* for a global synchronization point, *single* if only one thread of execution (either the host or one GPU thread) is to execute the annotated section, and *copy* for copying data into and out of the GPU memory). The programmer is not required to have CUDA C knowledge, and knowledge of CUDA-specific optimizations in particular. The advantage of the pragma approach is that the code can be compiled and run on conventional hardware since a standard compiler ignores any Mint-specific pragmas. Mint establishes a simple data parallel model, which a priori is not limited to stencil computations, but if the pragmas are inserted into C stencil code programmed in a straight-forward fashion, the Mint stencil optimizer carries out domain-specific optimizations when stencil computations are recognized. The optimizations include finding a good data decomposition and thread block layout, usage of shared memory inferring memory requirements by analyzing the stencil structure, and the use of GPU registers. The authors report that Mint reaches around 80% of the performance of that of aggressively hand-optimized CUDA code.

Other CUDA code generators use the circular queue time blocking method to increase the arithmetic intensity, and elaborate register reuse schemes for data locality optimizations [110].

The *STARGATES* project [135] is somewhat different in that it is a code generator not only for stencil computations, but for entire simulations using finite difference discretizations of PDEs both in space and time. The finite difference solvers are built from a DSL, a mathematical description of a PDE. Modular back-ends generate code for the Intel Threading Building Blocks, MPI, and CUDA.

Panorama [99, 149] was a research compiler for tiling iterative stencil computations in order to minimize cache misses; the authors established a relation between the skewing vector and cache misses and a model predicting performance and cache misses, respectively.

More general approaches, not only limited to stencil computations, consider tiling of perfectly and imperfectly nested loops in the polyhedral model [141]. Loop transformation and (automatic parallelizing) compiler infrastructures in the polyhedral model include *CHiLL* [80] and *PLuTo* [23].

Automatic parallelization, code optimizations and auto-tuning are also being factored into existing solver packages. For instance, CHOMBO [8, 44] is an adaptive mesh refinement finite difference solver framework, which has been used, e.g., for hydrodynamics simulations. Currently, some auto-parallelization and auto-tuning efforts are undertaken within the software [36].

Chapter 6

Intermezzo: Saving Bandwidth And Synchronization

> There are many ways in which it may be desired in special cases to distribute and keep separate the numerical values of different parts of an algebraical formula; and the power of effecting such distributions to any extent is essential to the algebraical character of the Analytical Engine.
>
> — Ada Lovelace (1815–1852)

6.1 Spatial Blocking

As we showed previously, the performance of stencil computations (i.e., the number of stencil computations per time unit) is typically limited by the available bandwidth to the memory subsystem. Hence we are interested in maximizing the use of the available bandwidth or in enhancing the arithmetic intensity to increase performance.

The key idea in reducing memory transfers from and to slower memory in the memory hierarchy is to reuse the data that have already been transferred to the faster and closer memory — e.g., the cache memories on a CPU system, or the shared memory on NVIDIA GPUs.

Assume we are given a 3D 7-point stencil dependent only on its immediate neighbors along the axes. Then, if the values for a fixed $z = z_0$ have been loaded into the fast memory and they still reside in fast memory when moving to the next plane $z = z_0 + 1$, all of these data can be reused for the computation of the stencil on the points $(x, y, z_0 + 1)$, which depend on the data points (x, y, z_0). However, if domain sizes are large, it is likely that the points $(x, y, z_0 - 1)$ or even (x, y, z_0) have been evicted from the fast memory before they can be reused in iteration $z = z_0 + 1$, and they have to be transferred once more.

This can be prevented by working only on small blocks of the domain at a time, with the block sizes chosen such that a sub-plane of data can be kept in fast memory until it is reused, i.e., by tiling or *blocking* the spatial loop nest. Indeed, tiling has been a well-known method to improve temporal data locality [142, 181].

Usually it is beneficial to apply this idea recursively with decreasing block sizes to account for the multiple layers in the memory hierarchy (L2 cache, which is possibly shared among multiple cores, per core L1 cache, and registers).

The idea also lends itself to parallelization. In a shared memory environment which introduces a layer of spatial blocking in which each thread is assigned a distinct subset of blocks, the stencil sweep can be executed in parallel without any synchronization points within the sweep, since the stencil computation on one point is completely independent of the calculation of other points.

Fig. 6.1 shows a possible hierarchical decomposition of a cube-shaped domain into blocks. In level 1, each thread is assigned a *thread block*. Thread blocks are executed in parallel. Each thread block is decomposed into *cache blocks* to optimize for highest level cache data locality. When choosing the cache block sizes, there are a number of facts that have to be kept in mind. Typically, the highest level cache is shared among multiple threads (e.g., on an AMD Opteron Magny Cours the six cores on a socket share a 6 MB L3 cache, and each core has its own unified 512 KB L2 cache and its own 64 KB L1 data cache). It is not necessary that the cache holds all of the data in the cache block. It is sufficient if the neighboring planes of the plane being calculated fit into the cache. To avoid loading a plane twice, thread blocks should not be cut in z direction if it can be avoided.

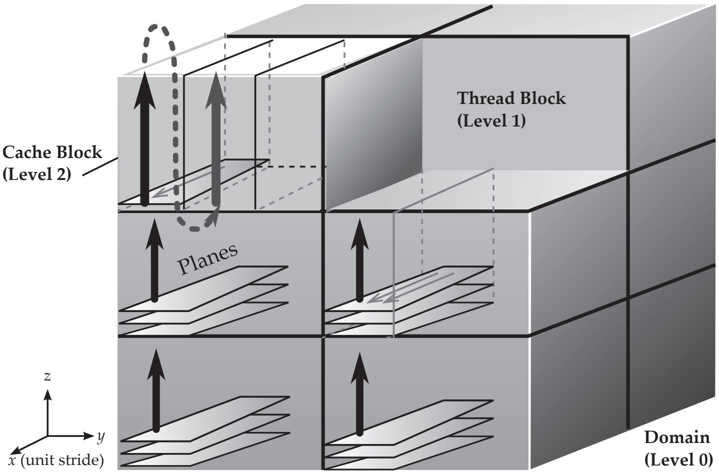


Figure 6.1: *Levels of blocking: decomposition of the domain into thread and cache blocks for parallelization and data locality optimization.*

6.2 Temporal Blocking

In iterative or time-dependent schemes, the stencil kernel does multiple sweeps over the entire grid. If intermediate results are of no interest, we can think of not only blocking the stencil computation in space, but also in the time dimension, i.e., blocking the outermost loop and thereby saving intermediate write-backs and loads — thus increasing temporal data locality, — as well as reducing the synchronization overhead. Instead of loading, writing, and synchronizing in every time step, these costly operations are only performed every t_{local} time steps. In this section several ways of parallelizing and handling the data dependences are discussed based on the idea of blocking in the temporal dimension. As temporal blocking methods reduce the number of data transfers, they increase the arithmetic intensity — by a factor of $\mathcal{O}(t_{\text{local}})$.

Although temporal blocking schemes have a high potential to increase performance — both data transfer operations in a bandwidth limited setting and synchronization are expensive operations — there are disadvantages to it. Firstly, the application must not request the state of the solution after every time step. Secondly, in order to integrate a temporally blocked compute kernel into an existing code, it typically has to be re-engineered since the outermost loop has to be altered. Also, since boundary conditions have to be applied after every time step, temporal blocking schemes require adaption to incorporate the boundary conditions dictated by the application and therefore they cannot be used in a plug-and-play fashion.

The PATUS software framework proposed in this thesis was designed to support temporal blocking schemes conceptually. However, the focus of the work is on generating stencil kernels that can easily be integrated into existing applications, and at the time of writing the PATUS code generator is still missing some essential parts to support temporal blocking schemes, which are planned to be completed in the future.

6.2.1 Time Skewing

Time skewing is a loop transformation which essentially modifies the iteration order such that temporal data locality is increased: by interchanging the temporal loop to an inner position, values computed in previous time steps can be reused through the entire iteration in time. The method requires that the number of time steps is known at runtime; alternatively the temporal iteration space can be blocked such that a constant number of time steps are performed per time block.

To keep the illustration of the method simple, consider a 1D 3-point stencil φ. We assume that the values $u[x;0]$ at time 0 as well as the boundary values $u[0;t], u[X+1;t]$ are given and set.

Listing 6.1: *A 1D example stencil.*

```
for $t$ = 0 .. $T-1$
    for $x$ = 1 .. $X$
        $u[x;t+1]$ = $\varphi(u[x-1;t], u[x;t], u[x+1;t])$
```

The t and the x loops cannot be interchanged due to data dependences: solving the equations $(x_1, t_1+1) = f(x_1, t_1) = g_j(x_2, t_2)$ with $g_j(x, t) = (x+j, t)$ for $j = -1, 0, 1$ as detailed in 3.4, we find the distance vectors

$$D = \left\{ \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ -1 \end{pmatrix} \right\}.$$

In the unimodular framework, loop interchange is represented by the matrix $\left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$. Multiplying the third distance vector by this matrix gives a new distance vector $(-1, 1)^\intercal$, which is not lexicographically positive, which in turn means that interchanging the loops is not legal.

However, we can skew the inner loop first and find a skewing factor such that interchanging becomes legal. The problem is to find $f \in \mathbb{Z}$ such that

$$\underbrace{\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}}_{\text{interchange}} \underbrace{\begin{pmatrix} 1 & 0 \\ f & 1 \end{pmatrix}}_{\text{skewing}} D = \left\{ \begin{pmatrix} f+1 \\ 1 \end{pmatrix}, \begin{pmatrix} f \\ 1 \end{pmatrix}, \begin{pmatrix} f-1 \\ 1 \end{pmatrix} \right\} \overset{!}{\succ} 0.$$

Clearly, the condition is fulfilled for $f = 1$. Hence, after skewing, the loop index x is replaced by $x' := t + x$, as

$$\begin{pmatrix} t' \\ x' \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} t \\ x \end{pmatrix} = \begin{pmatrix} t \\ t+x \end{pmatrix},$$

and the loop nest takes the following shape:

Listing 6.2: *Skewing the spatio-temporal loop nest.*

```
for $t' = 0 \;..\; T-1$
    for $x' = t'+1 \;..\; t'+X$
        $u[x'-t';t'+1] = \varphi(u[x'-t'-1;t'], u[x'-t';t'], u[x'-t'+1;t'])$
```

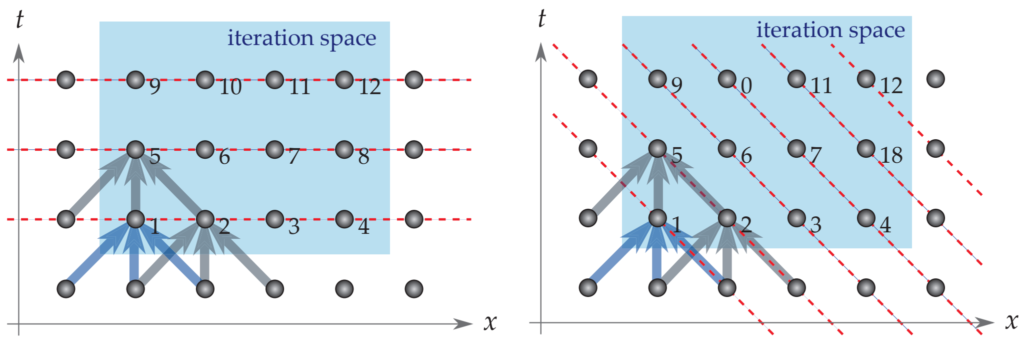
Interchanging the t' and x' is now legal, and we arrive at the loop nest shown in the next listing. Care has to be taken in adjusting the loop bounds, however. Since $x' = t + x$, for x' we now have $0 + 1 \leqslant x' \leqslant (T - 1) + X$. The more complicated bounds for the inner t' loop are derived by intersecting the original loop ranges.

Listing 6.3: *Exchanging temporal and spatial loops.*

```
for $x' = 1 \;..\; X+T-1$
    for $t' = \max(0, x'-X) \;..\; \min(x'-1, T-1)$
        $u[x'-t';t'+1] = \varphi(u[x'-t'-1;t'], u[x'-t';t'], u[x'-t'+1;t'])$
```

Fig. 6.2 shows the order of the iterates before skewing (a) and after skewing (b) for $X = 4$ and $T = 3$. Without skewing, one spatial sweep is completed before the proceeding to the next time step. After skewing, the spatio-temporal iteration space is traversed in diagonal lines shown in red. Note that the traversal order shown in (b) is indeed legal since the dependences, which are indicated by arrows, are respected.

The impact on data locality of time skewing is shown in Fig. 6.3. Assuming that only the values in the last time step are of interest, the intermediate values, shaded in gray, can be discarded as soon as they cannot



(a) (b)

Figure 6.2: *Traversal order before (a) and after (b) skewing. The numbers show the orders in which the data points are evaluated, the arrows symbolize the data dependences.*

be reused any longer. The outer iteration proceeds along the horizontal axis. One value, marked in blue and labeled “load”, is loaded from main memory. Having all the values in the area colored in blue in cache (in [107], McCalpin and Wonnacott propose a temporary array to hold these values), the values across *all* time steps symbolized with red dots and labeled “compute” can be computed in sequence, walking diagonally from bottom to top. Note that only the values in the blue sliding window are needed to satisfy the dependences doing one sweep along the skewed t'-axis. As soon as the inner sweep is completed, the oldest, left most values are not needed any longer and can be overwritten in the next iteration. The dark red “computed values” are written back to main memory.

The method also works in higher dimensions by interpreting the dots in Figs. 6.2 and 6.3 as lines (for 2D stencils) or planes (for 3D stencils) of data, which are loaded and computed *atomically*. This approach would do skewing in just one direction, which is the approach taken in [107, 184, 185]. Skewing can also be done in all directions [89].

Time skewing can be parallelized [107, 152]. The idea is illustrated in Fig. 6.4. Note that the parallelization cannot be straightforward, i.e., the outer loop cannot be *do-all* loop since no matrix $T \in GL_2(\mathbb{Z})$ exists such that the first components of all the transformed distance vectors are zero,

$$T \left\{ \binom{2}{1}, \binom{1}{1}, \binom{0}{1} \right\} \overset{!}{=} \binom{0}{*},$$

which, by Theorem 3.1, would mean that the outer loop were parallelizable. (The first component of the products is only zero if the first row of T

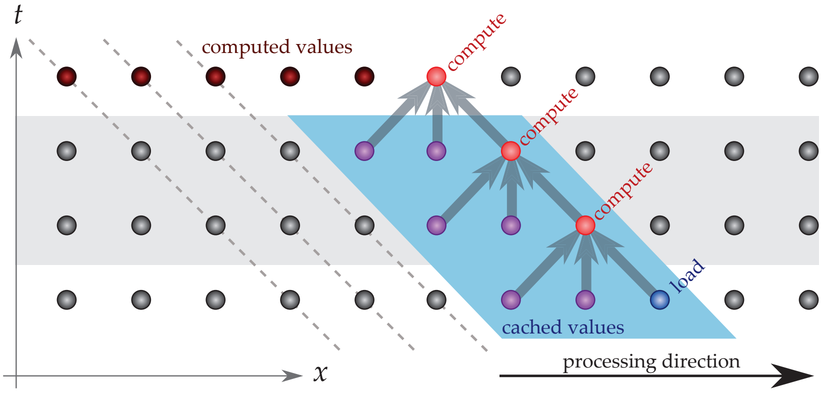


Figure 6.3: *Temporal data locality in time skewing: many stencil computations are done when only one point is loaded from main memory. The only values that need to be kept in fast local memory are symbolized by the points within the blue area.*

is zero, but then, clearly, $T \notin GL_2(\mathbb{Z})$.) This example shows that finding a good parallelization automatically is far from trivial. Note that the inner loop *is* parallelizable by Theorem 3.1, so an auto-parallelizing compiler would pick this loop for parallelization. However, because of the high synchronization overhead the performance will likely be inferior to the performance of the method described below.

The entire spatio-temporal iteration space is subdivided into blocks in the time dimension. In Fig. 6.4 only one time block is shown. A time block of the iteration space is then divided among the processes along the black diagonal lines. Due to this particular decomposition of the iteration space, each process can start computing without any dependences on values held by other processes. Per processor the same strategy applies as described above, with only keeping the data within the blue sliding window area in fast memory. The computation of the values within the red trapezoidal area on process p depend on values calculate by process $p+1$. Therefore process p has to wait until the values within the dark blue area on process $p+1$ have been computed, or, in a distributed memory setting, have been communicated to and received by process p. Note that if the processes work in a sufficiently synchronous manner and the data blocks in the spatial dimension (along the horizontal axis) are sufficiently large, the values on process $p+1$ required be process p, being the first values that are computed, are already available when process p reaches the *dependence area*.

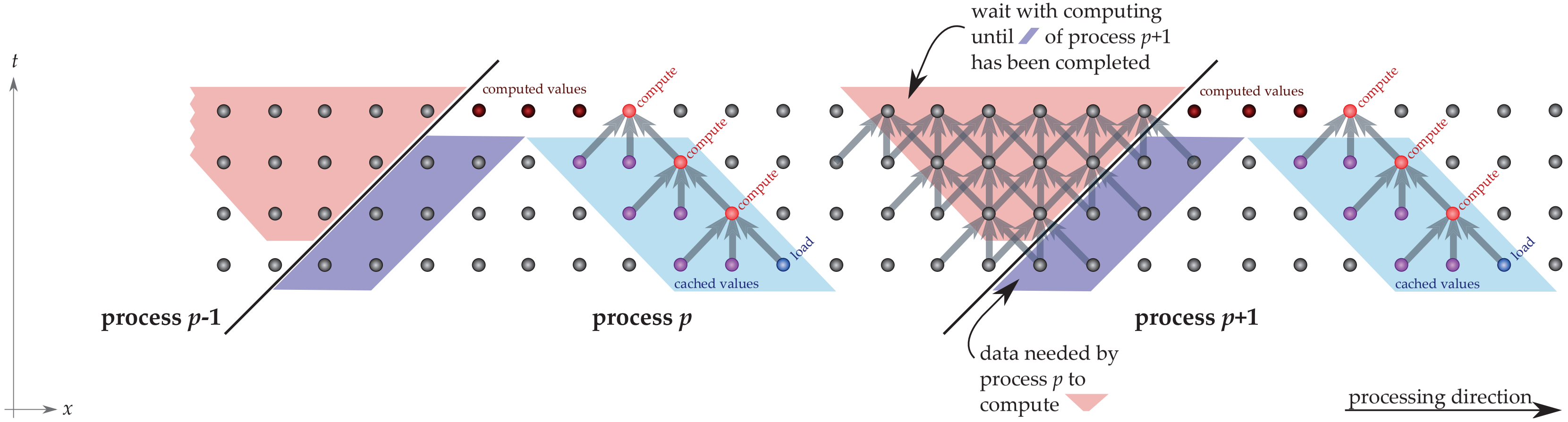


Figure 6.4: *Parallel time skewing. A time block is divided along the black diagonal lines among the processes. As in the sequential case, each processor only needs to cache the values within the blue sliding window area. Before computing the values in the red area, process* p *must wait for process* $p+1$ *to complete the computation of the values in the dark blue area.*

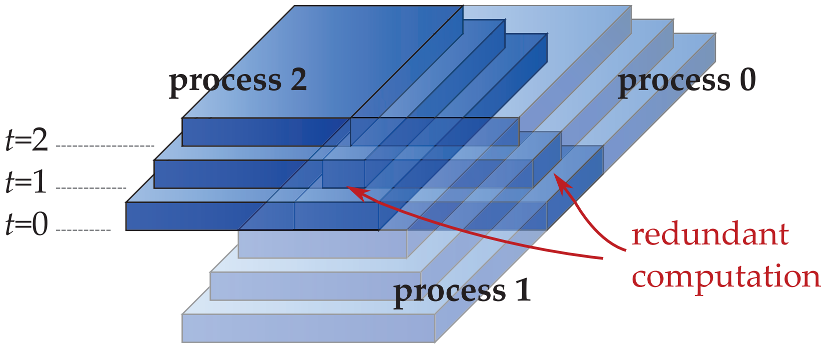


Figure 6.5: *The parallel circular queue method. Each process computes a number of time steps independently and without synchronizing at the cost of redundant computation at the artificial boundaries.*

6.2.2 Circular Queue Time Blocking

The idea behind the circular queue method is to use a more regular subdivision for parallelization compared to the one used in the parallel time skewing described above. Instead of skewing the spatial iteration space, overlapping "pyramids" of data are computed: Starting from a base tile at local time step 0, the tiles computable values shrink with each time step due to the data dependences as shown in Fig. 6.5. The upper most tiles computed in the last local time step have to be aligned such that the spatial domain is seamlessly tiled, which means that there are redundant computations at the inner artificial boundaries [177, 51]. This is the cost for a simpler code structure: no checking for data availability is needed during the computation of a block; the blocks can be worked on completely independently. The 3.5D blocking method described in [121] is proposes to stream z-planes (planes orthogonal to the z-axis, the axis with the slowest varying indices) through the local memory. But if circular queue time blocking is implemented reasonably, this is done anyway [40, 41, 42], so the methods are equivalent.

We want to maximize the number of local time steps as the arithmetic intensity increases about proportionally to the number of local time steps since we need to load only one input plane and store one output plane per set of local time steps. However, with each additional local time step the amount of redundant computation increases. Additionally, the size of the planes directly limits the number of time steps that can be performed simultaneously on the data, since the size of the intermediate planes decreases with each time step. Therefore, planes should be as large as pos-

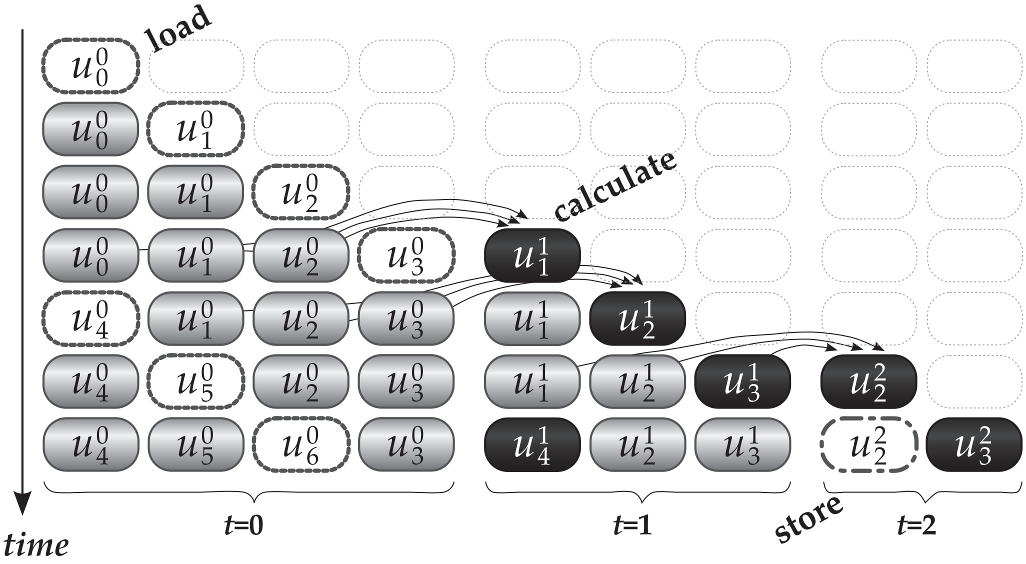


Figure 6.6: *Memory layout of planes in the time-blocking scheme.*

sible. On the other hand, with each additional local time step the number of intermediate planes that need to be kept in fast memory increases. The memory size limits the size of the planes. Hence a trade-off between the increased arithmetic intensity and the redundant computations must be found by determining the best number of local time steps.

In the shared memory environment, no explicit treatment of the artificial block boundary has to be carried out. We simply have to make sure that the overlaps of the input data of adjacent blocks are large enough so that the result planes are laid out seamlessly when written back to main memory. Distributed memory environments require the artificial boundaries to be exchanged, the width of which is multiplied by the number of time steps that are performed on the data between two synchronization points.

Boundary conditions have to be applied after each time step on the blocks that contain boundaries. Dirichlet boundary conditions (boundaries whose values do not change in time) can be easily implemented by simply copying the boundary values on the input plane forward to the planes belonging to the intermediate time levels. Note that, e.g., discretized Neumann or convective boundary conditions can be rewritten as Dirichlet boundary conditions, a process that involves only modifications of the stencil coefficients on the boundary [120].

Fig. 6.6 shows the layout of the planes u in memory for time block size 2. It shows the most compact memory layout that can be explicitly programmed on architectures with a software-managed cache (e.g., the shared memory of GPUs). The lower indices of u denote the spatial indices and the upper indices are the indices in time. The figure illustrates that the spaces in memory are filled as time advances (from top to bottom in the diagram): u_0^0, u_1^0, u_2^0, u_3^0 are loaded in succession (the white items symbolize loading of data into fast memory). In the fourth row, u_0^0, u_1^0, and u_2^0 are available, and the first time step u_1^1 can be computed. The memory locations that receive calculated data are drawn in black. In the next step, u_0^0 is no longer needed and can be overwritten to contain data of u_4^0, while simultaneously u_2^1 is calculated from u_1^0, u_2^0, and u_3^0. In the sixth row, after calculating u_3^1, the input planes to compute the first plane of the next second step, u_2^2, are available. As we want to overlap computation and communication, u_2^2 is written back from fast to slower memory only a step later (this can, of course, only be controlled in architectures which support explicit memory transfers). Note that for a given time in computation (i.e., on a specific row in the diagram), the spatial indices of the planes being calculated are skewed. For example, in the last row of the diagram, plane 4 of time step 1 (u_4^1) and plane 3 of time step 2 (u_3^2) are calculated.

This time blocking scheme was implemented on the Cell Broadband Engine Architecture [76] for the hyperthermia cancer treatment planning application (cf. Chapter 11.1), and the speedup when applying the circular queue time blocking method to the stencil kernel on that particular architecture has been found to be a factor of 2 [40, 41, 42]. In this application, numerous coefficient fields put a heavy pressure on the relatively small local stores, which was the main reason for the speedup limit.

The circular queue method has also been successfully adapted to GPUs [110]by using the shared memory or even devising a scheme how to explicitly reuse data through the register file, which is comparatively large on GPUs.

6.2.3 Wave Front Time Blocking

The idea of the wavefront parallelization [170, 162] is to have a team of threads cooperate on a set of planes of data and each thread computing one time step. Fig. 6.7 illustrates how the algorithm works. Thread 1 reads planes from the input array (depicted in blue) and after the computation writes its result plane (drawn in red) into a temporary array. The

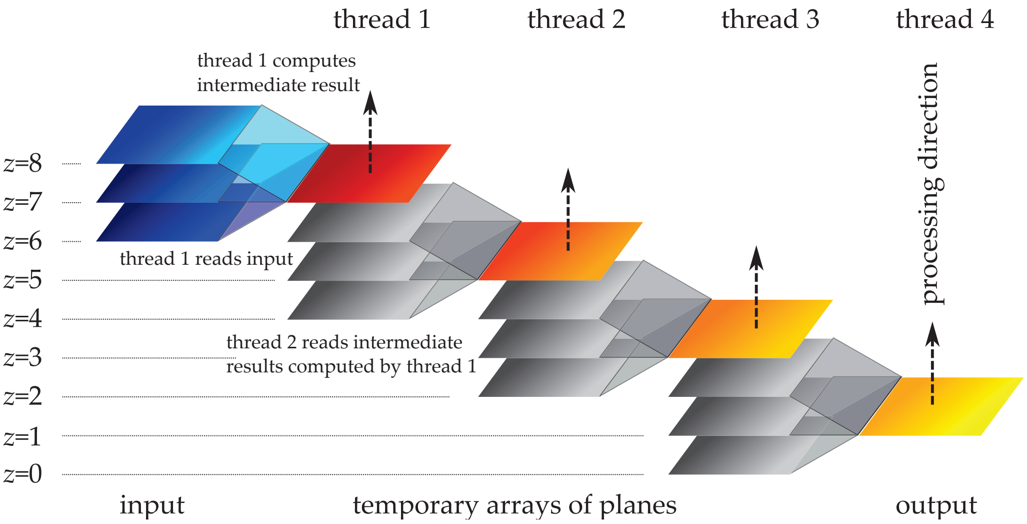


Figure 6.7: *Wavefront parallelization of a 7-point stencil in 3D. A team of* 4 *threads compute* 4 *time steps by having thread* i *perform a sweep on the data previously computed by thread* $i - 1$.

results produced previously by thread 1 are consumed by thread 2, which simultaneously computes the next time step. The last thread (thread 4 in Fig. 6.7) write the result to the output array, which can be identical to the input array, since the output lags behind and does not overwrite planes that are still needed as inputs for thread 1. To save temporary array space, intermediate threads can also write to the input array as long as the data that is still needed is not overwritten. The algorithm requires that all the threads within a team are synchronized after computing one plane.

If the threads work on planes that are cross-sections of the entire domain, each thread can apply boundary conditions before the result is consumed by the next thread. If the domain is too large to be handled by one thread team, the planes can be decomposed into smaller panels. Then, multiple thread teams can work on a stack of panels each, and there are two levels of parallelism. Handling the interior boundaries makes the algorithm more intricate, and the threads computing the lower local time steps have to load and compute redundantly, similarly as in the circular queue method, or they can avoid redundant computation by setting up a separate data structure holding the values on the artificial boundary. If redundant computation approach is chosen, a global synchronization point is required after each sweep.

6.3 Cache-Oblivious Blocking Algorithms

An algorithm is cache-oblivious if it is agnostic to concrete cache specifications such as the number of cache levels, the cache and cache line sizes, and yet makes optimal or near-optimal use of the cache hierarchy. The magic behind the concept is often a recursive divide-and-conquer-like recipe.

6.3.1 Cutting Trapezoids

Frigo and Strumpen present a cache-oblivious grid traversal algorithm for stencil computations [66] for a uniprocessor mapping which works by recursively dividing the trapezoid spanned by the spatio-temporal iteration space into two new trapezoids, which are obtained from cutting the old trapezoid through its center point either in space if the width is large enough so that both parts are again trapezoids, or from cutting the old trapezoid through its center point in the time dimension.

Again for the 1D case, let a trapezoid T be given by

$$T = (t_0, t_1, \mathbf{x}^0, \dot{\mathbf{x}}^0, \mathbf{x}^1, \dot{\mathbf{x}}^1),$$

where t_0 is the lower and t_1 the higher coordinate in time dimension (i.e., $t_0 < t_1$), $\mathbf{x}^0$ the lower and $\mathbf{x}^1$ higher spatial coordinate vectors ($\mathbf{x}_i^0 < \mathbf{x}_i^1$ for all i), and $\dot{\mathbf{x}}^0$ and $\dot{\mathbf{x}}^1$ are the slopes at $\mathbf{x}_i^0$ and $\mathbf{x}_i^1$, respectively. Let σ be the maximum slope between two consecutive time steps, e.g., $\sigma = 1$ for a 3D seven point stencil, or $\sigma = 2$ for a stencil depending on two neighbor points, etc. The exact process of cutting the trapezoids is given in Algorithm 6.1.

Algorithm 6.1: *A cache-oblivious blocking algorithm for stencil computations.*

```
procedure WALK(trapezoid $T = (t_0, t_1, \mathbf{x}^0, \dot{\mathbf{x}}^0, \mathbf{x}^1, \dot{\mathbf{x}}^1)$)
    $\Delta t \leftarrow t_1 - t_0$
    if $\Delta t = 1$ then
        evaluate stencil for all points in $T$
    else
        find the minimum $i$ such that $w_i \geqslant 2\sigma\Delta t$,
            where $w_i := \left(\mathbf{x}_i^1 - \mathbf{x}_i^0\right) + \frac{1}{2}\left(\dot{\mathbf{x}}_i^1 - \dot{\mathbf{x}}_i^0\right)\Delta t$ is the width
           in dimension $i$
        if $i$ exists then                    ▷ cut in space along dimension $i$
```

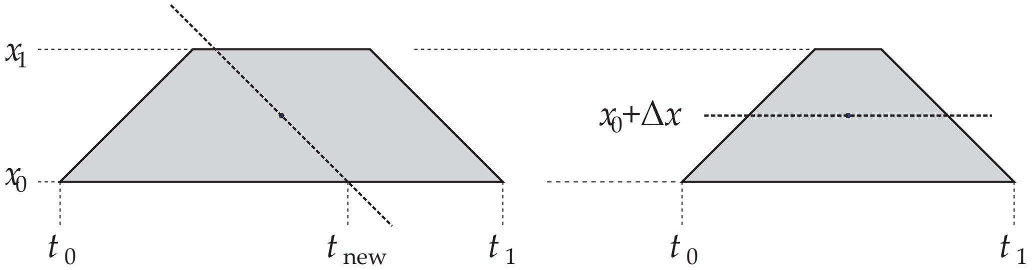


Figure 6.8: *Spatial (left) and temporal (right) cuts of a trapezoid in one spatial dimension.*

Algorithm 6.1: *A cache-oblivious blocking algorithm for stencil computations.* (cont.)

```
            $x_{\mathrm{new}} \leftarrow \frac{1}{2}\left(\mathbf{x}_i^1 + \mathbf{x}_i^0\right) + \frac{1}{4}\left(\dot{\mathbf{x}}_i^1 + \dot{\mathbf{x}}_i^0\right) + \frac{1}{2}\sigma\Delta t$   ▷ new base pt
            walk $(T[\mathbf{x}_i^1 \leftarrow x_{\mathrm{new}},\ \dot{\mathbf{x}}_i^1 \leftarrow -\sigma])$   ▷ walk "left" T first
            walk $(T[\mathbf{x}_i^0 \leftarrow x_{\mathrm{new}},\ \dot{\mathbf{x}}_i^0 \leftarrow -\sigma])$
        else   ▷ cut in the temporal dimension
            walk $(T[t_1 \leftarrow t_0 + \frac{1}{2}\Delta t])$   ▷ walk the $T$ with lower time
            walk $(T[t_0 \leftarrow t_0 + \frac{1}{2}\Delta t])$   ▷ coordinates first
        end if
    end if
end procedure
```

For 1D, cutting a trapezoid is illustrated in Fig. 6.8. The trapezoid on the left is wide enough so that a cut in the spatial cut is possible (here, $\sigma = 1$), whereas the width of the trapezoid on the right is too small, hence a cut in the spatial dimension (vertical axis) is performed as prescribed by the method.

In [66], Frigo and Strumpen generalize the method to arbitrary dimensions and prove that this method incurs $\mathcal{O}\left(\mathrm{Vol}(T)Z^{-\frac{1}{d}}\right)$ cache misses for an ideal cache of size Z. In [67, 155] a parallelization of the method is presented. The idea is to extend the spatial cuts such that multiple independent trapezoids and connecting dependent triangular shapes are obtained. The independent trapezoids can then be processed in parallel.

6.3.2 Cache-Oblivious Parallelograms

In [153], Strzodka et al. propose a cache-oblivious algorithm, *CORALS*, for temporal blocking of iterative stencil computations by considering the entire space-time iteration space and embedding it within a $(d+1)$-

dimensional parallelotope, which the algorithm subdivides recursively into 2^{d+1} smaller parallelotopes by halving each side. The observations are that parallelism can be extracted from the subdivision, which converges to the ideal speedup number as the number of subdivisions tends to infinity, and that assigning threads in a certain way guarantees good spatial and temporal data locality. Both a static and a dynamic assignment of threads are presented.

For illustration purposes, we show the algorithm for a 1D stencil with the static thread assignment. First, a parallelogram is chosen with width 2^w and height 2^h for some w and h such that the entire spatio-temporal iterations space of the stencil computation is covered by the parallelogram as shown in Fig. 6.9 (a). The powers of 2 are enforced to enable maximally many subdivisions.

The sub-figures (b) and (c) in Fig. 6.9 show the recipes proposed in [153] assigning numbers of threads to sub-parallelograms and steps of the recursive subdivision. After subdividing a parallelogram, the bottom left and the top right parts are assigned n threads, and the bottom right and top left parts $n/2$ threads. A parallelogram assigned one thread is not subdivided any further.

The main observation is that the top left and the bottom right sub-parallelograms in Fig. 6.9 (a) can be processed in parallel: The bottom right parallelogram obviously does not have any dependences, and can therefore be processed anytime, in particular after the bottom left parallelogram was processed. The top left parallelogram only depends on values from the bottom left parallelogram. Only one of the threads assigned to a parallelogram does the computation. Fig. 6.10 (a) shows the order in which the parallelograms are processed. Sub-parallelograms labeled with the same numbers are processed in parallel. In the example, 4 threads are used; Each of the 4 threads processes one of the sub-parallelograms with labels 5, 11, 13, 15, and 21. The remaining sub-parallelograms are not processed with maximum parallelism. However, as the number of subdivision steps tends to infinity, the parallelism becomes optimal. Fig. 6.10 (b) shows a possible assignment of threads to the sub-parallelograms; the labels are thread IDs. However, to process the original rectangular iteration space optimally, a more effective dynamic load balancing strategy is required, as shown in [153].

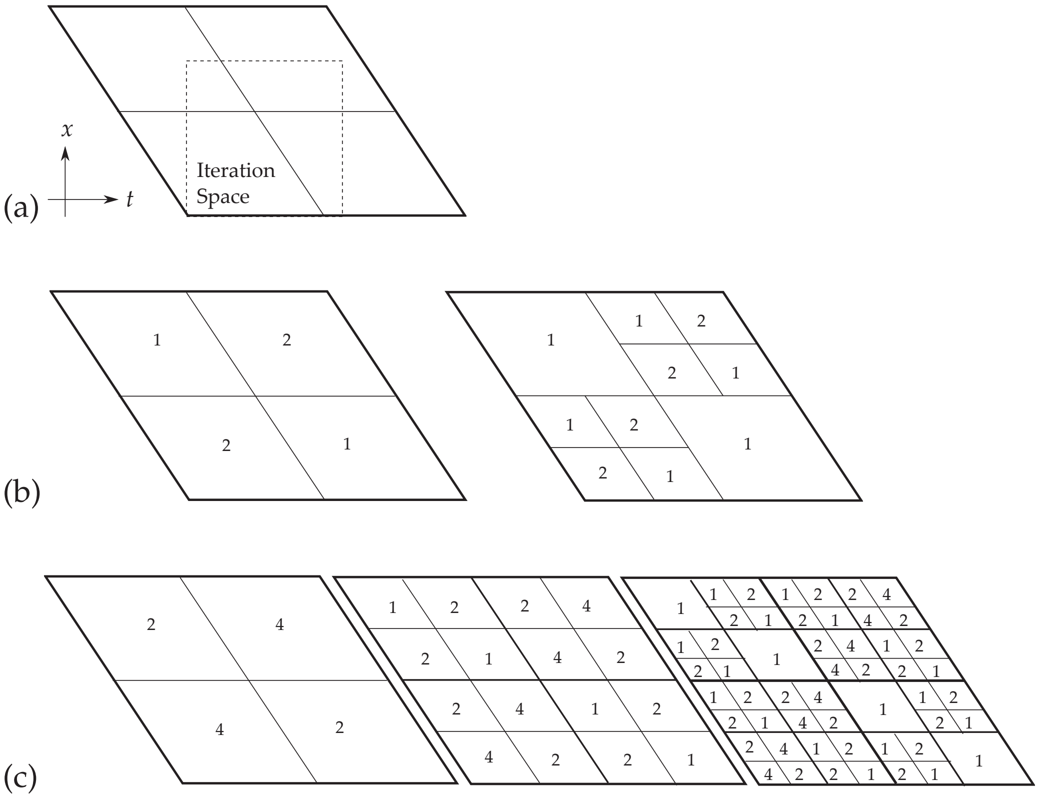


Figure 6.9: *(a) shows the spatio-temporal iteration space covered by a parallelogram. The spatial coordinate varies along the horizontal axis, the time along the vertical axis. (b) shows the recursive subdivision and the assignment of number of threads when using* 2 *threads. Similarly, (c) shows the recursive subdivision and assignment for* 4 *threads.*

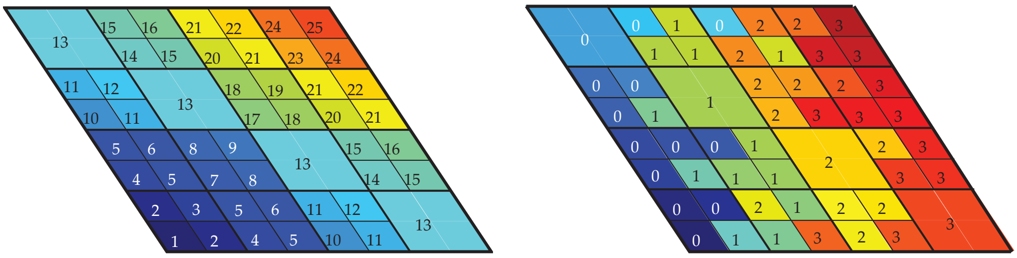

(a) (b)

Figure 6.10: *In (a), the sequence of the iteration over the parallelograms is shown when using* 4 *threads. Note that parallelograms with the same number are executed in parallel. (b) shows an assignment of parallelograms to threads (the numbers are thread IDs).*

6.4 Hardware-Aware Programming

In this section, techniques are presented which hardly qualify as algorithms, but rather are hardware-specific code optimizations which can significantly reduce memory bandwidth usage or bandwidth pollution.

6.4.1 Overlapping Computation and Communication

If cores share a common cache it can be beneficial — especially if few threads already exhaust the available memory bandwidth — to have two classes of threads: one class of threads doing the computation while the other class is responsible for bringing data from main memory into the cache shared with the compute thread(s). This approach guarantees a constant flow of data from the memory to the compute cores and thus maximizes memory throughput. Such a mechanism is built into the C++ framework detailed in [154].

In a distributed memory environment, usually a ghost layer is added at the block borders that replicates the nodes of the adjacent blocks in order to account for the data dependences. If multiple sweeps have to be run on the data, the ghost nodes need to be updated after each sweep. The simplest parallelization scheme simply synchronizes the execution after the sweep and exchanges the values on the artificial boundaries. A more elaborate scheme that allows overlapping the computation and the communication of the boundary values (and thus scales if the block sizes are large enough) is described in Algorithm 6.2.

Algorithm 6.2: *Scalable distributed memory parallelization.*

```
for each process do
    for $t = 0..t_{\max}$ do
        copy boundary data to send buffers
        initiate asynchronous transfers of the send buffers
            into the neighbors' receive buffers
        compute
        wait for the data transfers from the neighbors to complete
        update the boundary nodes with the data in the recv bufs
        apply boundary conditions
    end for
end for
```

The algorithm works with extra send and receive buffers. It is assumed that each process is assigned exactly one block. Before the actual computation, the data which have to be sent to the neighboring processes are copied to the send buffers and an asynchronous transfer is initiated, which copies the data from the send buffers into the neighbors' receive buffers while simultaneously the main computation is done. As soon as both the computation and the data transfers are complete, the nodes at the artificial boundaries are updated with the values from the receive buffers. Note that the update step in line 7 requires that the stencil expression can be separated into single components depending only on neighboring nodes in one direction and that the ghost zones are initialized with zero so that the nodes at the boundaries are not affected by the ghost zones when applying the regular stencil. Splitting can be done whenever the stencil computation is linear in the node values, e.g., if the stencil expression is a weighted sum over the values of the neighboring nodes. Note that there might be stencils for which the separation is not possible, and hence the algorithm is not applicable.

6.4.2 NUMA-Awareness and Thread Affinity

Currently, the majority of CPU clusters are, on the node level, multi-socket designs with cache coherent non-uniform memory architectures (cc-NUMA). This means that each socket or even a sub-entity of a socket has its own memory controller and DRAM, which can be accessed by other sockets, although doing so incurs a latency penalty since the data has to be transferred over the socket interconnect (Intel's Quick Path Interconnect or AMD's HyperTransport), and a bandwidth penalty, since the bandwidth to the DRAM module or the interconnect bandwidth has to be shared.

Ideally, if a NUMA architecture is used, each thread should not be allowed to migrate between sockets, i.e., threads should be pinned to the socket on which they were created or even to a single core to prevent cache thrashing and access of foreign DRAM modules (since the DRAM pages do not migrate along with the threads), and ideally they should only access the DRAM belonging to the socket they are running on. Also, going forward to exa-scale HPC systems, energy and power consumption are concerns that have to be addressed. On a small scale, placing data in memories as close as possible to the compute element, requires less energy than fetching data across the node and therefore is a contri-

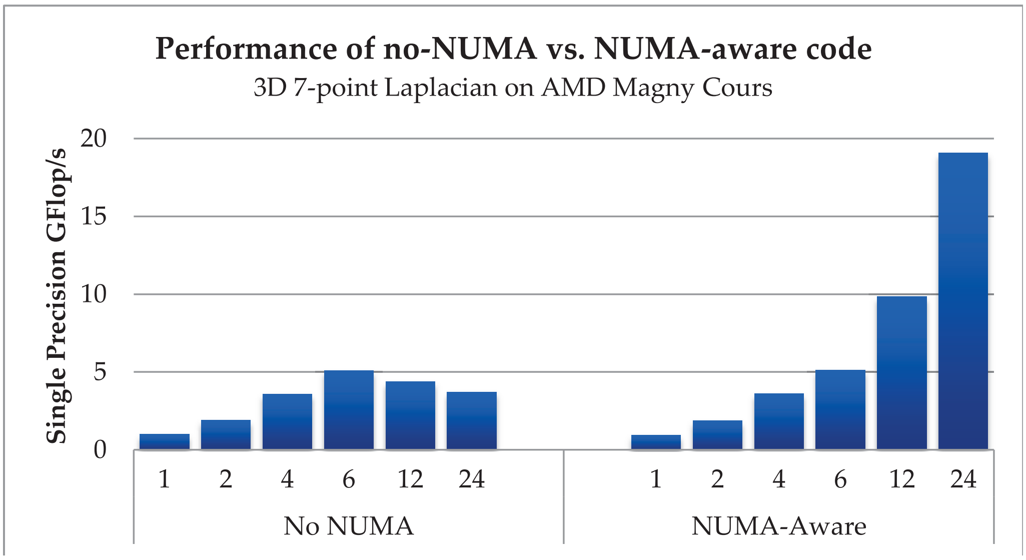


Figure 6.11: *NUMA-aware data initialization.*

bution to energy-aware programming.

Thread affinity can be set programmatically using the operating system API, and also most OpenMP implementations provide options to specify the thread affinity by means of setting environment variables*. Cray's `aprun` launcher pins threads automatically, by default to cores.

Memory affinity is harder to set programmatically. Typically, operating systems implement the *first touch* policy. I.e., the data will be placed in a page of the DRAM module belonging to the socket on which the thread is running which touches the data first. Hence it is important to to think carefully about the initialization of the data. Ideally, the data on which a thread computes should also be initialized by that thread. In that way, memory affinity is ensured implicitly on systems implementing the first touch policy.

Fig. 6.11 compares the performances of two versions of an example codes, once without and once with NUMA-aware data initialization. NUMA-aware initialization is not really a code optimization technique, but if it is possible to employ NUMA-awareness, there is a huge performance benefit on NUMA architectures.

---

*The GNU OpenMP implementation recognizes the `GOMP_CPU_AFFINITY` environment variable, which can be assigned a comma-separated core ID list; Intel's OpenMP implementation reads the `KMP_AFFINITY` environment variable, which can also be assigned a core ID list, or, more conveniently, a list of keywords describing the affinity pattern. In addition, the `GOMP_CPU_AFFINITY` variable is also recognized.

The left part of the figure shows the version of the code in which no NUMA-aware initialization of the data was done; the right part shows the scaling behavior with NUMA-awareness. The minor horizontal axis shows the number of threads used, and the vertical axis the performance in GFlop/s for the single precision 3D 7-point Laplacian stencil computation.

In the *No NUMA* version of the code, the grid arrays were set to zero using the stdlib C function `memset` after allocating (equivalently, `calloc` could have been used to allocate and initialize the data to zero). In contrast, in the NUMA-aware version, relying on the first-touch policy, the initialization was done in a loop nest parallelized identically to the compute loop nest.

The machine on which the benchmark was run is a dual-socket AMD Opteron Magny Cours system, cf. Chapter 9. Since one socket consists of two hexa core dies, each of which has its own memory interface, there are 4 NUMA domains. One to six threads ran on the same NUMA domain, so both parts of the figure with 1 to 6 threads show the same (linear) scaling behavior. In the result for 12 threads, the threads ran on the two dies of a single socket. Since all the data is initialized by one thread in the *No NUMA* case, all the data is owned by one NUMA domain, and the second die has to send data requests to the first die, which has to satisfy both memory requests of its own cores as well as of the other die's cores, and the data sent to the foreign cores additionally have to travel over the HyperTransport interconnect. As the bandwidth is exhausted, the result is the slowdown shown in the left part of Fig. 6.11. The situation becomes even worse when 4 dies send their data requests to the single die which has initialized and therefore owns the data. In the right part of the figure this problem does not occur since all the dies operate on data served by their own memory modules.

6.4.3 Bypassing the Cache

Another viable optimization is bypassing the cache entirely when data cannot be reused. While data *read* from main memory can be reused, data written back to main memory (at least in the last sweep) cannot. So-called *streaming* or *non-temporal* stores prevent that data being written back to main memory is first updated in the cache by transferring them from DRAM, and transferred back to the main memory again. This optimization is potentially valuable as without it, the so-called write allocation traffic (bringing data from DRAM into cache before writing back) doubles the amount of transferred data. Bypassing the cache is especially valuable in cc-NUMA systems because no additional bandwidth is consumed by the cache coherency protocol [176].

6.4.4 Software Prefetching

The idea of software prefetching is to give the processor hints about the data that will be used in a future computation. The processor might choose to act on these hints and bring the data into cache in advance. For stencil computations, the data pattern is known and static, so the data movement can be choreographed easily. However, in current processor architectures the hardware prefetchers have become so sophisticated that explicit software prefetching hints are hardly noticeable in terms of performance benefits [53, 176]. Yet, certain architectures require that data is brought into the local memories explicitly (e.g., into the shared memory of CUDA-based GPUs or into the local store of the Cell Broadband Engine Architecture's processing elements). In this context, software prefetching is also referred to as *double buffering*, since an additional data buffer is allocated into which the data are streamed before being consumed from the computation buffer by the computation.

Chapter 7

Stencils, Strategies, and Architectures

> We might even invent laws for series or formulæ in an arbitrary manner, and set the engine to work upon them, and thus deduce numerical results which we might not otherwise have thought of obtaining; but this would hardly perhaps in any instance be productive of any great practical utility, or calculated to rank higher than as a philosophical amusement.
>
> — Ada Lovelace (1815–1852)

7.1 More Details on Patus Stencil Specifications

In the current version, PATUS stencil specifications consist of 3 parts: the size of the domain, a d-dimensional grid, to the points of which the stencil is applied; a specification of the number of time steps (sweeps) that is performed within one stencil kernel call; and, most importantly, the *operation*: the definition of the actual stencil expression(s), which is applied

to each point of the grid(s) over which the computation sweeps.
The following listing shows a skeleton of a stencil specification:

```
stencil name
{
    domainsize = (x_min .. x_max, y_min .. y_max, ...);
    t_max = number of sweeps;

    operation (grid argument and parameter declarations)
    {
      stencil expressions
    }
}
```

Sweeps are not programmed explicitly in the stencil specification; the operation only contains a localized, point-wise stencil expression. Currently, PATUS supports Jacobi iterations, which means that the grids which are written to are distinct in memory from the grids that are read. In the future, other types of grid traversals, e.g., red-black Gauss-Seidel sweeps, may be supported. Given the Jacobi iteration as structural principle, there are no loop carried dependences within the spatial iteration over the grid. Hence, the order in which the grid points are visited and the stencil expressions evaluated does not matter. This fact is actually exploited by the Strategies.

Both the domain size and the number of sweeps can contain symbolic identifiers, in which case these identifiers are added as arguments to the stencil kernel function in the generated code. The dimensionality of the stencil is implicitly defined by the dimensionality of the domain size and the subscripts in the stencil expressions. Currently, it is not possible to mix grids of different dimensionalities. $x_{\min}, x_{\max}, \ldots$ can be integer literals, identifiers, or arithmetic expressions. *Number of sweeps* can be either an integer literal or an identifier if the number of time steps has to be controlled from outside the stencil kernel. Both minimum and maximum expressions in domain size specifications are inclusive, i.e., the domain size $(x_{\min}..x_{\max}, \ldots)$ has $x_{\max} - x_{\min} + 1$ points in x-direction.

There can be arbitrarily many grid arguments to the stencil operation, depending on the application. For instance, discretizing the divergence operator, which maps vector fields to a scalar function, might be implemented in the 3D case with three grids, which are read, and one grid to which the result is written. Grid arguments (in first line in the following listing) and parameter declarations (second line) have the following form:

Listing 7.1: *Operation arguments in the stencil specification.*

```
[const] (float|double) grid grid name ↵
      [( $x'_{\min}$ .. $x'_{\max}$ , $y'_{\min}$ .. $y'_{\max}$ , ... )] [[ number of grids ]]
(float|double) param parameter name [[ array size ]]
```

Both grids and parameters can either be based on single or double precision floating point data types. If a grid is declared to be `const`, it is assumed that the values do not change in time, i.e., the grid is read-only. The optional grid size specification after the grid name defines the size of the array as it is allocated in memory. This is useful if the kernel is to be built into an existing code. If no explicit size of the grid is specified, the size is calculated automatically by PATUS by taking the size of the iteration space `domainsize`, the inner domain, and adding the border layers required so that the stencil can be evaluated on each point of the inner domain. Again, $x'_{\min}, x'_{\max}, \ldots$ can be integer literals, identifiers, which can be distinct from the identifiers used for the `domainsize` specification, or arithmetic expressions. Any additional identifiers used to specify the grid size will be added as arguments to the stencil kernel function. Multiple grids can be subsumed in one grid identifier. Instead of using 3 distinct grid identifiers in the above example of the divergence operator, the inputs can be declared as `float grid X[3]` rather than, e.g., `float grid Xx, float grid Xy, float grid Xz`. In the generated code, an array of grids will be split into distinct grid identifiers, however. The *number of grids* has to be an integer literal. Also, parameters can be both scalar identifiers or arrays.

The body of the `operation` consists of one or more assignment expression statements. The left hand side of the assignments can be a grid access to which the right hand side is assigned or a temporary scalar variable, which can be used later in the computation. The arithmetic expressions on the right hand side involve previously calculated scalar variables or references to a grid point.

In Table 7.1 some flavors of assigning and referencing points in a grid are shown.

Assignment to a grid	`u[x,y,z; t+1] = …`
Assignment to temporary variables	`float tmp1 = …` `double tmp2 = …`

Referencing a `float grid` `u`	`// center point`
	`… = u[x,y,z; t] + …`
	`// right neighbor`
	`… = u[x+1,y,z; t] + …`
	`// center pt of prev time step`
	`… = u[x,y,z; t−1] + …`

Referencing an array `float grid` `X[3]`	`// center point of component 0` `… = X[x,y,z; t; 0] + …`
Referencing a `const float grid` `c`	`// center point of a const grid` `… = c[x,y,z] + …`
Referencing an array `const float grid` `c[3]`	`// center point of component 0` `… = c[x,y,z; 0] + …`

Table 7.1: *Assigning and referencing points in a grid in the stencil specification language.*

The center point of a stencil[*] is always denoted by `u[x,y,z; •]`, neighboring points by $\texttt{u[x}\pm\delta_x\texttt{, y}\pm\delta_y\texttt{, z}\pm\delta_z\texttt{; }\bullet\texttt{]}$ if `u` is the identifier of the grid we wish to access and the stencil is defined in 3 dimensions. 2-dimensional grids would only use `x` and `y` as spatial references, for arbitrary-dimensional stencils spatial reference identifiers `x0`, `x1`, . . . are defined. The $\delta_\bullet$ must be integer literals, i.e., the neighborhood relationship to the center point must be constant and known at compile time.

Non-constant grids, i.e., grids which change their values in the time dimension and are read from and written to, require an additional index denoting the time step, which is interpreted relatively to the time step of the corresponding left hand side grid. The temporal reference identifier is always `t`. Note that no references to future time steps can occur on the right hand side, i.e., the temporal indices of the grid references in expressions on the right hand side must be strictly less than the temporal

---

[*] Here, "center" always refers to the current point within a sweep. Sweeps are not explicitly programmed in the stencil specification as detailed above.

index on the left hand side. (If this was not the case, the method would be implicit, and a linear solver would be required to solve the problem.)

If an identifier has been declared to be an array of grids, an additional index has to be appended after the temporal one, which determines which array element to use. The index has to be an integer literal.

The complete grammar of the stencil specification DSL is given in Appendix B. Also, more examples of stencil specifications — the ones for which benchmarking results are given in Chapter 10 and the stencils occurring in the applications discussed in Chapter 11 — can be found in Appendix C.

7.2 Strategy Examples and Hardware Architecture Considerations

A PATUS Strategy is the means by which we aspire to implement parallelization and bandwidth-optimizing methods such as the ones discussed in Chapter 6. In practice it mostly is at least cumbersome to adapt an implementation for a toy stencil (e.g., a 3D 7-point constant coefficient Laplacian) to a stencil occurring in an application, since the implementation of the algorithm most likely depends on the shape of the stencil and the grid arrays used, and very probably on the dimensionality of the stencil. Extending a coded template to incorporate a stencil computation for which it was not designed initially is tedious and error prone.

The idea of Strategies is to provide a clean mechanism which separates the implementation of parallelization and bandwidth-optimizing methods from the actual stencil computation. In this way, the implementation of the algorithm can be reused for arbitrary stencils.

7.2.1 A Cache Blocking Strategy

We start by looking at the implementations of a cache blocking method. The Strategy in Listing 7.2 iterates over all the time steps in the `t` loop, and within one time step in blocks `v` of size `cb` over the *root domain* `u`, i.e., the entire domain to which to apply the stencil. Both the root domain and the size of the subdomain `v` are given as Strategy parameters. The blocks `v` are executed in parallel by virtue of the `parallel` keyword, which means that the subdomains `v` are dealt out in a cyclic fashion to the worker threads. The parameter `chunk` to the `schedule` keyword defines how many consecutive blocks one thread is given. Then, the stencil is applied for each point in the subdomain `v`.

The Strategy argument `cb` has a specifier, `auto`, which means that this parameter will be interfaced with the auto-tuner: it is exposed on the command line of the benchmarking harness so that the auto-tuner can provide values for `cb`$= (c_1, c_2, \ldots, c_d)$, where d is the dimensionality of the stencil, and pick the one for which the best performance is measured.

Listing 7.2: *A cache blocking Strategy implementation.*

```
strategy cacheblocked (domain u, auto dim cb,
  auto int chunk)
{
  // iterate over time steps
  for t = 1 .. stencil.t_max
  {
    // iterate over subdomain
    for subdomain v(cb) in u(:;t) parallel schedule chunk
    {
      // calculate the stencil for each point
      // in the subdomain
      for point p in v(:; t)
        v[p; t+1] = stencil (v[p; t]);
    }
  }
}
```

In the Strategy of Listing 7.2, the parameter `cb` controls both the granularity of the parallelization and consequently the load balancing, and the size of the cache blocks. The size of the blocks ultimately limits the number of threads participating in the computation. If the blocks become too large the entire domain will be divided into less subdomains than there are threads available, and the performance will drop. In practice, the auto-tuner will prevent such cases. But the consequence is that a configuration for `cb`, which runs well for a specific number of threads might not perform well for another number of threads.

The Strategy in Listing 7.3 this one block `v` was split into smaller blocks `w`. Here, the idea is that `v` is responsible for the parallelization and load balancing, while the inner subdomain `w` is reserved for cache blocking.

Listing 7.3: *Another way of subdividing for cache blocking.*

```
strategy cacheblocked2 (domain u,
  auto dim tb, auto dim cb, auto int chunk)
{
  // iterate over time steps
  for t = 1 .. stencil.t_max
  {
    // parallelization
    for subdomain v(tb) in u(:;t) parallel schedule chunk
    {
      // cache blocking
      for subdomain w(cb) in v(:; t)
      {
        for point pt in w(:; t)
          w[pt; t+1] = stencil (w[pt; t]);
      }
    }
  }
}
```

Although this is currently not done yet, restrictions could be inferred for `w`, limiting the search space the auto-tuner has to explore: Since `w` is an iterator within the subdomain `v`, we could restrict `cb` by $\texttt{cb}_\bullet \leqslant \texttt{tb}_\bullet$ (where `tb` is the size of `v`).

Also, we could infer a restriction preventing that threads are sitting idle by calculating the number of blocks `v`: Let $(s_1, \ldots, s_d)$ be the size of the root domain `u`. Then we could require that the following inequality holds:

$$\prod_{i=1}^{d} \left\lceil \frac{s_i}{\texttt{tb}_i} \right\rceil \geqslant T,$$

where T is the number of threads.

7.2.2 Independence of the Stencil

By concept, Strategies are designed to be independent of both the stencil and the concrete hardware platform. The obvious means to achieve independence of the stencil is to not specify the actual calculation in the Strategy, but have a reference to it instead. This is done by the formal `stencil` call, which expects a grid reference as argument (actually, a point within a domain) and assigns the result to another grid reference. Strategies therefore also must have the notion of grids, but interpreted as index spaces rather than actual mappings to values. Each Strategy has one required argument of type `domain`, the *root domain*, which represents the entire index space over which the Strategy is supposed to iterate. In the Strategy, this `domain` can be subdivided, defining the way in which the actual stencil grids are traversed. This is done using *subdomain iterators*, which are constructs that advance a smaller box, the iterator, within a larger one, its parent domain. The size of a Strategy domain is always the size of the computed output, i.e., not including the border layers required for a stencil computation.

Being independent of the stencil means in particular being independent of the stencil's dimensionality. The root domain inherits the dimensionality of the stencil. Subdomains, however, need a notion of the dimensionality in order to specify their size. Strategies provide data types `dim` and `codim`(n), which can be used in Strategy arguments. If the stencil dimensionality is d, a `dim` variable is a d-dimensional vector, and a `codim`(n) variable is a $(d - n)$-dimensional vector (a vector of codimension n).

Again consider the cache blocking Strategy in Listing 7.2. The size of the subdomain `v` being iterated over the root domain `u` is specified by `cb`, an argument to the strategy of type `dim`. This means that the subdomain `v` will have the same dimensionality as the stencil and the root domain.

The instruments provided by Strategies to deal with the unknown dimensionalities are the following:

- subdomain iterators rather than simple `for` loops,
- `dim` and `codim`(n) typed variables,
- subdomain and stencil properties: when a Strategy is parsed, both w`.dim` and `stencil.dim` are replaced by the dimensionality of the stencil and of the subdomain w, respectively,
- a subdomain's `size` property, which is a `dim`-typed variable containing the size of a subdomain (i.e., w`.size` is a d-dimensional vector with its components set to the size of w),
- subscripting `dim` or `codim`(n) type variables by an index vector. An index vector can be either

- a single integer (e.g., the value of w`.size(1)` is the first component of the size vector of w);
- a vector of integers (e.g., w`.size(1,2,4)` returns a 3-dimensional vector containing the first, second, and fourth component of the size of w);
- a range (e.g., w`.size(2 .. 4)` returns a 3-dimensional vector containing the second, third, and fourth component of the size of w, or w`.size(1 ..` w`.dim-1)` returns a vector containing all the components of the size of w except the last one);
- ellipses, which fill a vector in a non-greedy fashion so that the vector is of dimension `stencil.dim`:
  - `(`a `...)` is a d-dimensional vector with each component set to a (a must be a compile-time constant);
  - `(`a`,` b `...)` is a d-dimensional vector with the first component set to a, and all the others to b;
  - `(`a`,` b `...` c`)` is a vector with the first component set to a and the last set to c, and all the components in the middle set to b;
- any combinations of the above.

The size of a subdomain might also depend on the structure or the *bounding box* of a stencil. This can be achieved by the `stencil.box` property, which can be used to enlarge a subdomain.

7.2.3 Circular Queue Time Blocking

In the following, a circular queue time blocking method is shown how it is envisioned as a Strategy implementation. In the current state at the time of writing, the code generator still lacks certain mechanisms required for generation of the final C code.

The algorithm is in the spirit of [40, 41, 121]. For the sake of simplicity of presentation, no pre-loading scheme is implemented. The parameters to tune for is the number of local time steps `timeblocksize` in the inner time blocked iteration, and the cache block size `cb`. The implementation assumes that the stencil only requires the data of time step t in order to compute the result at time step $t + 1$.

Listing 7.4: *A circular queue time blocking Strategy implementation.*

```
strategy circularqueue (domain u, auto int timeblocksize,
  auto dim cb)
{
  int lower = -stencil.min(stencil.dim);
  int upper = stencil.max(stencil.dim);

  // number of regular planes
  int numplanes = lower + upper + 1;

  for t = 1 .. stencil.t_max by timeblocksize
  {
    for subdomain v(cb) in u parallel
    {
      // local timesteps 0 .. last-1
      domain pln(
        v.size(1 .. stencil.dim-1) +
          timeblocksize*stencil.box(1..stencil.dim-1), 1;
        timeblocksize-1;
        lower+upper+1);
      // last timestep
      domain pnlL(v.size(1 .. stencil.dim-1), 1);

      // preloading phase and filling phase omitted...

      // working phase
      for z = (timeblocksize-1)*upper + 1..v.max(v.dim)-2
      {
        // load
        memcpy (
          pnl[:; 0; (z + upper + 1) % numplanes0],
          v(:+timeblocksize*stencil.box(1..stencil.dim-1),
            z + upper + 1)
        );

        // compute
        for t0 = 0 .. timeblocksize - 2
        {
          int idx = (z - t0 * upper) % numplanes;
          for point pt in v(
            : + (timeblocksize-t0-1) *
              stencil.box(1..stencil.dim-1), 1;
            t+t0)
          {
            pnl[pt;t0;idx] = stencil (pnl[pt;t0-1;idx]);
```

Listing 7.4: *A circular queue time blocking Strategy implementation.* (cont.)

```
          }
        }
        for point pt in v(:, 1; t+t0)
        {
          pnlL[pt] = stencil (pnl[
            pt;
            timeblocksize - 2;
            (z - (timeblocksize-1)*upper) % numplanes]);
        }

        // write back
        memcpy (v(:, z-(timeblocksize-1)*upper-1), pnlL);
      }

      // draining phase omitted...
    }
    // implicit synchronization point from parallel v loop
  }
}
```

The d-dimensional root domain is cut into $(d-1)$-dimensional slices orthogonal to the last axis, called *planes* in the following and in Listing 7.4. The number of planes required is given by the shape of the stencil. For a calculation of one output plane, `lower+upper+1` planes are required, where `lower` is the number of stencil nodes below the center node, and `upper` the number of nodes above the center.

The outer temporal `t` loop iterates over all the time steps, increasing the global time step by the `timeblocksize`, the number of inner temporal iterations. The actual algorithm applies to the subdomain `v`. The algorithm is a sort of software pipelining; the pipelined operations are load-compute-store blocks. For brevity, Listing 7.4 only shows the steady-state phase of the pipeline. `pnl` and `pnlL` are sets of temporary data buffers. `pnlL` is the plane into which the last local time step is written, and which is copied back to the main memory. Within the `z` loop, which processes all the planes in the subdomain `v`, data is loaded from `v` into the temporary buffer `pnl` and consumed in the compute loops. Data at the border is loaded and computed redundantly and in an overlapping fashion with respect to other subdomains `v`. To express this, the subdomain on which the stencil is evaluated is enlarged by the expression

```
v(: + (timeblocksize − t0 − 1) *
  stencil.box(1 .. stencil.dim−1), 1;
  t+t0)
```

which means that the first $d-1$ coordinates of bounding box of the stencil is added `(timeblocksize-t0-1)`-times to the size of `v`, while the size in the last dimension remains 1. There is an implicit synchronization point after the parallel `v` loop.

7.2.4 Independence of the Hardware Architecture

A Strategy's independence of the hardware architecture is given by the notions of general iterators over subdomains and generic data copy operations, and, more importantly, by the liberty of interpreting the parallelism defined in a strategy as a *may parallelism* rather than a *must parallelism*.

The underlying model of the hardware is a simple hierarchical model, to some extent inspired by the OpenCL execution model [95]. The execution units are indexed by multi-dimensional indices, similar to OpenCL's *NDRange* index spaces. This guarantees that there is an optimal mapping to architectures that have hardware support for multi-dimensional indexing or have multidimensional indices built into the programming model such as CUDA or OpenCL. We call a level in the hierarchy a *parallelism level*. The dimension of the indices may differ in each parallelism level.

Each parallelism level entity can have its own local memory, which is only visible within and below that parallelism level. We allow the data transfers to the local memories to be either implicit or explicit, i.e., managed by hardware or software. Furthermore, we permit both synchronous and asynchronous data transfers.

According to this model, a shared-memory CPU architecture has one parallelism level with local memory (cache) with implicit data transfer, and a CUDA-capable GPU has two parallelism levels, streaming multiprocessors and streaming processors – or thread blocks and threads, respectively. The thread block level has an explicit transfer local memory, namely the per-multiprocessor shared on-chip memory.

An architecture description together with the PATUS back-end code generators specific for a hardware platform are responsible for the correct mapping to the programming model. In particular, nested parallelism within a Strategy is mapped to subsequent parallelism levels in the model. If a hardware platform has less parallelism levels than given in a Strategy, the parallel Strategy entities will be just mapped onto the last parallelism level of the architecture and executed sequentially.

Domain decomposition and mapping to the hardware is implicitly given by a Strategy. Every subdomain iterator, e.g.,

```
for subdomain v(size_v) in u(:; t)
  ...
```

decomposes the domain u into subdomains of smaller size `size_v`. When, in addition, the `parallel` keyword is used on a subdomain iterator, the loop is assigned to the next parallelism level (if there is one available), and each of the iterator boxes is assigned to an execution unit on that parallelism level. All the loops within the iterator also belong to the same parallelism level (i.e., are executed by the units on that level), until another parallel loop is encountered in the loop nest.

If a parallelism level requests explicit data copies, memory objects are allocated for an iterator box as "late" as possible: since local memories tend to be small, the iterator within the parallelism level with the smallest boxes, i.e., as deep down in the loop nest as possible (such that the loop still belongs to the parallelism level), is selected to be associated with the memory objects. The sizes of the memory objects are derived from the box size of that iterator, and data from the memory objects associated with the parallelism level above are transferred. In the case that the iterator contains a stencil call within a nested loop on the same parallelism level, the iterator immediately above a point iterator

```
for point p in v(:; t)
  ...
```

is selected, if there is one, or if there is no such iterator, the iterator containing the stencil call is selected.

The Strategy excerpt

```
for subdomain v(size_v) in u(:; t) parallel
  for subdomain w(size_w) in v(:; t) parallel
    for point p in w(:; t)
      ...
```

and the resulting data decomposition and the ideal hardware mapping are visualized in Fig. 7.1. The upper layer shows the hierarchic domain decomposition of `u` into `v` and `w`. The bottom layer shows an abstraction of the hardware architecture with 2 parallelism levels, work groups and

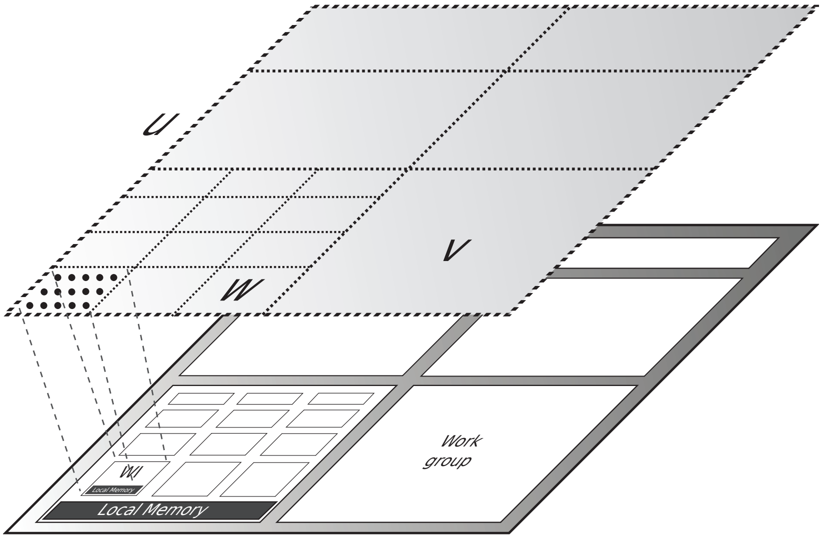


Figure 7.1: *Mapping between data and hardware. Both hardware architecture (bottom layer) and data (top layer) are viewed hierarchically: the domain* `u` *is subdivided into* `v` *and* `w`*, the hardware architecture groups parallel execution units on multiple levels together.*

work items, which both can have a local memory. By making the `v` loop in the Strategy `parallel`, it is assigned to the first parallelism level, labeled *work groups* in the figure (following the OpenCL nomenclature). And by making the nested `w` loop `parallel`, this, in turn, is assigned to the second parallelism level, the work items within a work group. The points `p` in `w` are all assigned to the same work item that "owns" `w`.

7.2.5 Examples of Generated Code

To conclude this chapter, we show two examples how a simple strategy iterator is mapped to code, once for a OpenMP-parallelized shared memory CPU system, and once for a CUDA-programmed NVIDIA GPU.

Listing 7.5 shows an example of a generated C code using OpenMP for parallelization. It was generated for a 3D stencil from the parallel Strategy subdomain iterator

```
for subdomain v(cb) in u(:; t) parallel
   ...
```

OpenMP provides one level of one-dimensional indexing (by the thread number, `omp_get_thread_num ()`), but the domain to decomposed is 3-

dimensional. Thus, the 3-dimensional index range

$$[\texttt{vidxx}, \texttt{vidxxmax}] \times [\texttt{vidxy}, \texttt{vidxymax}] \times [\texttt{vidxz}, \texttt{vidxzmax}]$$

is calculated based on the thread ID. By incrementing the loop index `v_idx` by the number of threads, the v blocks are dealt out cyclically.

Listing 7.5: *C/OpenMP code generated for a 3D stencil.*

```c
int dimidx0 = omp_get_thread_num();
int dimsize0 = omp_get_num_threads();
int v_numblocks =
  ((x_max+cb_x-1)/cb_x)*((y_max+cb_y-1)/cb_y)*
  ((z_max+cb_z-1)/cb_z);
for (v_idx=dimidx0; v_idx <= v_numblocks-1;
  v_idx += dimsize0)
{
  tmp_stride_0z = ((x_max+cb_x-1)/cb_x)*
    ((y_max+cb_y-1)/cb_y);
  v_idx_z = v_idx/tmp_stride_0z;
  tmpidxc0 = v_idx-(v_idx_z*tmp_stride_0z);
  v_idx_y = tmpidxc0/((x_max+cb_x-1)/cb_x);
  tmpidxc0 -= v_idx_y*((x_max+cb_x-1)/cb_x);
  v_idx_x = tmpidxc0;
  v_idx_x = v_idx_x*cb_x+1;
  v_idx_x_max = min(v_idx_x+cb_x, x_max+1);
  v_idx_y = v_idx_y*cb_y+1;
  v_idx_y_max = min(v_idx_y+cb_y, y_max+1);
  v_idx_z = v_idx_z*cb_z+1;
  v_idx_z_max = min(v_idx_z+cb_z, z_max+1);

  // inner loops/computation within
  // [v_idx_x, v_idx_x_max] x ...
}
```

In contrast, CUDA provides 2 levels of indexing, the thread block and the thread level. Moreover, the indices on the thread block level can be 1 or 2-dimensional (prior to CUDA 4.0) or 1, 2, or 3-dimensional in CUDA 4.0 on a Fermi GPU, and thread indices can be 1, 2, or 3-dimensional. Also, a GPU being a manycore architecture, we assume that there are enough threads to compute the iterates in v completely in parallel. Hence, there is no loop iterating over the domain, but a conditional guarding the “iteration” space instead as shown in Listing 7.6.

Listing 7.6: *C for CUDA code generated for a 3D stencil.*

```
stride_1 = (blockDim.y+y_max-1)/blockDim.y;
tmp = blockIdx.y;
idx_1_2 = tmp/stride_1;
tmp -= idx_1_2*stride_1;
idx_1_1 = tmp;
v_idx_x = 1+(blockDim.x*blockIdx.x+threadIdx.x)*cbx;
v_idx_x_max = v_idx_x+cbx;
v_idx_y = threadIdx.y+idx_1_1*blockDim.y+1;
v_idx_y_max = v_idx_y+1;
v_idx_z = threadIdx.z+idx_1_2*blockDim.z+1;
v_idx_z_max = v_idx_z+1;

if ((((v_idx_x<=x_max)&&(v_idx_y<=y_max))&&
  (v_idx_z<=z_max)))
{
  // inner loops/computation within
  // [v_idx_x, v_idx_x_max] x ...
}
```

Again, the 3-dimensional index range

$$[\texttt{vidxx}, \texttt{vidxxmax}] \times [\texttt{vidxy}, \texttt{vidxymax}] \times [\texttt{vidxz}, \texttt{vidxzmax}]$$

is calculated before the actual calculation, but now based on the 2-dimensional thread block indices (`blockIdx.x, blockIdx.y`) and the 3-dimensional thread indices (`threadIdx.x, threadIdx.y, threadIdx.z`). `blockIdx` and `threadIdx` are built-in variables in C for CUDA.

Chapter 8

Auto-Tuning

> In almost every computation a great variety of arrangements for the succession of the processes is possible, and various considerations must influence the selections amongst them for the purposes of a calculating engine. One essential object is to choose that arrangement which shall tend to reduce to a minimum the time necessary for completing the calculation.
>
> — Ada Lovelace (1815–1852)

8.1 Why Auto-Tuning?

Automated tuning or *auto-tuning* is the use of search to select the best performing code variant and parameter configuration from a set of possible versions. This means that a benchmarking executable is built and actually run on the target machine. The code versions can range from valid re-parameterizations that have an impact on cache behavior, — in the context of PATUS finding the best combination of Strategy parameters — to code transformations as described in Chapter 3.4, which can be done using dedicated source-to-source transformation frameworks such as CHiLL [80], or also by determining the best configuration for an optimizing compiler [129], to data structure transformations, or even to different algorithmic design, i.e., substituting the original algorithm with a “smarter” variant, which performs better under the given circumstances. In PATUS this would mean exchanging a Strategy by another one and finding the one with the best overall performance behavior.

Another — famous — example for the latter would be to replace the naïve $\mathcal{O}(N^2)$ N-body algorithm by a tree code such as the $\mathcal{O}(N \log N)$ Barnes-Hut algorithm or the $\mathcal{O}(N)$ Fast Multipole Method (FMM). All three are approaches to solve the same problem, so the results, given the same input set, will be equivalent. The performance, however, is dependent on the size of the input set: for small problem sizes the $\mathcal{O}(N^2)$ method will win because in this case Barnes-Hut and FMM have non-negligible overheads associated with them (in other words, albeit being $\mathcal{O}(N \log N)$ and $\mathcal{O}(N)$ algorithms, respectively, the constants are quite large). On the other hand, if N is large, the Fast Multipole Method will win.

However, although auto-tuning has become a widely accepted technique, at least in the research community, and has been applied successfully in a number of scientific computing libraries including the pioneers of auto-tuned libraries for scientific computing: ATLAS [172] for dense linear algebra, FFTW [65] and SPIRAL [139] for spectral algorithms, and OSKI [166] for sparse linear algebra, the optimization scope of today’s auto-tuners encompasses only code transformations and maybe data structure transformations. As these projects demonstrate, auto-tuning is still a valuable approach for creating high performance codes. Rather than hand-tuning and specifically tailoring code to a specific hardware architecture, which is done, e.g., in vendor libraries such as Intel’s Math Kernel Library (MKL), auto-tuning is an attempt at transferring performance across hardware platforms and letting the code adapt to the architecture’s peculiarities. Of course, this comes at a cost. While Intel’s MKL can just be copied to a new machine and is ready to be used, the installation of ATLAS requires going through a one-time procedure of adapting the kernels in an offline time intensive auto-tuning phase. But the benefit is that a single code base can be reused across many different architectures without needing to re-engineer and re-optimize compute kernels for every new emerging architecture in an error prone and time intensive process.

The same holds true for the PATUS approach: migrating to another platform requires a one-time offline re-tuning and possibly a preceding code generation phase, but none of the existing code has to be changed.

The three major concepts of auto-tuning are the optimization scope, code generation for the specific optimizations enumerated, and exploration of the code variant and parameter space [176]. In our approach we aim at an integration of all three concepts. The optimization scope is defined by the Strategy set and ultimately the Strategy design, code generation is handled by the PATUS code generator which depends on both the problem description in the form of the stencil specification and the on a Strategy, and Strategies also form the bridge to the PATUS auto-tuning module, which is dedicated to the search space exploration. Other approaches prefer a more composable structure, such as Active Harmony [159], which essentially provides a means to navigate the search space, while code generation is delegated to a decoupled transformation framework (currently, the loop transformation framework CHiLL [80] is used). Both FFTW and SPIRAL are appealing in that the search space essentially is constituted by code variants emerging from transformations within a rigorous mathematical framework, which enables enumerating valid algorithms.

The difficult task of an optimizing compiler is to decide on the potentially best combination of transformations without knowledge of the problem domain and completely obliviously of the data. The output is a single executable, based on a heuristically chosen path in the decision tree. In contrast, the auto-tuning approach encourages a choice of code variants for the tuning phase. Auto-tuners mostly are domain specific: In the code generation phase, variants can be generated which an optimizing compiler has to reject because it has to err on the conservative side for safety's sake. Also, the resultant optimization configuration returned by an auto-tuner is necessarily data-aware since an input data set (which ideally is real data or a training data set reflecting the properties of the real data) is required to run the benchmarks. Thus, optimizing compilers aim at best performance while being agnostic of the problem, the execution platform (to some extent), and the data, while auto-tuners aim at best performance for a specific problem domain, a specific hardware platform, and a specific input data set. By performance we do not necessarily mean maximum number of Flops per time unit, but a more general term, which can indeed include GFlop/s, but can also mean energy efficiency or maximum GFlop/s subject to constraints such as memory usage or total power consumption.

On the other hand, it can be too restrictive to optimize for a single input data set since the performance behavior can depend sensitively on the input data. Approaches to mitigate this include online auto-tuning, i.e., adapting the code while the program is running. This is an approach taken by Tiwari et al. in [159]. Another way would be to tune for multiple probable input data sets simultaneously, and choose the best code variant based on a probability distribution of the input data sets.

Another question is how to deal with and navigate the typically enormous search space, which is exponential in the number of parameters, and therefore how to deal with the time required to execute the benchmarks while traversing the search space. A purely theoretical approach is to find a performance model predicting the performance for each parameter configuration. This would not solve the problem of navigating the search space, but it could give insights which areas of the search space are "bad" and can be avoided. It might be the case that model-based pruning of the search space is sufficient so that the remaining configurations are so low in number so they can be examined exhaustively. In the context of our work, it is not clear whether and how this approach is applicable. It would require that a performance model is created (preferably automatically) for each PATUS Strategy.

Navigating the search space is a combinatorial optimization [22, 130] or integer programming [183] problem. Lacking a performance model, the cost function is not given as a mathematical model. Instead, evaluating the cost function for a certain input means executing the benchmarking executable under a specific parameter configuration and measuring the time spent in the kernel. Clearly, gradient-based optimization techniques are not applicable. In the next chapter, a choice of gradient-free search methods are discussed that could be used to traverse the search space. Gradient-free search methods include direct search methods, which were popularized in the 1960s and 1970s as well as metaheuristics such as genetic algorithms, which have enjoyed some popularity since the 1990s and can be counted to the state-of-the-art algorithms in integer programming today.

Finally, an essential requirement for the actual auto-tuning benchmarking process is a "clean" environment. When running the benchmarks, it is assumed that no other (significantly compute intensive) process is running besides the application being tuned so that measurements are not perturbed. Also, it is assumed that background noise (e.g., produced by the operating system) is negligible. Typically multiple runs are done and the performance is averaged or the median is taken.

8.2 Search Methods

Direct search methods have been characterized as optimization methods that are derivative-free methods (and therefore sometimes are referred to as *zero-order* methods), which only require the range of the objective function to be equipped with a partial order [98]. Some of the algorithms developed in the 1960's and 1970's, the golden era of direct search methods, are still in use in practice today since they have proven to be robust in that they are able to deliver at least a local optimum and they are simple and therefore easy to implement. The probably most famous of these algorithms is the Nelder-Mead method [118], also called simplex search (but not to be confused with the simplex algorithm for linear programming). In [98], Lewis et al. partition classical direct search methods into three categories: pattern search methods, simplex methods, and adaptive search direction methods. Originally, direct search methods were designed for continuously differentiable objective functions $f : \Omega \subseteq \mathbb{R}^n \longrightarrow \mathbb{R}$, and in that case convergence proofs can be given. Because they do not require the calculation of derivatives they can be applied to our discrete setting and actually work well enough for our purpose.

All the algorithms in pseudo code presented in this section try to find a local *minimum* of the objective function f, but they can be re-written easily to find a local maximum instead. In the text we use the generic term, *optimum*.

8.2.1 Exhaustive Search

Exhaustive search, i.e., enumeration of the entire search space, is the most basic approach as a search method and is guaranteed to find the global optimum. Obviously, due to tight time constraints this method has only limited use in practice since the search space grows exponentially with the number of parameters. However, it might be valuable to search a very limited number of parameters exhaustively, for which only a small range of values needs to be covered. An example within PATUS could be loop unrolling configurations. On the other hand, if heuristics can be found which can be used to prune the search space so that a manageable set of combination is left, these combinations can be searched exhaustively. Obviously, exhaustive search is trivially parallelizable. Thus, if a cluster of n equal machines is available, the time required for the exhaustive search can be cut by a factor of n.

8.2.2 A Greedy Heuristic

A simple greedy search method, which is only additive in the number of parameter values instead of multiplicative as the exhaustive search, varies one parameter at a time while the others remain fixed. Specifically, starting from an initial point, all the points parallel to the first axis are enumerated (potentially in parallel). From these points, the best one is picked and fixed, and from this new point, the method proceeds to enumerating all the points parallel to the second axis, and so forth. In Algorithm 8.1 we assume that the entire search space is given by the integers in $[x_1^{\min}, x_1^{\max}] \times \cdots \times [x_d^{\min}, x_d^{\max}]$.

Algorithm 8.1: *A greedy search method.*

```
Input: Objective function $f$, starting point $\mathbf{x}_0$
Output: Local minimizer $\hat{\mathbf{x}}$ for $f$
$\hat{\mathbf{x}} \leftarrow \mathbf{x}_0$
for $i = 1, \ldots, d$ do
    for all $\zeta = x_i^{\min}, \ldots, x_i^{\max}$ do
        $\boldsymbol{\xi} \leftarrow (x_1, \ldots, x_{i-1}, \zeta, x_{i+1}, \ldots, x_d)$    ▷ write: $\hat{\mathbf{x}} =: (x_1, \ldots, x_d)$
        if $f(\boldsymbol{\xi}) < f(\hat{\mathbf{x}})$ then    ▷ if $\boldsymbol{\xi}$ is better than $\hat{\mathbf{x}}$
            $\hat{\mathbf{x}} \leftarrow \boldsymbol{\xi}$ [atomically]    ▷ accept $\boldsymbol{\xi}$ as new point
        end if
    end for
end for
```

8.2.3 General Combined Elimination

This greedy algorithm was proposed by Pan and Eigenmann in [129]. The basic idea is to fix the parameters that have the most beneficial effect on performance one by one and iterate as long as parameters can be found which have not been fixed and improve the performance.

As for the greedy heuristic above, we assume that the d-dimensional search space is given by the integers in $[x_1^{\min}, x_1^{\max}] \times \cdots \times [x_d^{\min}, x_d^{\max}]$. In Algorithm 8.2 we write $\hat{\mathbf{x}} = (x_1, \ldots, x_d)$, and we use the notation

$$\hat{\mathbf{x}}|_{x_i=\xi} := (x_1, \ldots, x_{i-1}, \xi, x_{i+1}, \ldots, x_d)$$

to replace the i^{th} coordinate by ξ.

Algorithm 8.2: *General combined elimination.*

```
Input: Objective function $f$, starting point $\mathbf{x}_0$
Output: Local minimizer $\hat{\mathbf{x}}$ for $f$
 1: $\hat{\mathbf{x}} \leftarrow \mathbf{x}_0$
 2: $I = \{1, \dots, d\}$
 3: repeat
 4:     $\rho_{i\xi} = \frac{f(\hat{\mathbf{x}}|_{x_i=\xi}) - f(\hat{\mathbf{x}})}{f(\hat{\mathbf{x}})}$    for all $i \in I$,   $\xi = x_i^{\mathrm{min}}, \dots, x_i^{\mathrm{max}}$
 5:     $X \leftarrow \left\{ (i, \xi) : \rho_{i\xi} < 0 \text{ and } \xi = \arg\min_{\zeta} \rho_{i\zeta} \right\}$
 6:     for all $(i, \xi) \in X$ ordered ascendingly by $\rho_{i\xi}$ do
 7:         if $f(\hat{\mathbf{x}}|_{x_i=\xi}) < f(\hat{\mathbf{x}})$ then
 8:             $I \leftarrow I \backslash \{i\}$    ▷ remove processed index from index set
 9:             $\hat{\mathbf{x}} \leftarrow \hat{\mathbf{x}}|_{x_i=\xi}$    ▷ replace the old configuration
10:         end if
11:     end for
12: until $X = \{\}$
```

Algorithm 8.2 iteratively reduces the parameter index set I. In line 4, the *relative improvement percentage* ρ is calculated for each parameter still referenced by the index set I and for each admissible value in $[x_\bullet^{\mathrm{min}}, x_\bullet^{\mathrm{max}}]$ (in an embarrassingly parallel fashion). ρ is the relative improvement over the currently best parameter configuration $\hat{\mathbf{x}}$. In line 5, the set X of index-value pairs is formed, consisting of parameter configurations which improve the objective function over the old configuration $\hat{\mathbf{x}}$, i.e., $\rho_\bullet < 0$, and for each index only the value which results in the most improvement of the objective function. Then, in the `for` loop (lines 6 to 11), iteratively all the parameters are fixed to their best values ξ replacing the old configuration $\hat{\mathbf{x}}$, and the parameter reference is removed from the index set, provided that the objective function is still improved over the new configuration $\hat{\mathbf{x}}$. The process is repeated as long as there are parameters which improve the objective function, i.e., as long as X is not empty.

8.2.4 The Hooke-Jeeves Algorithm

The idea of the Hooke-Jeeves algorithm [82, 137], shown in Algorithm 8.3 is to make exploratory moves from the current best point by taking discrete steps in each positive and negative directions parallel to the coordinate axes. If an improvement could be found, the new point is accepted and the process is repeated. If no improvement was found, the step size is halved, and the process is repeated. If no improvement can be found and the step size has reached the tolerance, the algorithm stops and the current best point is returned as local optimum.

Hence this algorithm does a local search trying to improve a starting point by searching in its neighborhood. The algorithm also qualifies as a pattern search: Pattern search methods explore the search space by visiting a pattern of points in the search space, which lie on a rational lattice. For integer search, the algorithm terminates as soon as the step size becomes < 1.

Algorithm 8.3: *The Hooke-Jeeves algorithm.*

```
Input: Objective function $f$, starting point $\mathbf{x}_0$, initial step width $h$
Output: Local minimizer $\hat{\mathbf{x}}$ for $f$
$\mathbf{x} \leftarrow \mathbf{x}_0$
while not converged do
    repeat
        $\mathbf{x}_c \leftarrow \mathbf{x}$
        for $i = 1, \dots, d$ do                    ▷ Try to find improvements at
            if $f(\mathbf{x}_c + h\mathbf{e}_i) < f(\mathbf{x})$ then        ▷ $\mathbf{x} + h\sum_{i=1}^{d}(\pm\mathbf{e}_i)$
                $\mathbf{x}_c \leftarrow \mathbf{x}_c + h\mathbf{e}_i$              ▷ Accept improvement
            else
                if $f(\mathbf{x}_c - h\mathbf{e}_i) < f(\mathbf{x})$ then
                    $\mathbf{x}_c \leftarrow \mathbf{x}_c - h\mathbf{e}_i$          ▷ Accept improvement
                end if
            end if
        end for
        if $\mathbf{x} \neq \mathbf{x}_c$ then                      ▷ Value of $f$ was improved
            $\mathbf{d} \leftarrow \mathbf{x}_c - \mathbf{x}$
            $\mathbf{x} \leftarrow \mathbf{x}_c + \mathbf{d}$
        end if
    until $\mathbf{x} = \mathbf{x}_c$                          ▷ No improvement possible
    $h \leftarrow h/2$        ▷ Tighten mesh if no improvement can be found
end while
$\hat{\mathbf{x}} \leftarrow \mathbf{x}$
```

8.2.5 Powell's Method

Powell's Method [136] is an extension of the greedy search method described above. It starts in the same way as the greedy method, searching a local optimum along the coordinate axes. However, in contrast to the greedy method, in Powell's method after completing the run through the axes, a new search direction, $\boldsymbol{\xi}_d$ in Algorithm 8.4 is constructed, which is based on the local optimum found during the iteration and is linearly independent of the other search directions $\boldsymbol{\xi}_1, \ldots, \boldsymbol{\xi}_{d-1}$. The process is repeated with the new set of search directions until no improvement can be made.

In [136], Powell proves that for a convex quadratic the set of search directions always forms a set of conjugate directions (i.e., assuming that the objective function can be written as $f(\mathbf{x}) = \mathbf{x}^\top \mathbf{Q}\mathbf{x}$ for a symmetric positive definite matrix $\mathbf{Q} \in \mathbb{R}^{d \times d}$, $\boldsymbol{\xi}_j \mathbf{Q} \boldsymbol{\xi}_k = 0$ holds for any distinct search directions $\boldsymbol{\xi}_j, \boldsymbol{\xi}_k$), and that the method finds the optimum of the quadratic in finitely many steps. Note that, as in the greedy search, line 4 in Algorithm 8.4 can be done by parallel function evaluations.

Algorithm 8.4: *Powell's method.*

```
Input: Objective function $f$, starting point $\mathbf{x}_0$
Output: Local minimizer $\hat{\mathbf{x}}$ for $f$
 1: $\boldsymbol{\xi}_i \leftarrow \mathbf{e}_i \quad (i = 1, \ldots, d)$    ▷ Initialize search directions parallel to axes
 2: while $f(\mathbf{x}_0) < f\left(\mathbf{x}_0^{\text{old}}\right)$ do
 3:     for $i = 1, \ldots, d$ do
 4:         Find $\lambda_i$ such that $f(\mathbf{x}_{i-1} + \lambda_i \boldsymbol{\xi}_i)$ is a minimum
 5:         $\mathbf{x}_i \leftarrow \mathbf{x}_{i-1} + \lambda_i \boldsymbol{\xi}_i$
 6:     end for
 7:     $\boldsymbol{\xi}_i \leftarrow \boldsymbol{\xi}_{i+1} \quad (i = 1, \ldots, d-1)$    ▷ Construct new search directions
 8:     $\boldsymbol{\xi}_d \leftarrow \mathbf{x}_d - \mathbf{x}_0$
 9:     Find $\lambda_d$ such that $f(\mathbf{x}_d + \lambda_d \boldsymbol{\xi}_d)$ is a minimum
10:     $\mathbf{x}_0^{\text{old}} \leftarrow \mathbf{x}_0$
11:     $\mathbf{x}_0 \leftarrow \mathbf{x}_0 + \lambda_d \boldsymbol{\xi}_{i+1}$
12: end while
13: $\hat{\mathbf{x}} \leftarrow \mathbf{x}_0$
```

8.2.6 The Nelder-Mead Method

The Nelder-Mead method [118], also called simplex search or downhill simplex method, is probably the best known direct search method. One variant of the simplex search is given in Algorithm 8.5. The search works like so: After choosing $d+1$ points spanning a non-degenerate simplex in the d-dimensional search space and sorting them according to the function values such that $\mathbf{x}_1$ has the "best" value for the objective function relative to the other points, and $\mathbf{x}_{d+1}$ the "worst", four types of transformations are applied to the simplex by *reflecting*, *expanding*, or *contracting* the point with the worst function value at the centroid of the simplex. If one of these new transformed points happens to be superior to the worst point, the worst point is replaced by a transformed point (lines 5–18, and the process is repeated. If no better point is found, the simplex is *shrunk* around the best point (line 20). Nelder and Mead give a set of rules when the worst point is replaced by which transformed point. This is encoded in conditionals in lines 6–18 in Algorithm 8.5. Standard values for the parameters occurring in the algorithm (which were also used in PATUS's Nelder-Mead implementation, are $\alpha = 1.$, $\gamma = 2$, $\rho = \frac{1}{2}$, and $\sigma = \frac{1}{2}$.

The search method used in the ActiveHarmony auto-tuning framework [159], for instance, is based on the Nelder-Mead method, extended by a parallel evaluation of the objective function on the vertices after the transformation of the simplex (multiple transformations are done simultaneously).

Algorithm 8.5: *One variant of the Nelder-Mead method.*

```
Input: Objective function $f$
Output: Local minimizer $\hat{\mathbf{x}}$ for $f$
1: Choose $d+1$ points $\mathbf{x}_1, \dots, \mathbf{x}_{d+1}$ in the search space spanning a non-degenerate $d$-dimensional simplex
2: Sort the points such that $f(\mathbf{x}_1) \leqslant \cdots \leqslant f(\mathbf{x}_{d+1})$
3: while not converged do
4:     $\bar{\mathbf{x}} \leftarrow \frac{1}{d} \sum_{i=1}^{d} \mathbf{x}_i$        ▷ compute the centroid of all points except worst
5:     $\mathbf{x}^r \leftarrow \bar{\mathbf{x}} + \alpha (\bar{\mathbf{x}} - \mathbf{x}_{d+1})$        ▷ compute reflected point
6:     if $f(\mathbf{x}_1) \leqslant f(\mathbf{x}^r) < f(\mathbf{x}_d)$ then
7:         $\mathbf{x}_{d+1} \leftarrow \mathbf{x}^r$        ▷ accept reflected point
8:     else if $f(\mathbf{x}^r) < f(\mathbf{x}_1)$ then
9:         $\mathbf{x}^e \leftarrow \bar{\mathbf{x}} + \gamma (\bar{\mathbf{x}} - \mathbf{x}_{d+1})$        ▷ compute expansion
```

Algorithm 8.5: *One variant of the Nelder-Mead method.* (cont.)

```
        if $f(\mathbf{x}^e) < f(\mathbf{x}^r)$ then
            $\mathbf{x}_{d+1} \leftarrow \mathbf{x}^e$                          ▷ accept expansion
        else
            $\mathbf{x}_{d+1} \leftarrow \mathbf{x}^r$                          ▷ accept reflected point
        end if
    else
        $\mathbf{x}^c \leftarrow \mathbf{x}_{d+1} + \rho(\bar{\mathbf{x}} - \mathbf{x}_{d+1})$           ▷ compute contraction
        if $f(\mathbf{x}^c) < f(\mathbf{x}_{d+1})$ then
            $\mathbf{x}_{d+1} \leftarrow \mathbf{x}^r$                          ▷ accept reflection
        else
            $\mathbf{x}_i \leftarrow \mathbf{x}_1 + \sigma(\mathbf{x}_i - \mathbf{x}_1),\ i = 2, \ldots, d+1$  ▷ reduce simplex
        end if
    end if
end while
$\hat{\mathbf{x}} \leftarrow \mathbf{x}_1$
```

8.2.7 The DIRECT Method

The DIRECT method [68, 90] is another pattern search, which, in contrast to the Hooke-Jeeves algorithm, was devised for global optimization. The basic idea is simple: The search space is covered with a rectilinear, initially coarse lattice, and the objective function is evaluated at the centers of the lattice cells. The cells with the best function values are subdivided recursively (each cell is cut into 3 sub-cells along one axis), and the objective function is evaluated at the cell centers of the sub-cells. This subdivide-and-evaluate process is repeated until a potential global optimum is found.

For the implementation in PATUS, the original idea has been modified slightly to match the problem structure better. The objective function evaluations are done at the cell corners instead of the cell centers, and cells are recursively bi-sected instead of tri-sected.

8.2.8 Genetic Algorithms

An approach to combinatorial optimization radically different from the direct search methods presented above are nature-inspired evolutionary algorithms [22]. The main conceptual difference between the direct search methods and evolutionary algorithms is that the former are deterministic methods, whereas the basic principle for the latter is randomization.

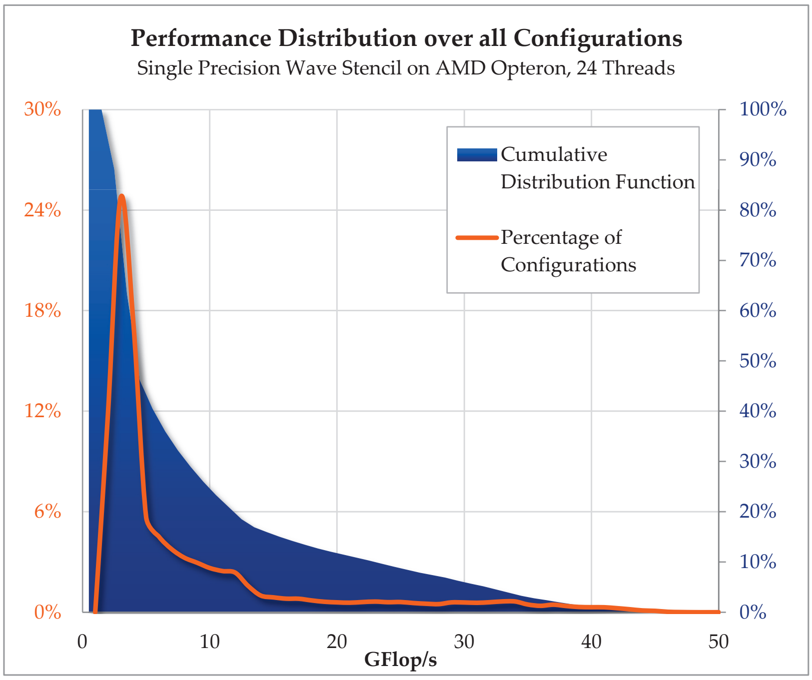
Genetic algorithms, ant colony optimization, and particle swarm optimization are popular types of evolutionary algorithms. A genetic algorithm is included in PATUS, which is based on the JGAP [109] package. The idea behind genetic algorithms is natural selection. Roughly speaking, a population — a set of potential solutions — is evolved iteratively towards a "better" population, a set of solutions more optimal than the one before the evolution. Trivially, as the individuals in the population are independent, the population can be evolved in an embarrassingly parallel fashion. The actions driving an evolution are *crossover*, i.e., interbreeding two potential solutions, and *mutation*, i.e., modifying one or multiple *genes* of a potential solution, and *selection*, i.e., selecting the individuals of the population, which survive the evolution step. The latter is done by means of the *fitness function* — the objective function in evolutionary algorithm lingo, — which measures the quality of a solution.

In the genetic algorithm used currently in PATUS, the crossover operator mixes some of the genes of two individuals, i.e., swaps the values of certain parameters. Mutation is applied on a randomly selected, small set of individuals in the population: mutating means altering the value of a parameter by a random percentage between -100% and $+100\%$.

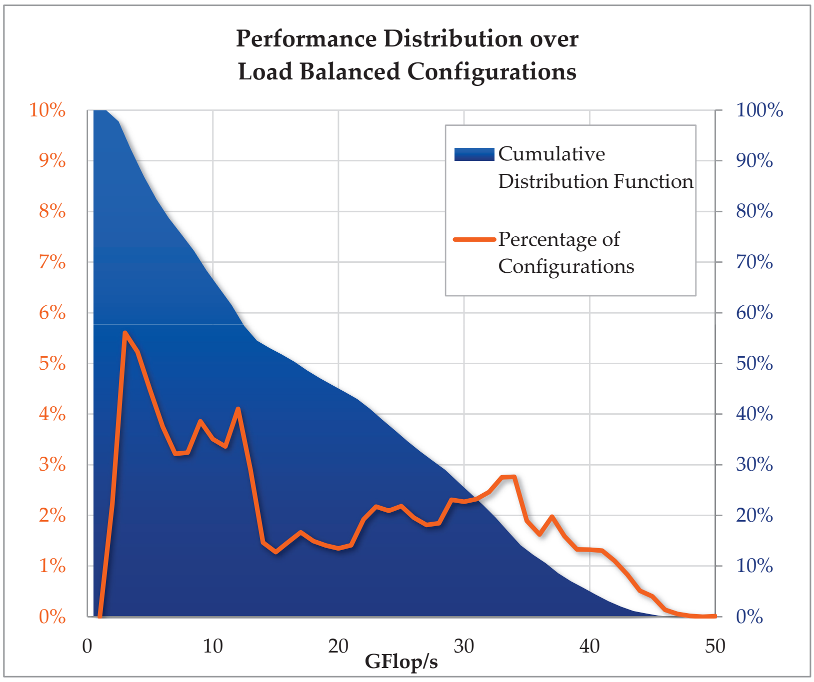
8.3 Experiments with the Patus Auto-Tuner — Search Method Evaluation

Fig. 8.1 summarizes the need for auto-tuning and the need for a good search method. It shows the performance distribution of the search space for a specific stencil — the *Wave* stencil from the example in Chapter 5.2 for a 200^3 problem size — and for a specific Strategy. The Strategy chosen is the chunked cache blocking strategy presented in Listing 7.2, with loop unrolling enabled. The Strategy has 4 parameters for 3D stencils, the first three ranging from 2 to 200 in increments of 2, the fourth accepting values in $\{1, 2, 4, 8, 16, 32\}$. There were 8 loop unrolling configurations. Thus, the total search space contained $100^3 \cdot 6 \cdot 8 = 4.8 \cdot 10^7$ points.

The data for Fig. 8.1 was obtained by conducting a partial exhaustive search, fixing the first parameter to 200 (a reasonable choice, since the first parameter corresponds to the block size in unit stride direction, and



(a)



(b)

Figure 8.1: *Performance distribution over the full search space (a) and restricted to load balanced configurations (b).*

by deciding not to cut along the unit stride dimension, the use of the hardware prefetcher is maximized, and also data is loaded in as large contiguous chunks as possible), and letting parameters 2 and 3 accept values between 4 and 200 in increments of 4. Thus, the exhaustive search could be constrained to a tractable number of $120,000$ benchmark runs, for which 150 one-hour jobs were allocated on a cluster.

Fig. 8.1 (a) shows the distribution over the this entire search space; Fig. 8.1 (b) restricts the benchmark runs to load balanced ones, i.e., runs in which all of the 24 threads of the AMD Opteron Mangy Cours (cf. Chapter 9), the hardware platform on which the experiment was conducted, participated in the computation.*

The orange lines show how many configurations, in percent, caused the benchmark executable to run at the performance given on the horizontal axis, i.e., it visualizes the probability distribution of the random variable mapping parameter configurations to the corresponding performance. The number percentages are labeled to the left of the figures. In the total search space, in sub-figure (a), almost one fourth of the configuration ran at around 5 GFlop/s, whereas the maximum attainable performance for this stencil and this strategy is around 50 GFlop/s. Restricting the block sizes to load balanced configurations effectuates a more uniform distribution of the configurations across the performance range, although the spike at 5 GFlop/s remains, if less pronounced. The blue areas are cumulative distribution functions with percentages labeled to the right of the figures: Given a specific performance number, the upper boundary of the blue area gives the percentage of configurations which achieved *at least* that performance. Thus, e.g., the probability to pick a load balanced configuration which achieves at least 40 GFlop/s is around 5% for this particular stencil and Strategy on this particular hardware platform.

For the evaluation of the search methods for auto-tuning, two stencil kernels were picked for which tuning had different benefits: the single precision *Wave* stencil and a double precision 6^{th} order Laplacian, called *Upstream* in the following charts. On the AMD Opteron Magny Cours, on which the auto-tuning was performed, the *Wave* stencil benefited mostly from explicit vectorization, and the *Upstream* stencil, in contrast, from blocking (cf. Chapter 10). Vectorization obviously is not a tunable optimization (it can be either turned on or off), whereas blocking is. Thus, for

---

*In the Strategy, the size of the blocks determines how many threads can participate; if the block sizes are too large, some threads remain idle.

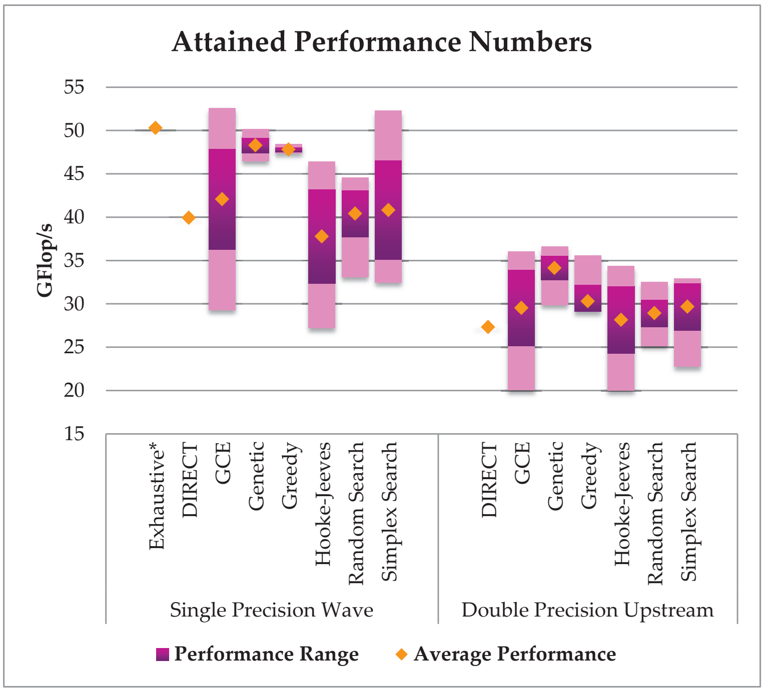
the *Wave* stencil we want to verify that the performance does not deteriorate with respect to the base line when using the auto-tuner with a specific search method, and for *Upstream*, for which blocking about doubles the performance, we want to test whether the various search methods are able to find blocking configurations which improve the performance.

For the search method evaluation, we added a method that was not discussed in the previous section: random search. This method just takes randomly a number (here: 500) of samples from the search space and evaluates them. For the *Wave* stencil, we also included the partial exhaustive search described above, labeled "Exhaustive*" in the charts.

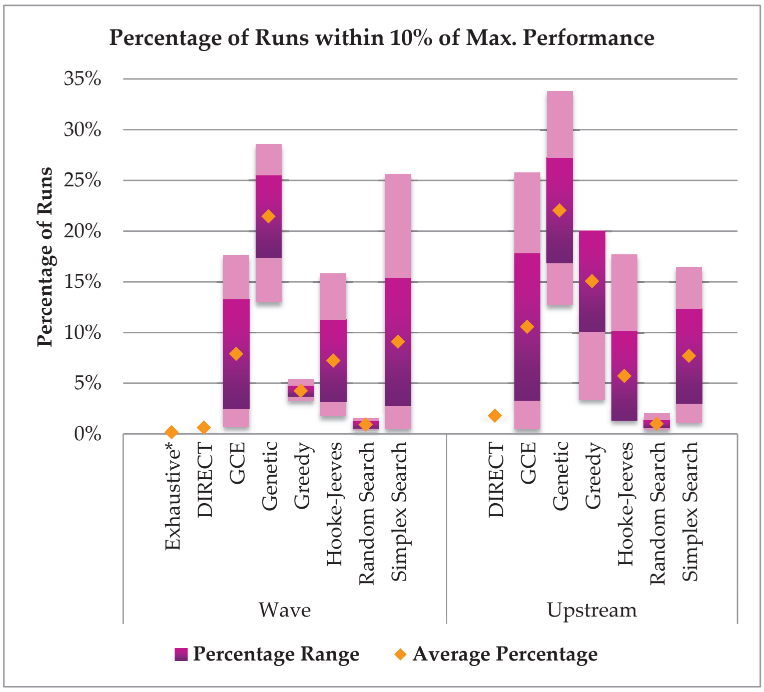
We did 25 auto-tuning runs with each search method. We only did one run using the DIRECT algorithm since it is deterministic. All other algorithms have some stochastic elements, at least for picking a starting point. The performance numbers shown for the DIRECT algorithm might be suboptimal, since it exceeded the 3 hour time budget, causing an early termination of the algorithm.

The purple and blue bars in Figs. 8.2 and 8.3 visualize ranges; the lower end of a bar is the minimum and the upper end the maximum value. The more strongly colored middle part of the bar shows the standard deviation around the mean, which is marked by a yellow or green diamond.

Fig. 8.2 (a) shows the performance ranges, the standard deviations, and the average performance numbers achieved by the search methods for both the *Wave* (left) and the *Upstream* stencils (right). In both cases, the genetic algorithm (except from the exhaustive search), reached the highest average performance. In the *Wave* example, general combined elimination (GCE) and simplex search reached higher maximum absolute performances, but both methods have a large variance in the performance outcome; so does Hooke-Jeeves. Judging by the standard deviation, the genetic algorithm was still able to outperform both GCE and simplex search. The large variance implies that the methods are sensitive to the choice of the starting point. Apparently, a poor choice for a starting point cannot improve performance very far; the result being that the outcome can even be worse than when applying random search. Conversely, if a good starting point is given, both GCE and simplex search yield very good results. Greedy search and the genetic algorithm, both having a more "global view," have a significantly smaller variance. Greedy search, however, is likely to get stuck in a local optimum; thus the genetic algorithm overall delivers higher average and maximum performances —



(a)



(b)

Figure 8.2: *Ranges, standard deviations, and averages of (a) attained performance and (b) percentage of runs with at least 90% of the maximum performance of the respective search method.*

which, however, can be less than the best achieved with GCE and simplex search as shown in the *Wave* example. For the *Upstream* stencil, GCE, genetic, and greedy search are about on par; here simplex search performs slightly worse.

15%–30% of *all benchmark runs* in the genetic algorithm search have a performance, which is at least 90% of the maximum of that auto-tuning run. This is shown in Fig. 8.2 (b). The fact is quite surprising, but it means that the chosen mutation and crossover operators are well suited to the problem (at least for the chosen Strategy). Naturally by construction, exhaustive search and the DIRECT method, which evaluates many points spread over the entire search space, perform many benchmark runs in "bad regions," therefore only a very low percentage of the runs fall into the high performing category. Equally, random search, agnostic of the run history during the tuning process, does many low performing benchmark runs. For greedy search, the percentage of "good runs" obviously depends on the search space structure.

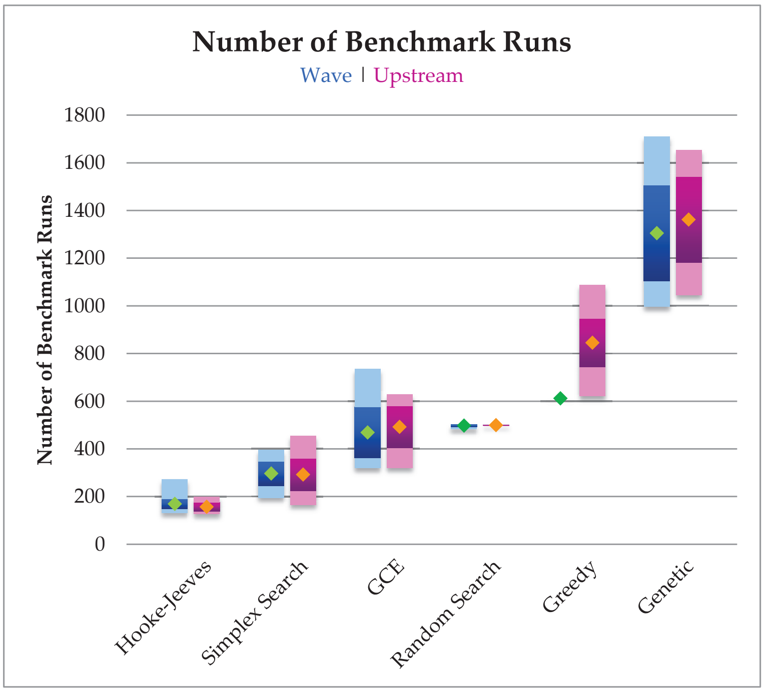
Fig. 8.3 shows the price that was paid for the good performance outcome in the genetic algorithm. The blue bars represent the results for the *Wave* stencil, the purple bars for the *Upstream* stencil. Where Hooke-Jeeves, simplex search, and GCE stayed under 500 benchmark runs on average, more than 1000 (up to 1700) runs were done when applying the genetic algorithm. As a local search method, Hooke-Jeeves terminates quickly[†]; each auto-tuning run took well under 10 minutes as shown in Fig. 8.3 (b), one benchmark run taking 2.3 seconds on average. With around 10 and around 20 minutes tuning time, simplex search[‡] and GCE, respectively, are the methods of choice if fast tuning is desired, keeping in mind that performance might deteriorate for poor choices for starting points. The running time of greedy search again depends on the structure of the search space; *Upstream*, with a higher potential of performance increase due to tuning, requires more adjusting than *Wave*.

Fig. 8.4 shows the convergence of the search methods under study. The lines show the maximum attained performance after $2, 4, 8, \dots, 1024$ benchmark runs — for *Wave* in (a) and *Upstream* in (b). Note the logarithmic scale of the horizontal axis. Also note that the performances shown in the figures are *averages* over the conducted experiments. The figures show that the genetic algorithm — on average — beats Hooke-

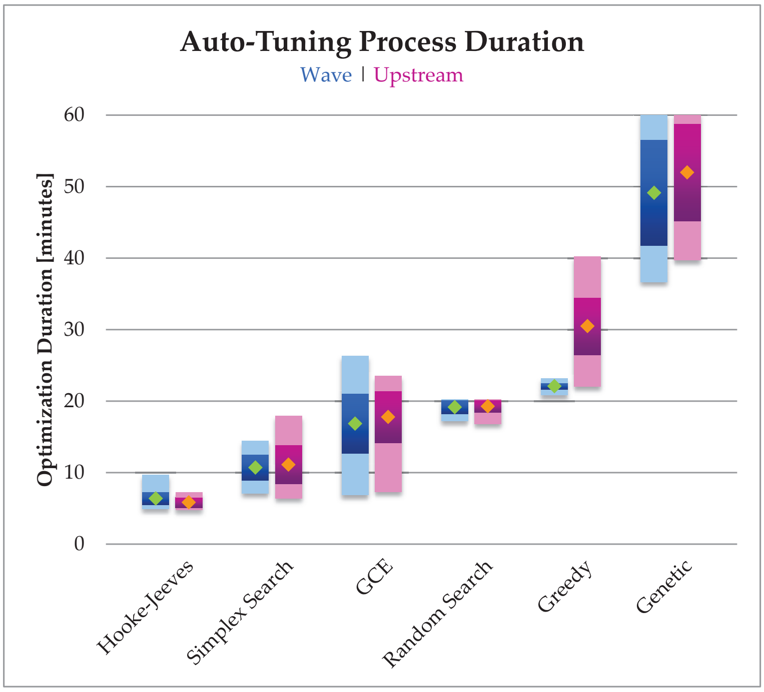
---

[†] In fact, 5 Hooke-Jeeves runs were carried out, each starting from another randomly chosen starting point.

[‡] Again, 5 runs were done starting from different randomly chosen simplices.

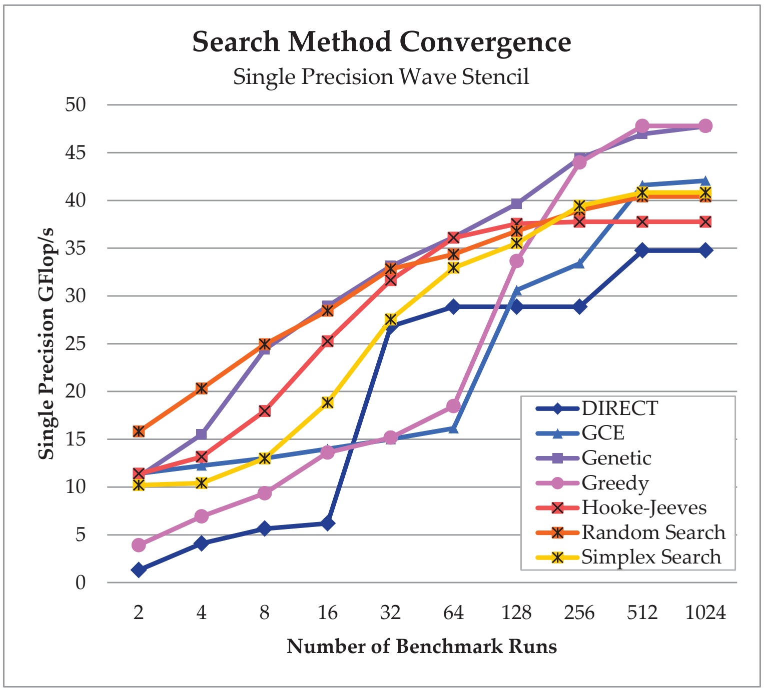


(a)

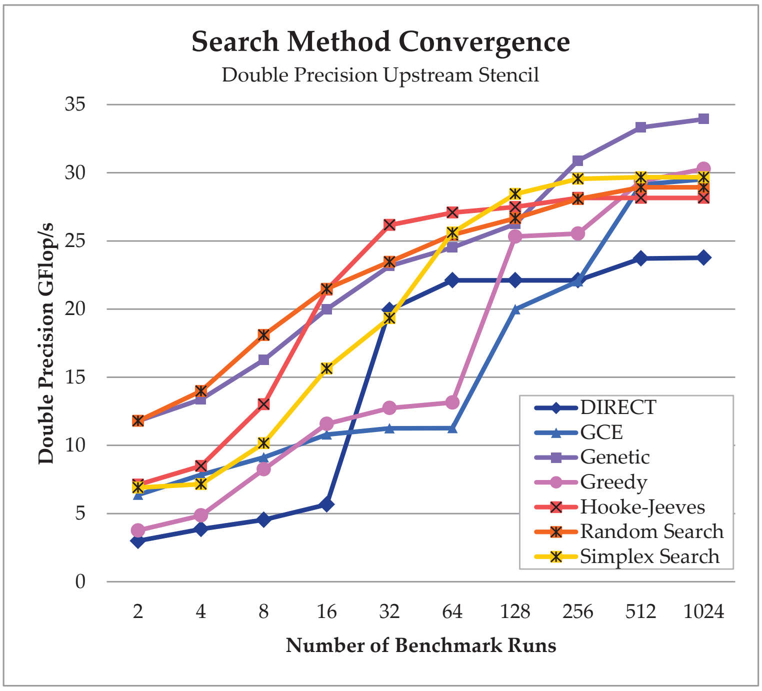


(b)

Figure 8.3: *Number of benchmark runs in the auto-tuning process (a) and total duration of the auto-tuning process (b).*



(a)



(b)

Figure 8.4: *Performance convergence of the search methods for (a) the* Wave *and (b) the* Upstream *stencils. The performance numbers are averages over all runs. Note the logarithmic scale of the horizontal axis.*

Jeeves, GCE, and simplex search, even if not as many as > 1000 iterations are carried out. As the figures show, in fact it can be stopped earlier, as the performance increase from 512 to 1024 runs is not as pronounced. On average, the performance during a GCE run seems to increase rather slowly, the convergence rates of Hooke-Jeeves and simplex search are about equal. Maybe surprisingly, random search shows a steady convergence and rather good performance numbers, but remember that the numbers are statistically expected values; thus, in a specific instance the performance outcome might not be as desired.

For the performance benchmark results in Chapter 10, we used the genetic algorithm as search method.

Part III

Applications & Results

Chapter 9

Experimental Testbeds

> It is desirable to guard against the possibility
> of exaggerated ideas that might arise as to
> the powers of the Analytical Engine.
>
> — Ada Lovelace (1815–1852)

In this chapter, we give an overview over the hardware architecture that were used to conduct the performance benchmark experiments which will be discussed in the next chapter. Table 9.1 shows a summary of the architectural features.

The bandwidth was measured using the STREAM triad benchmark [106], and the performance is for a single precision matrix-matrix multiplication of two large (8192×8192) square matrices. The compute balance measures how many floating point operations have to be carried out per data element D transferred from main memory to the compute units so that there is no pipeline stall. Table 9.1 shows that the numbers are in the range of 20–30 Flops per datum on these current architectures — a high number which is not reached for typical stencils. This once again highlights the importance of algorithms making efficient use of the cache hierarchy.

	AMD Opteron 6172 *Magny Cours*	**Intel** Xeon E7540 *Nehalem*	**NVIDIA** Tesla C2050 *Fermi*
Cores	$2 \times 2 \times 6$	2×6	14
Concurrency	24 HW threads	24 HW threads	448 ALUs
Clock	2.1 GHz	2 GHz	1.15 GHz
L1 Data Cache	64 KB	32 KB	$48 + 16$ KB
L2 Cache	512 KB	256 KB	—
Sh'd Last Level Cache	6 MB	18 MB	768 KB
Avg. Sh'd L3/HW Thd	1 MB	1.5 MB	—
Measured Bandwidth	53.1 GB/s	35.0 GB/s	84.4 GB/s
Measured SP Perf.	209 GFlop/s	155 GFlops/s	618 GFlop/s
Compute Balance	23.3 Flop/D	17.7 Flop/D	29.3 Flop/D

Table 9.1: *Hardware architecture characteristics summary.*

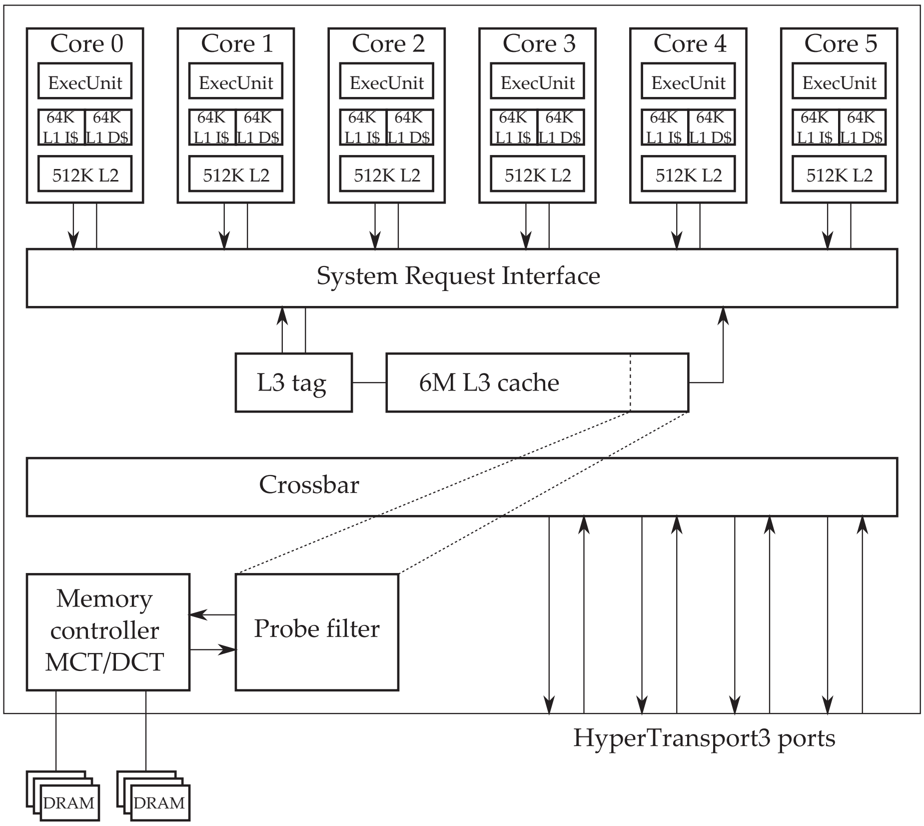


Figure 9.1: *Block diagram of one die of the* 12 *core AMD Opteron Magny Cours architecture.*

9.1 AMD Opteron Magny Cours

AMD's Magny Cours processor integrates two dies of six x86-64 cores, each manufactured in 45 nm silicon on insulator (SOI) process technology, in one package, which total around 2.3 billion transistors.

The cores are out-of-order, three-way superscalar processors, i.e., they can fetch and decode up to three x86-64 instructions per cycle to dispatch them to independent functional units. Specifically, variable length x86-64 instructions are converted to fixed-length macro-operations, which are dispatched to two independent schedulers for integer and to a floating point/multimedia operations, respectively. There are three integer pipelines for integer operations and address generation, and three pipelines for floating point and multimedia operations.

Each core has its own two-way associative 64 KB L1 caches (two, one for instructions and one for data) and its own on-chip 16-way associative unified 512 KB L2 cache, and all the cores on a die share a 16-way associative 6 MB L3 cache. The load-to-use latencies for the L1 cache are 3 cycles, and 12 cycles for the L2 cache. The L2 cache is a victim cache for the L1 caches, i.e., DRAM fetches are moved directly into L1, and cache lines evicted from the L1 cache are caught by the L2 cache. Similarly, the L3 cache is a victim cache for the L2 caches of the individual cores. A sophisticated algorithm prevents L3 cache thrashing based on the cores' cache efficiency.

AMD's solution to the cache coherence problem in multiprocessor settings in the Magny Cours architecture is *HT Assist*, also known as the *Probe Filter*: a cache directory that keeps track of all the cache lines that are in the caches of other processors in the system and therefore eliminates the need of broadcasting probes. If activated, the cache directory reserves 1 MB of the L3 cache for its use [46].

Each of the two dies has two DDR3 memory channels and four HyperTransport 3.0 links, cf. Fig. 9.1. This implies that the package itself is a NUMA architecture: each die is a NUMA node, each having its own memory controller. Two of the per-die HyperTransport links are used to connect the dies internally. Hence, the package exposes four memory channels and four HyperTransport links. The maximum theoretical memory bandwidth per socket is 25.6 GB/s. We measured an aggregate bandwidth of 53.1 GB/s using all 24 threads on all 4 dies on the dual-socket node using the STREAM triad benchmark [106].

The benchmarks were run on one node of a Cray XE6. On the ma-

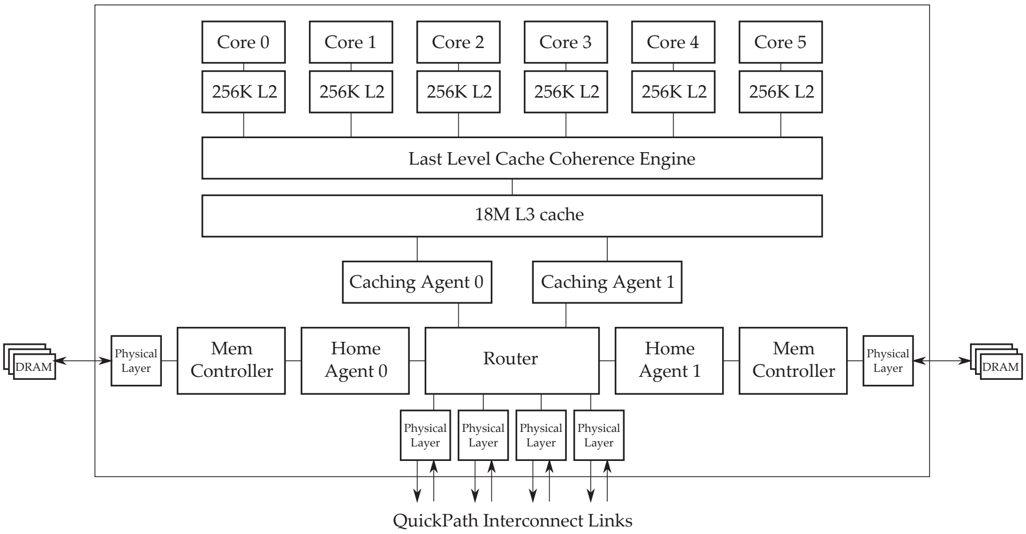


Figure 9.2: *Block diagram of one die of the* 6 *core Intel Xeon Nehalem architecture.*

chine, the Cray Linux Environment 3.1 was run as operating system. The GNU gcc 4.5.2 C compiler was used.

9.2 Intel Nehalem

The Intel Xeon E7540 Beckton belongs to the family of the recently released *Nehalem* architectures, which differs drastically from the previous generations in that the frontside bus, which used to connect processors via a north-bridge chip to memory, has been forgone in favor of Quick Path Interconnect (QPI) links, which is Intel's point-to-point coherence interface, and in favor of on-chip memory controllers. This feature effectuates a considerably higher bandwidth compared to older generation systems. Coherency is managed through distributed or directed snoop messages.

The chip is also manufactured in 45 nm technology and contains also around 2.3 billion transistors. The system we used is a dual-socket hexacore architecture running at 2 GHz clock frequency. We used the CPUs in Hyper Threading mode, thus there are 24 hardware threads available. The cores are 4-wide out-of-order cores. Each of the cores is equipped with a 32 KB L1 data and a 32 KB L1 instruction cache and a shared 256 KB L2 cache. The six cores on one socket share an 18 MB L3 cache. There are four Scalable Memory Interconnect channels. We measured 35 GB/s

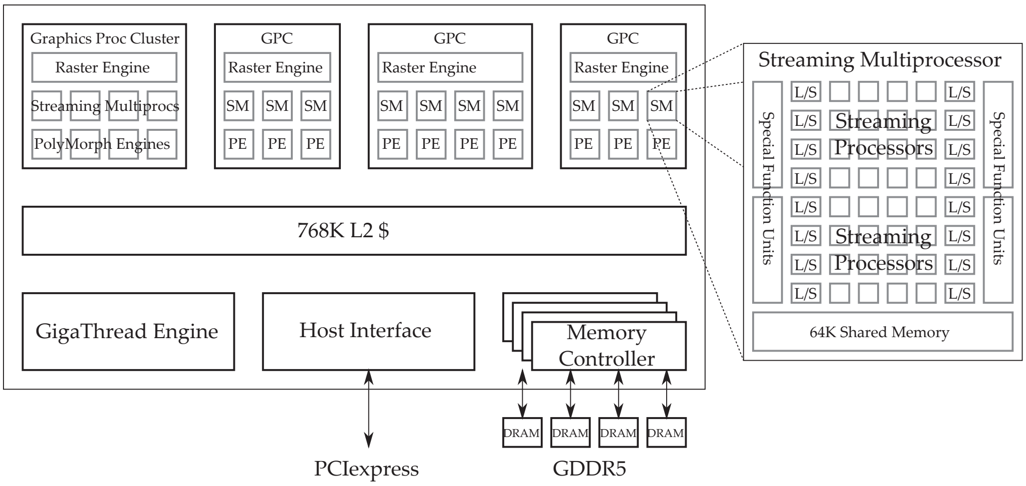


Figure 9.3: *Block diagram of one die of the NVIDIA Fermi GPU Tesla C2050 architecture.*

of sustained bandwidth from the memory to the sockets. The bandwidth measurement have been done using the STREAM Triad benchmark [106].

The Caching Agents connect the cores to the un-core interconnect and the L3 cache. The 8-port router manages the QPI layer and is directly connected to the Caching and Home Agents. The remaining four ports are connected to the external QPI ports. The Home Agents handle read and write requests to the memory controllers, as well as data write-backs from L3 cache replacement victims.

For the benchmarks, we used one node of an IBM x3850 M2 system running SUSE SLES 11.1. The benchmarks were compiled with Intel's C compiler, icc 11.1.

9.3 NVIDIA GPUs

CUDA, the *Compute Unified Device Architecture*, is NVIDIA's graphics processing unit (GPU) architecture introduced with the G80 series of their GeForce GPUs in the fall of 2006. The GeForce 8800 was the first CUDA-programmable GPU. In CUDA GPUs, the traditional graphics pipeline consisting of stages of special-purpose compute units was replaced by unified cores, which could carry out any of the graphics-specific operations previously reserved to dedicated geometry, vertex, and pixel shaders. Advantages of the unified architecture are better balanced workloads, with the added benefit that the by nature massively parallel device could be used much more easily for general purpose computations*. Around the same time, AMD-ATI released their version of unified architecture GPUs, initially with their *Stream Processor* based on the Radeon R580 GPU.

Since then, GPU computing has become increasingly popular — understandably, since GPUs, as inexpensive commodity devices are ubiquitous and deliver a high floating point performance. They have been so successful that they have been readily adapted in high performance computing; in fact, 3 of the top five supercomputers at the time of writing (TOP500 [161], June 2011) use NVIDIA GPUs as hardware accelerators.

The high-end version of NVIDIA's first CUDA GPU had 128 unified shaders — so-called streaming processors. The device used for the benchmarks in this thesis, a Tesla C2050, which is explicitly dedicated to computation, has 448. The C2050 is one of the first GPUs in NVIDIA's Fermi [124] architecture. It is fabricated in a 40 nm process and has around 3 billion transistors. The core clock is 1.15 GHz, which is lower than the clock rate of graphics-dedicated GPUs, in favor of increased reliability. The theoretical single precision peak performance therefore is around 1 TFlop/s, and the double precision peak performance is about 500 GFlop/s. In the Fermi architecture, the double precision performance has been greatly improved over the previous GPU generations.

For computation, the GPU is understood as a co-processor connected to the CPU *host* system over the PCIexpress bus. A block diagram of the internals of the GPU is shown in Fig. 9.3. The diagram reveals that the 448 light-weight cores are organized in 14 *streaming multiprocessors*, containing 32 cores each. From another viewpoint, the C2050 is a 14 core architecture with each core being a 32-way SIMD unit, since all the streaming processors within a scheduling unit carry out the same instruction. The streaming multiprocessors are again organized into *graphics processing clusters*, grouping 3 or 4 streaming multiprocessors and containing one raster engine.

Each streaming multiprocessor contains 64 KB of *shared memory* (here "shared" means that the memory is shared among all the streaming processors within a streaming multiprocessor), which can be configured as

---

*GPGPU, i.e., general purpose computing on GPUs, has been done prior to the launch of unified device architectures. However, graphics knowledge was required, and computation had to be translated to the graphics metaphor; e.g., running an algorithm would correspond to rendering an image, or to transfer data, the image had to be projected onto a surface. Also, non-IEEE compliant floating point arithmetic could lead to occasional surprises.

a 16 KB cache and 48 KB of explicit, software-controlled local memory or 48 KB of cache and 16 KB of local memory. Each streaming multiprocessor has a rather large register file: there are 32768 32-bit wide registers. Access latencies to the register file is one clock cycle, and a couple of clock cycles to the shared memory. The entire device comes with a modest 768 KB last level cache.

The GPU is equipped with 3 GB of on-board ECC-protected GDDR5 memory, for which we measured 84.4 GB/s of sustained bandwidth with ECC turned on using NVIDIA's bandwidth measurement utility. DRAM latency is in the range of several hundred clock cycles.

Parallel to the hardware, NVIDIA developed C for CUDA [123], the general-purpose language used to program their CUDA GPUs. CUDA C is a slight extension of C/C++. In particular, there are new specifiers identifying a *kernel*, the program portion which is executed on the GPU. The CUDA programming model follows the SPMD model; each logical thread executes the same kernel, and a kernel therefore is a thread-specific program, in which special built-in variables have to be used to identify a thread and the portion of the data the thread is supposed to operate on.

In correspondence to the hierarchical structure of the hardware, the threads — which are logically executed by streaming processors — are grouped into *thread blocks*, which are mapped onto streaming multiprocessors. Thread blocks, in turn, are grouped into the *grid*, which, in the execution model, corresponds to the entire GPU. In the Fermi architecture, both thread blocks and the grid can be one, two, or three-dimensional (i.e., indexed by one, two, or three-dimensional thread and thread block IDs), which eases index translations for three-dimensional physical simulations, for instance.

For the benchmarks, version 4.0 of the CUDA SDK and runtime was used, compiled with nvcc release 4.0, V0.2.1221 wrapping GNU gcc 4.4.3.

Chapter 10

Performance Benchmark Experiments

> In studying the action of the Analytical
> Engine, we find that the peculiar and
> independent nature of the considerations
> which in all mathematical analysis belong to
> operations, as distinguished from the objects
> operated upon and from the results of the
> operations performed upon those objects, is
> very strikingly defined and separated.
>
> — Ada Lovelace (1815–1852)

10.1 Performance Benchmarks

In this chapter, we conduct performance benchmark experiments for six selected stencil kernels. For presentation reasons they were divided into two sets: Set 1 contains the low-arithmetic intensity stencils of the basic low-order discretization differential operators *Gradient*, *Divergence*, and *Laplacian* as described in Chapter 5.1. Set 2 contains the higher-arithmetic intensity kernels *Wave*, *Upstream*, and *Tricubic*. *Wave* is the 3D 13-point stencil of a fourth-order discretization of the classical wave equation used in the example in Chapter 5.2. *Upstream* is the 3D 19-point stencil of a sixth-order discretization of the Laplacian shown in Chapter 5.1, and *Tricubic* is the tricubic interpolation, also shown in Chapter 5.1: a 3D 64-point stencil. Both *Upstream* and *Tricubic* stencils are inspired by a real-world application, the weather code COSMO [64].

All the stencils were auto-tuned and benchmarked both in single and double precision, as some striking optimization impact phenomena can be observed when switching the precision modes. As search method for the auto-tuner the genetic algorithm was used. The benchmarks were all done on a grid with 200^3 interior grid points. To obtain the performance numbers, the run time of 5 one-sweep runs was measured after one warm-up run.

Table 10.1 summarizes the arithmetic intensities and the performance bounds for both sets of stencils used in the benchmarks. In contrast to the asymptotic numbers given earlier in Table 5.3, the numbers in Table 10.1 account for the boundary data. Furthermore, for each stencil, two arithmetic intensities in Flops per data element are given: The one in the upper row is the ideal arithmetic intensity, counting the number of compulsory data transfers for the actual problem size. The arithmetic intensities printed in italics in the lower row, are the numbers corresponding to the actual transfer done by the hardware, including the write-allocate traffic: On a CPU system, before every write-back, the data is brought from DRAM into the cache before the actual write-back to DRAM occurs. This causes the write-back transfer volume to be doubled, resulting in a significantly lower arithmetic intensity for most of the stencils, and therefore in a lower effective performance. The *Tricubic* stencil is compute-bound on all of the three selected architectures.

10.1.1 AMD Opteron Magny Cours

On the AMD Opteron Magny Cours, we auto-tuned and benchmarked stencil codes from three cache blocking Strategy variations.

Strategy 1, shown in Listing 10.1, decomposes the total domain into smaller domains for parallelization, assigning one or multiple consecutive subdomains to one thread, and doing cache blocking within the thread's local domain. Both the thread block and the cache block sizes are parameters to be tuned. For a 3D stencil, this Strategy requires tuning of 7 parameters.

Stencil	**Arith. Int.** [Flop/data elt.]	**Max. SP Perf.** [GFlop/s] *AMD*	*Intel*	*NVIDIA*	**Max. DP Perf.** [GFlop/s] *AMD*	*Intel*	*NVIDIA*
Gradient	1.50	19.9	13.1	31.7	10.0	6.6	15.8
	0.86	*11.4*	*7.5*	—	*5.7*	*3.8*	—
Divergence	2.00	26.6	17.5	42.2	13.3	8.8	21.1
	1.60	*21.1*	*14.0*	—	*10.6*	*7.0*	—
Laplacian	3.94	52.3	34.5	83.1	26.2	17.2	41.6
	2.64	*35.0*	*23.1*	—	*17.5*	*11.6*	—
Wave	9.22	122.4	80.7	194.5	61.2	40.3	97.2
	6.21	*82.4*	*54.3*	—	*41.2*	*27.2*	—
Upstream	10.51	139.6	92.0	221.8	69.8	46.0	110.9
	7.11	*94.4*	*62.2*	—	*47.2*	*31.1*	—
Tricubic	63.02	(CB)	(CB)	(CB)	(CB)	(CB)	(CB)
	52.60			—			—

Table 10.1: *Arithmetic intensities of the stencils in Flops per data element neglecting (upper row) and accounting for (lower row in italics) write allocate transfers. The Performance bounds for single (SP and double precision (DP) stencils are inferred from the bandwidth measurements in Table 9.1. (CB) means* compute bound.

Listing 10.1: *Strategy* 1: *Thread block parallelization and cache blocking.*

```
strategy thdblk (domain u, auto dim tb, auto dim cb,
  auto int chunk)
{
  // iterate over time steps
  for t = 1 .. stencil.t_max {
    // iterate over subdomain, parallelization
    for subdomain v(tb) in u(:;t) parallel schedule chunk
    {
      // cache blocking
      for subdomain w(cb) in v(:; t)
        for point pt in w(:; t)
          w[pt; t+1] = stencil (w[pt; t]);
    }
  }
}
```

Strategy 2 collapses the parallelization and cache blocking subdivision of Strategy 1 into one level of decomposition. The size of the blocks is to be tuned. In 3D, this Strategy has 3 tuning parameters.

Listing 10.2: *Strategy 2: Simple cache blocking.*

```
strategy compact (domain u, auto dim cb) {
  // iterate over time steps
  for t = 1 .. stencil.t_max {
    // iterate over subdomain
    for subdomain v(cb) in u(:; t) parallel {
      // calculate the stencil for each point
      // in the subdomain
      for point p in v(:; t)
        v[p; t+1] = stencil (v[p; t]);
    }
  }
}
```

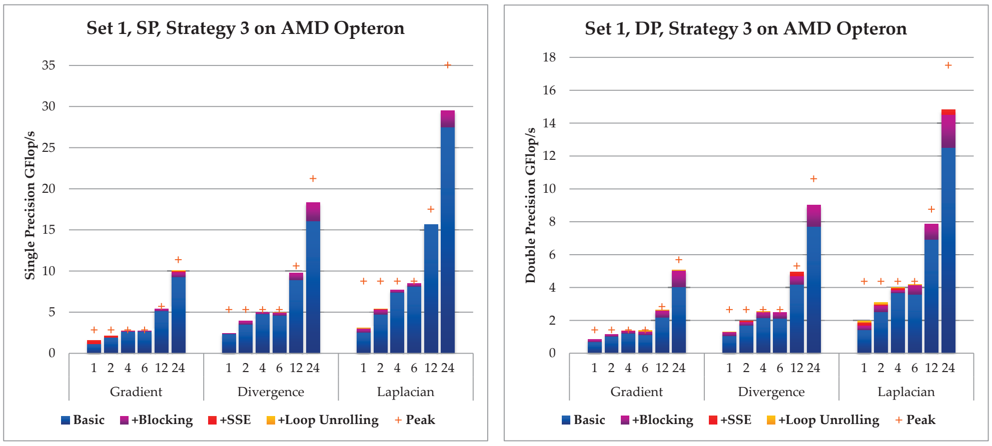
Strategy 3 is a variation of Strategy 2, allowing multiple subsequent blocks to be assigned to one thread ("chunking"). For a 3D stencil, there are 4 parameters to be tuned.

Listing 10.3: *Strategy 3: Cache blocking with chunking.*

```
strategy chunked (domain u, auto dim cb, auto int chunk)
{
  // iterate over time steps
  for t = 1 .. stencil.t_max {
    // iterate over subdomain
    for subdomain v(cb) in u(:;t) parallel schedule chunk
    {
      // calculate the stencil for each point
      // in the subdomain
      for point p in v(:; t)
        v[p; t+1] = stencil (v[p; t]);
    }
  }
}
```

The benchmarks executables were compiled with the GNU C compiler gcc 4.5.2 using the `-O3` optimization flag.

All the following charts are done in the following way: The blue bars visualize the performance numbers for the basic, NUMA-aware parallelization scheme, slicing the domain into equally sized subdomains



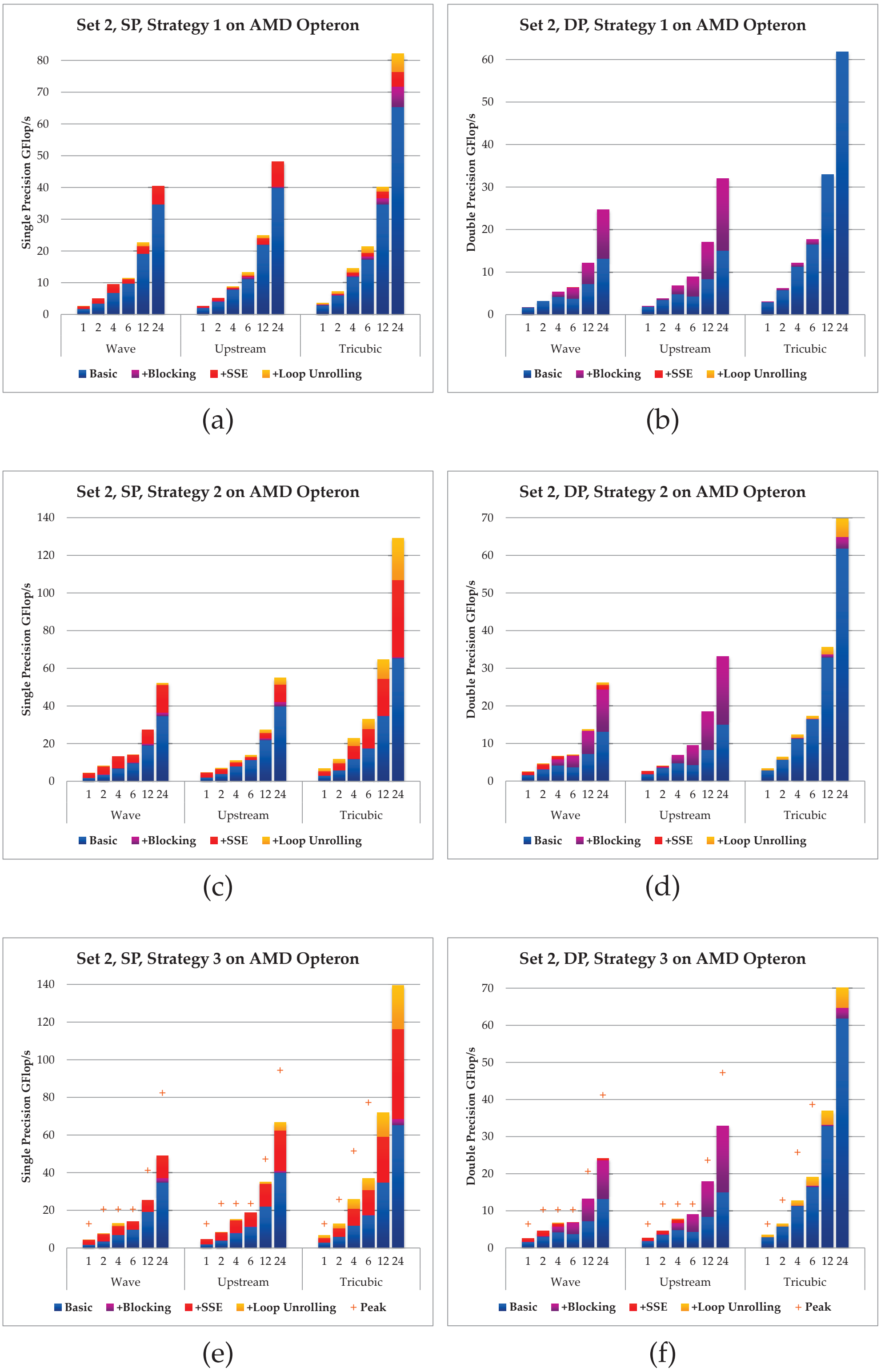
(a) (b)

Figure 10.1: *Set 1 benchmarks on AMD Opteron Magny Cours in single (SP) and double precision (DP) for Strategy 3.*

along the z-axis, the direction in which the indices vary slowest. The purple bars on top show the performance increase due to blocking, applying the best blocking size found by the auto-tuner. The red bars visualize the added benefit of explicit vectorization, additionally to blocking, and the yellow bars show the performance gain from loop unrolling, on top of the other optimizations. When shown, the orange plus signs denote the maximum achievable performance, limited by bandwidth or compute capability.

For Set 1, we only show the performance results of Strategy 3. The other strategies gave almost identical figures. Single precision results (SP) are shown in Fig. 10.1 (a), and double precision results (DP) in Fig. 10.1 (b). The scaling is almost linear in both cases, except when moving from 4 to 6 threads. As the figures show by the orange plus signs, the available bandwidth is already exhausted when using 4 threads, so increasing the on-die concurrency further has no added benefit. As the basic parallelization scheme already almost reaches the maximum performance, blocking does not increase performance significantly.

On one die, around 95% of the maximum attainable performance is reached in double precision. In single precision, both *Gradient* and *Divergence* stencils reach also 95%; the "Laplacian" reaches 90%. Using all 24 threads, in both single and double precision around 85% of the peak performance is reached, except for the single precision *Laplacian*, which attains 80%.


Set 2, SP, Strategy 1 on AMD Opteron

Set 2, DP, Strategy 1 on AMD Opteron

(a)

(b)

Set 2, SP, Strategy 2 on AMD Opteron

Set 2, DP, Strategy 2 on AMD Opteron

(c)

(d)

Set 2, SP, Strategy 3 on AMD Opteron

Set 2, DP, Strategy 3 on AMD Opteron

(e)

(f)

Figure 10.2: *Set 2 benchmarks on AMD Opteron Magny Cours in single (SP) and double precision (DP) for all three Strategies.*

For Set 2, the performance numbers for all of the three Strategies are given in Fig. 10.2. The Strategies are varied from top to bottom in the figure; the sub-figures to the left show single precision results (SP), the sub-figures to the right show double precision results (DP).

Although the basic parallelization delivers the same performance in all Strategies, vectorization and loop unrolling do not have as large an effect in Strategy 1 as in the other Strategies, resulting in poorer over all performance of Strategy 1. This is due to the added control overhead; emulating Strategy 2 with Strategy 1 using the best parameter configuration shows that the poorer performance is not an artifact of the auto-tuner.

The additional parameter in Strategy 3 gives the stencils in single precision some additional performance boost over the performance achieved with Strategy 2, while the double precision results are not affected. The figures show that the gains in performance in single precision come mostly from explicit vectorization and from loop unrolling (especially in the compute-bound *Tricubic* stencil), whereas in the double precision case the explicit SSE code cannot improve the performance gained by optimally blocking the *Wave* and *Upstream* stencil codes. In the compute-bound case, only loop unrolling is able to give a performance gain, the effect of blocking is negligible.

In the best case, the *Wave* stencil reaches 65% of the peak on one die and 60% with all 24 threads in both single and double precision. For the *Upstream* stencil, PATUS achieves 65% of the peak in single precision mode. In double precision the numbers are 80% using one die and 70% when all threads are used. The bandwidth bound *Tricubic* stencil runs at 50% of the peak on one die and at 45% using all 24 threads, both in single and double precision modes.

Analyzing the cache performance (cf. Table 10.2) shows that for the double precision *Wave* stencil blocking significantly reduces the number of L2 misses by around 60%, whereas in single precision mode the L2 misses are not decreased by much. Indeed, for the 200^3 problem used in the benchmarks one plane of data as used in the basic parallelization, has $40,000$ grid points. In single precision, three such planes, each having a data volume of 160 KB, fit into the Opteron's 512 KB L2 cache, but in double precision only one plane (320 KB) fits.

The GNU C compiler auto-vectorizes the code, but using explicit SSE intrinsics and forcing aligned loads removes slow non-aligned moves. In SSE, single precision vectors contain 4 data elements, and double precision vectors contain 2. Thus, higher order stencils with more than the

Wave, 200^3		L1 Cache		L2 Cache	
		Accesses$/10^6$	*Misses*$/10^6$	*Accesses*$/10^6$	*Misses*$/10^6$
SP	Basic	615	0.74	18.7	0.74
	Blocked	624	1.95	21.1	0.51
	Vectorized	191	1.52	19.1	0.51
DP	Basic	758	5.13	36.6	5.08
	Blocked	627	6.99	38.1	1.94
	Vectorized	365	4.78	38.5	1.82

Table 10.2: *Numbers (in millions) of L1 and L2 cache accesses and misses for the single (SP) and double precision (DP)* 200^3 Wave *problem.*

immediate neighbors in the unit stride direction (such as the stencils in Set 2) have higher data reuse in the single precision case (less loads are required, as neighboring values are brought into the registers automatically when loading a SIMD vector). Indeed, the cache analysis shows, that the number of L1 accesses is halved in the double precision case when turning on explicit vectorization, and divided by 4 in single precision.

10.1.2 Intel Xeon Nehalem Beckton

For the experiments on the Intel Xeon platform, the benchmark executables were compiled with Intel C compiler icc 11.1, again using the `-O3` optimization flag. Thus, we could take advantage of Intel's OpenMP implementation, which allegedly has less overhead than GNU's. Also, it allows us to control thread affinity by setting the `KMP_AFFINITY` environment variable. We chose the `compact` scheme, which fills up cores first (the CPU is uses the Hyper-Threading Technology, thus each core accommodates 2 hardware threads), and then sockets. The reverse case, i.e., filling up sockets first, and then cores, which is implemented by the `scatter` scheme, might be the obvious choice for bandwidth-limited compute kernels; however, as we want to run the benchmarks with up to 24 threads, using all available hardware resources. The choice does not really matter as long as we are aware of the fact that at some point (which is when moving from 1 to 2 threads in the `compact` scheme (cf. Fig. 10.3) and when moving from 12 to 24 threads in the `scatter` scheme) the bandwidth does not scale, thus limiting the performance.

Again, both sets of stencils were auto-tuned and benchmarked in both precision modes, but we omitted the Strategies 1 and 2, which performed more poorly. The results are shown in Fig. 10.3.

On the Intel platform, explicit vectorization (and also loop unrolling) is much more pronounced than on AMD, whereas blocking has an almost negligible effect. The reason why vectorization has such a large effect on the Intel platform is that the Intel C compiler does not vectorize this particular code using the `-O3` optimization option. It does, however, vectorize when compiling with `-fast`, but in our tests the performance gain were not substantial. Therefore, explicit SSE intrinsics and forcing data alignment leverage the full power of the Streaming SIMD Extensions. The effect can be seen in all the benchmarks except for the double precision *Gradient* and *Divergence* stencil, for which already the basic parallelization gives the maximum yield.

10.1.3 NVIDIA Fermi GPU (Tesla C2050)

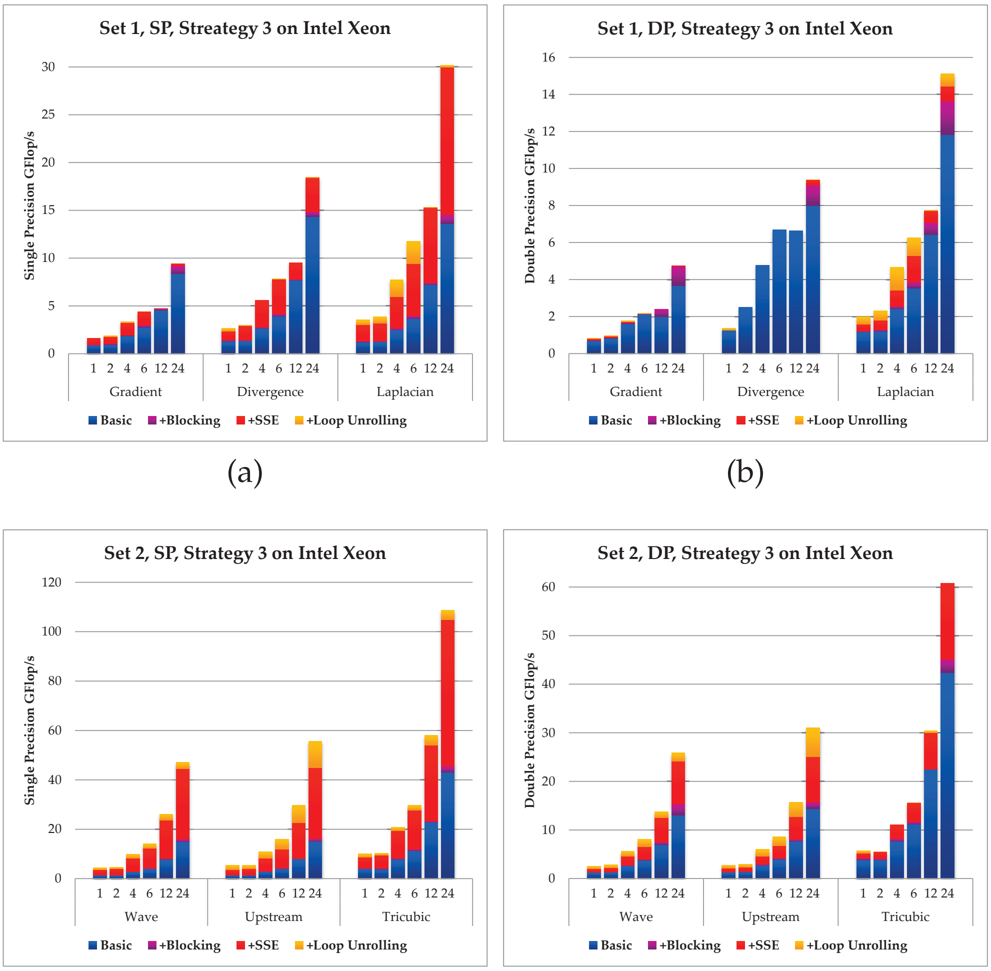
As a proof of concept, GPU results are also included here. While CUDA programming can be learned relatively easily, experience shows that optimizing CUDA C code is non-trivial and a hardware-specific coding style (e.g., making use of shared memory and registers, adhering to data alignment rules, etc.) is absolutely essential for well performing GPU codes. Thus, these results are to be treated as a first step towards GPU support in Patus.

The results shown in Fig. 10.4 are again the stencils from Sets 1 (upper row) and 2 (lower row) in single (to the left) and double precision (to the right). We used only one Strategy: the one shown in Listing 10.4.

Listing 10.4: *Strategy used on the GPU.*

```
strategy gpu (domain u, auto int cbx) {
  // iterate over time steps
  for t = 1 .. stencil.t_max {
    // iterate over subdomain
    for subdomain v(cbx, 1 ...) in u(:; t) parallel
      for point pt in v(:; t)
        v[pt; t+1] = stencil (v[pt; t]);
  }
}
```

It has only one parallelism level, which then is assigned to the lowest level in the hierarchy, i.e., to threads. Each thread receives a small subdomain of size $(\mathtt{cbx}, 1, 1)$ to process, `cbx` being a parameter to be determined



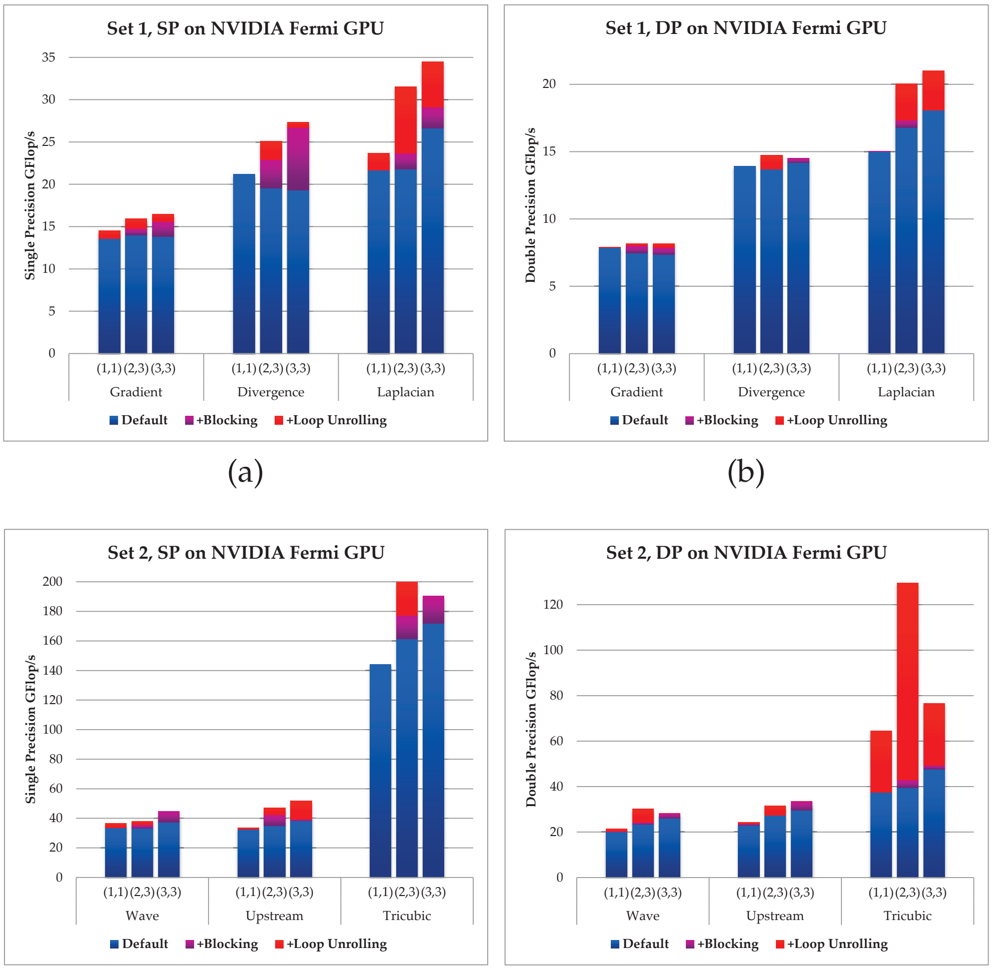
(a) (b)

(c) (d)

Figure 10.3: *Set* 1 *and* 2 *benchmarks on Intel Xeon Nehalem in single (SP) and double precision (DP) for Strategy* 3*.*

by the auto-tuner, in addition to the thread block size, which is implicit in this Strategy.

We use the Strategy in three different indexing modes, though, to determine which of the three yields the best performance result. The indexing modes are plotted as juxtaposed bars in Fig. 10.4. The $(1, 1)$ indexing mode maps the 3D domain onto a one-dimensional thread block and one-dimensional thread index, thus ignoring the fact that thread and thread block indices can have a higher dimensionality. The $(2, 3)$ indexing mode uses a 2D grid and 3D thread blocks, which were the highest supported indexing dimensionalities in pre-Fermi cards and CUDA SDKs before version 4.0. $(2, 3)$-indexing means that the third problem dimension is



(a) (b)

(c) (d)

Figure 10.4: *Set 1 and 2 benchmarks on NVIDIA Fermi in single (SP) and double precision (DP) for one GPU strategy in different indexing modes.*

emulated by the second grid dimension, cf. Chapter 13.2. Therefore the index calculations are more complicated than in $(3,3)$-indexing mode.

The blue bars show the performance of an arbitrarily chosen default thread block sizes of $16 \times 4 \times 4$ and $200 \times 1 \times 1$ for $(1,1)$-indexing, respectively — sizes which exhibited reasonable performance, — setting `cbx` to 1. On top of that, the purple bars show the performance after selecting the best thread block size, and the red bars after choosing the best loop unrolling factor. Loop unrolling included both picking the best value for `cbx` *and* picking the best configuration of unrolling the loop in the unit stride dimension (which obviously is only possible if `cbx` > 1). Generally speaking, the charts show that the performance are about on par for all indexing modes, in most cases, native $(3,3)$-indexing performs slightly better than the other modes, as the index calculations are done by the hardware to the largest extent. An exception is the double precision *Tricubic* stencil, for which two-fold unrolling in $(2,3)$-indexing in fact triples the performance.

There is still room for optimization of the generated code; PATUS currently reaches between 40% and 60% of the peak for the stencils in Set 1 in single precision and between 50% and 70% in double precision. The relative performance of higher-arithmetic intensity kernels in Set 2 is more disappointing, as only 23%–30% of the maximum is reached.

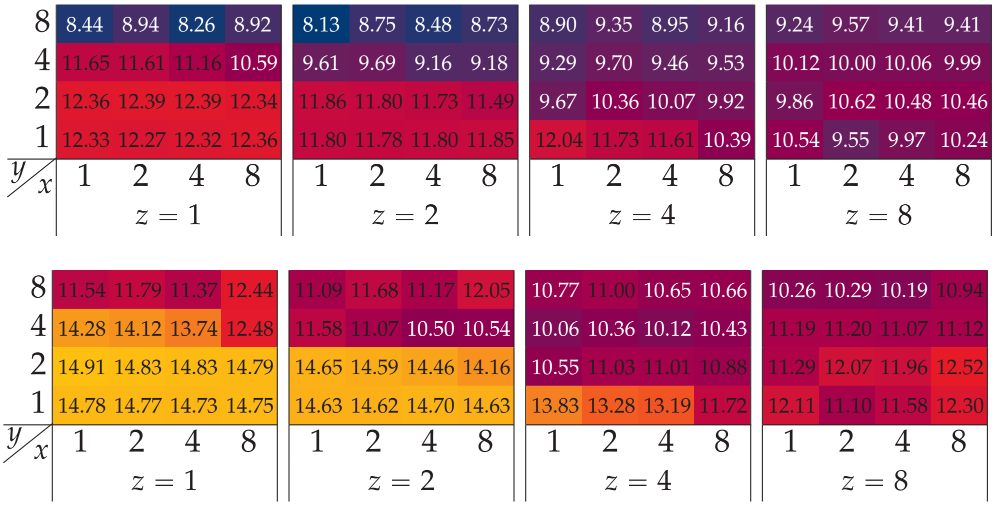
10.2 Impact of Internal Optimizations

10.2.1 Loop Unrolling

We examine various loop unrolling configurations for the basic parallelization scheme and the blocking configuration that was found to give the best performance. The benchmarks for this experiment were run on the AMD Opteron platform using 24 threads. We chose two stencils, the double precision *Laplacian* and the single precision *Tricubic*; the first did not seem to profit from loop unrolling (cf. Fig. 10.1) whereas the second did (cf. Fig. 10.2). We want to evaluate by what amount the performance degrades or improves, and how the performances are distributed.

Figs. 10.5 and 10.6 show heat map plots for the *Laplacian* and the *Tricubic*, respectively. The loop unrolling factors are varied among the x-, y-, and z-axes as powers of 2. The 4 plots in the upper row show the performance numbers in the basic parallelization scheme, the lower rows the numbers after the best block sizes were picked.

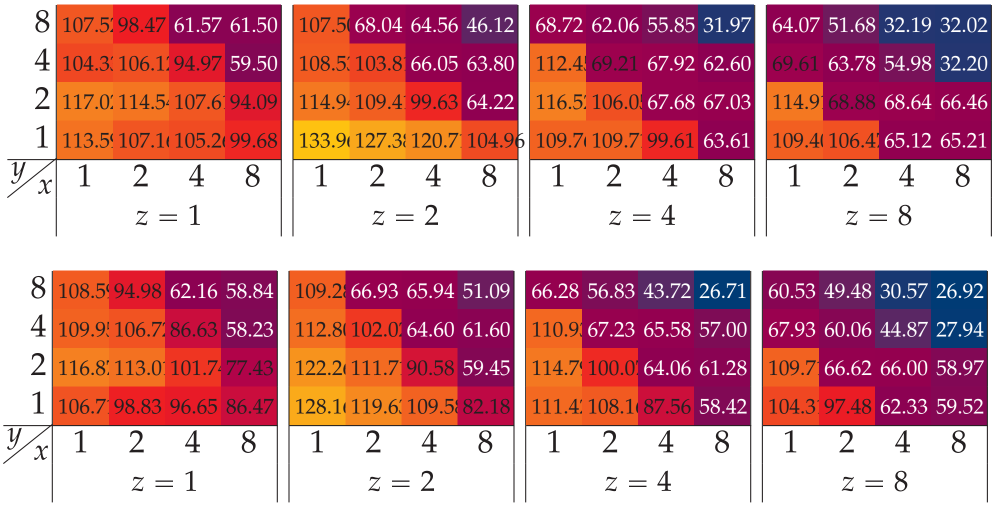
As expected, small loop unrolling numbers yield the best performance for the *Laplacian*, in both basic and blocked cases. More precisely, any loop unrolling factor in unit stride direction (along the x-axis) between 1 and 8 gives an acceptable performance as long as in y- and z-directions the unrolling factors are 1 or 2. This coincides with the deliberation on spatial data locality along the unit stride direction, which is disrupted when the stride within the inner loop becomes too large (which is the case for larger unrolling factors in non-unit stride directions). In both parallelization schemes, the performance degradation is only to around 67% of the maximum for the examined unrolling factors. For the *Tricubic* stencil, in contrast, the performance degrades down to 20% of the maxi-



	z = 1				z = 2				z = 4				z = 8			
y \ x	1	2	4	8	1	2	4	8	1	2	4	8	1	2	4	8
8	8.44	8.94	8.26	8.92	8.13	8.75	8.48	8.73	8.90	9.35	8.95	9.16	9.24	9.57	9.41	9.41
4	11.65	11.61	11.16	10.59	9.61	9.69	9.16	9.18	9.29	9.70	9.46	9.53	10.12	10.00	10.06	9.99
2	12.36	12.39	12.39	12.34	11.86	11.80	11.73	11.49	9.67	10.36	10.07	9.92	9.86	10.62	10.48	10.46
1	12.33	12.27	12.32	12.36	11.80	11.78	11.80	11.85	12.04	11.73	11.61	10.39	10.54	9.55	9.97	10.24

	z = 1				z = 2				z = 4				z = 8			
y \ x	1	2	4	8	1	2	4	8	1	2	4	8	1	2	4	8
8	11.54	11.79	11.37	12.44	11.09	11.68	11.17	12.05	10.77	11.00	10.65	10.66	10.26	10.29	10.19	10.94
4	14.28	14.12	13.74	12.48	11.58	11.07	10.50	10.54	10.06	10.36	10.12	10.43	11.19	11.20	11.07	11.12
2	14.91	14.83	14.83	14.79	14.65	14.59	14.46	14.16	10.55	11.03	11.01	10.88	11.29	12.07	11.96	12.52
1	14.78	14.77	14.73	14.75	14.63	14.62	14.70	14.63	13.83	13.28	13.19	11.72	12.11	11.10	11.58	12.30

Figure 10.5: *Impact of varying loop unrolling factors on a fixed blocking size. Double precision* Laplacian *stencil. Basic parallelization in the upper row, and best block sizes in the lower row.*



	z = 1				z = 2				z = 4				z = 8			
y \ x	1	2	4	8	1	2	4	8	1	2	4	8	1	2	4	8
8	107.5	98.47	61.57	61.50	107.5	68.04	64.56	46.12	68.72	62.06	55.85	31.97	64.07	51.68	32.19	32.02
4	104.3	106.1	94.97	59.50	108.5	103.8	66.05	63.80	112.4	69.21	67.92	62.60	69.61	63.78	54.98	32.20
2	117.0	114.5	107.6	94.09	114.9	109.4	99.63	64.22	116.5	106.0	67.68	67.03	114.9	68.88	68.64	66.46
1	113.5	107.1	105.2	99.68	133.9	127.3	120.7	104.96	109.7	109.7	99.61	63.61	109.4	106.4	65.12	65.21

	z = 1				z = 2				z = 4				z = 8			
y \ x	1	2	4	8	1	2	4	8	1	2	4	8	1	2	4	8
8	108.5	94.98	62.16	58.84	109.2	66.93	65.94	51.09	66.28	56.83	43.72	26.71	60.53	49.48	30.57	26.92
4	109.9	106.7	86.63	58.23	112.8	102.0	64.60	61.60	110.9	67.23	65.58	57.00	67.93	60.06	44.87	27.94
2	116.8	113.0	101.7	77.43	122.2	111.7	90.58	59.45	114.7	100.0	64.06	61.28	109.7	66.62	66.00	58.97
1	106.7	98.83	96.65	86.47	128.1	119.6	109.5	82.18	111.4	108.1	87.56	58.42	104.3	97.48	62.33	59.52

Figure 10.6: *Impact of varying loop unrolling factors on a fixed blocking size. Single precision* Tricubic *stencil. Basic parallelization in the upper row, and best block sizes in the lower row.*

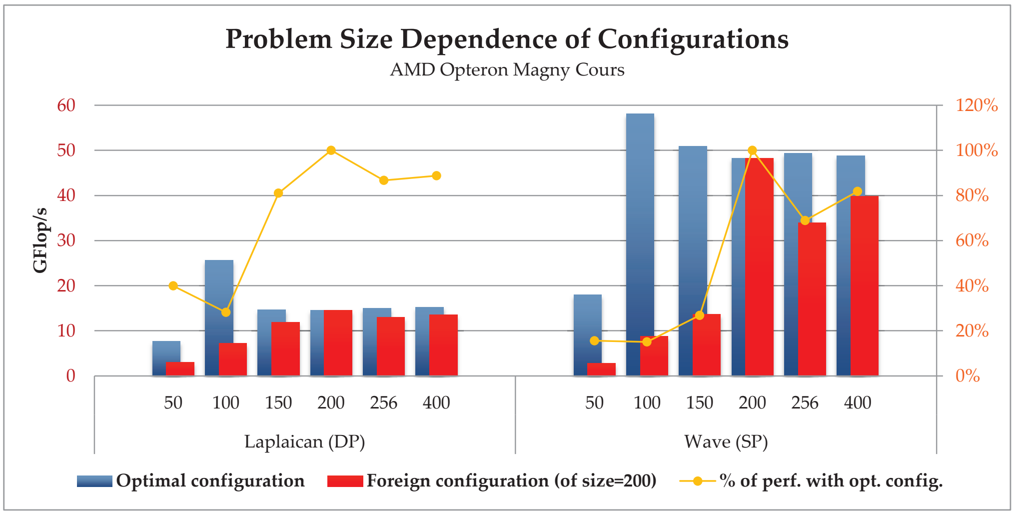


Figure 10.7: *Performance impact of using the optimal configuration for one problem size for other problem sizes.*

mum performance, i.e., loop unrolling has an a lot larger impact on performance. The best performance numbers are achieved for an unrolling factor of 2 in z-direction. As the unrolling factor in z-direction becomes larger, the performance decreases rapidly due to the increased register pressure.

10.3 Impact of Foreign Configurations

After tuning Strategy and code generation characteristics (e.g., block sizes, loop unrolling factors), the resulting configuration works best if in the production run the settings are exactly replicated, in particular, if the problem size is not modified, the number of threads used remains unchanged, and the program is run on the same hardware platform. In this section, we evaluate the consequences when one of the characteristics is changed.

10.3.1 Problem Size Dependence

Fig. 10.7 shows the impact on performance when the problem size is varied for an unchanged parameter configuration obtained for a 200^3 problem. We call this the *default configuration*. We study two stencils, the double precision *Laplacian* and the single precision *Wave* kernel. The benchmarks are run on the AMD Opteron with 24 threads. The blue bars show the maximum performance obtained by applying the appropriate parameter configuration; the overlaid red bars show the performance numbers obtained when using the default configuration. The yellow line shows the percentage of the performance that was achieved with the default configuration; the percentages are given on the right vertical axis. Both stencils exhibit a significantly above-average maximum performance for the 100^3 problem: evidently a sweet spot on the given architecture; the line lengths in unit stride direction are long enough to engage the hardware prefetcher, and the problem is still small enough so that the required planes — after subdividing among the 24 threads — fit into cache so that temporal data locality could be fully exploited.

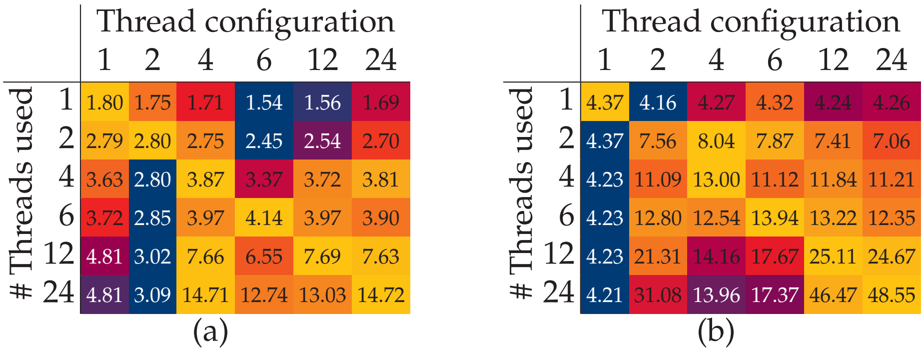
Due to the nature of the chosen Strategy, the performance is abysmal for problem sizes smaller than nominal size: block sizes that are well suited to the original problem size most likely are too large for the smaller problem sizes, and a set of threads is underutilized. The performance is acceptable for problem sizes larger than the nominal one, in particular if the larger problem size is divisible by the original one.

To deal with multiple problem sizes successfully, all the problem sizes could be benchmarked for each of the configurations picked by the search method in the auto-tuning process — provided that the set of admissible problem sizes is limited and the probability distribution over the problem sizes is known — and the objective function to maximize would be modified to be the weighted sum of the performance numbers, with weights corresponding to the probability distribution.

10.3.2 Dependence on Number of Threads

On a fixed architecture, we could alter the number of threads used for the computation. The performance behavior for constant block sizes and varying numbers of threads is shown in Fig. 10.8 for the double precision *Laplacian* (a) and the single precision *Wave* (b) for a 200^3 problem run on the AMD Opteron.

On the horizontal axis in the sub-figures in Fig. 10.8, the configurations are varied (i.e., the caption "1" identifies the configuration obtained for one thread, etc.), and on the vertical axis the number of threads used to execute the benchmark program is varied. Thus, the original and highest numbers occur on the diagonals. (Interestingly, there is one deviation: according to Fig. 10.8 (b), the *Wave* stencil ran faster with two threads us-



	Thread configuration					
# Threads used	1	2	4	6	12	24
1	1.80	1.75	1.71	1.54	1.56	1.69
2	2.79	2.80	2.75	2.45	2.54	2.70
4	3.63	2.80	3.87	3.37	3.72	3.81
6	3.72	2.85	3.97	4.14	3.97	3.90
12	4.81	3.02	7.66	6.55	7.69	7.63
24	4.81	3.09	14.71	12.74	13.03	14.72

(a)

	Thread configuration					
# Threads used	1	2	4	6	12	24
1	4.37	4.16	4.27	4.32	4.24	4.26
2	4.37	7.56	8.04	7.87	7.41	7.06
4	4.23	11.09	13.00	11.12	11.84	11.21
6	4.23	12.80	12.54	13.94	13.22	12.35
12	4.23	21.31	14.16	17.67	25.11	24.67
24	4.21	31.08	13.96	17.37	46.47	48.55

(b)

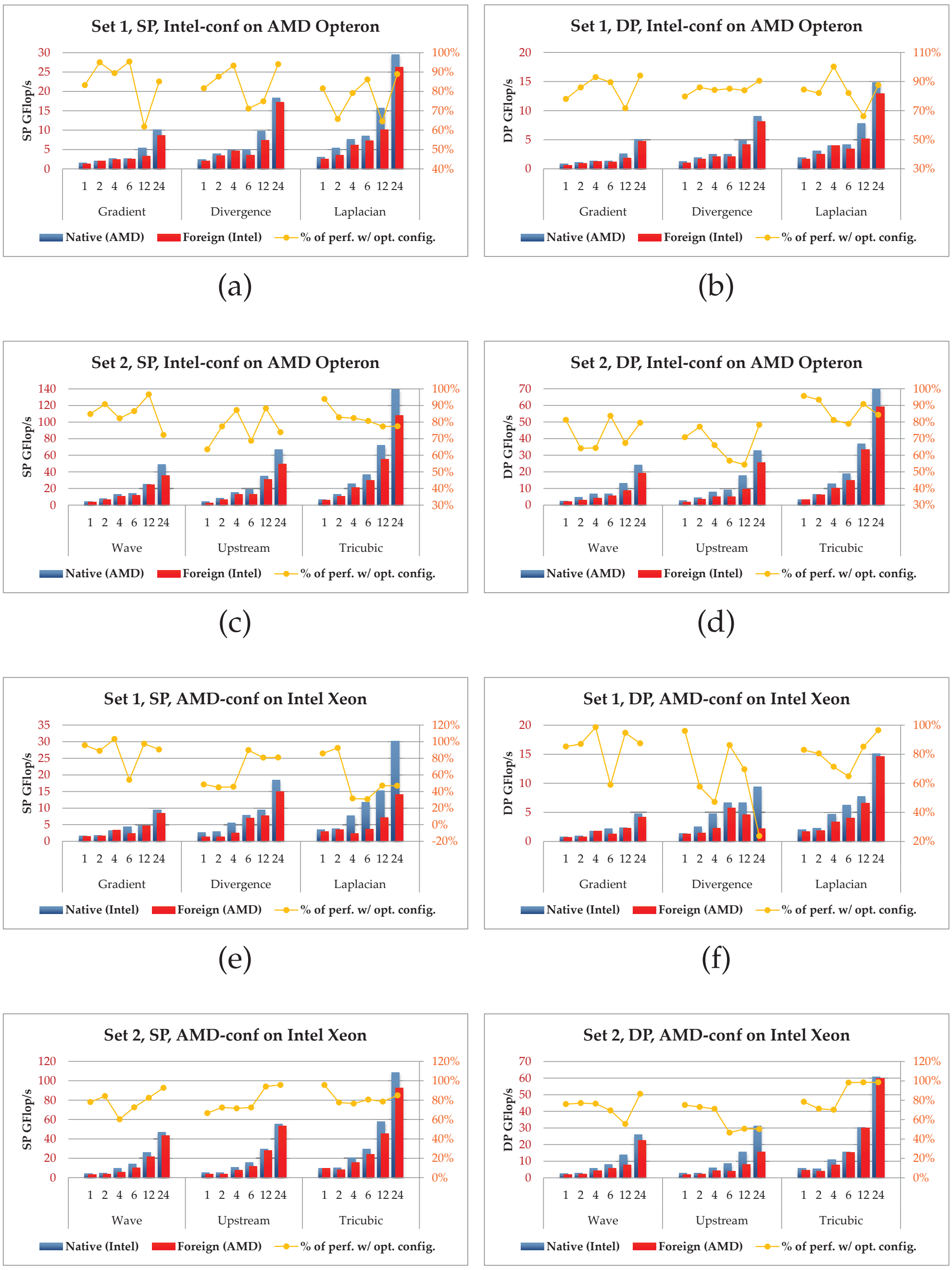
Figure 10.8: *Varying the number of threads for a fixed configuration. Double precision* Laplacian *stencil (a) and single precision* Wave *stencil (b).*

ing the 4-thread configuration than with the native configuration. This is an auto-tuner artifact.) Ideally, the numbers in a horizontal row would remain constant. We can observe the tendency that the configurations to the right of a diagonal element are larger than the ones to the left. This is again due to the nature of the Strategy, which guides the auto-tuner to pick larger block sizes for lower concurrencies. The lacking granularity, when switching to a number of threads larger than the nominal one might cause additional threads to remain idle. In the *Laplacian* stencil the effect is particularly pronounced when more than two threads run the two-thread configuration, and in the *Wave* stencil when more than one thread run the one-thread configuration.

On average, 91% (one thread) down to 66% (24 threads) of the performance of the best configuration is reached in the double precision *Laplacian*, and 97% (one thread) down to 47% (24 threads) in case of the single precision *Wave* stencil.

10.3.3 Hardware Architecture Dependence

Finally, also the hardware platform can be exchanged. We ran all the benchmarks with exchanged configuration sets, i.e., the Intel configurations were applied to benchmarks run on the AMD platform and vice versa. The architectures are quite similar. The major difference is the size of the L2 cache, which is 512 KB on the Opteron cores, and only 256 KB on the Nehalem cores, which, in addition, accommodate two hardware threads each. Thus, we expect that the Intel configurations run reasonably well on the AMD processor, but Magny Cours configurations might degrade the performance on the Intel CPU.



(a) (b) (c) (d) (e) (f) (g) (h)

Figure 10.9: *Using optimal configurations on foreign hardware platforms.*

Indeed, Fig. 10.9 shows acceptable performance matches of Intel configurations on the AMD Opteron in sub-figures (a)–(d). The matches of the AMD configurations on the Intel CPU in sub-figures (e)–(h) exhibit larger deviations.

The blue bars are the native performances in GFlop/s, the overlaid red bars the foreign ones. The yellow lines visualize the relative performance with respect to the native one; the percentages are shown on the right vertical axis.

Generally, the performance numbers on the Magny Cours are no less than 60% of the maximum numbers when using the foreign configurations, 80% on average, in both double and single precision modes and for both sets of stencils. In contrast, on the Intel platform the performance drops down to almost 20% of the original performance in certain cases; on average 75% of the original performance is reached. Again, the average does not change when switching precision modes or stencil sets. However, we can also observe that the AMD configurations give almost perfect results on Intel in certain cases, e.g., in case of single precision *Gradient* or *Upstream* stencils, or the double precision *Tricubic* stencil, where almost 100% of the original performance is brought back for 6, 12, and 24 threads.

Chapter 11

Applications

> Without, however, stepping into the region of conjecture, we will mention a particular problem which occurs to us at this moment as being an apt illustration of the use to which such an engine may be turned for determining that which human brains find it difficult or impossible to work out unerringly.
>
> — Ada Lovelace (1815–1852)

11.1 Hyperthermia Cancer Treatment Planning

Hyperthermia cancer treatment, i.e., application of moderate heat to the body, is a promising modality in oncology that is used for a wide variety of cancer types (including pelvic, breast, cervical, uterine, bladder, and rectal cancers, as well as melanoma and lymphoma). Both animal and clinical studies have proven that hyperthermia intensifies both radio- and chemo-therapies by factors of 1.2 up to 10, depending on heating quality, treatment modality combination, and type of cancer [57, 164, 186]. Hyperthermia is therefore applied in conjunction with both radio- and chemo-therapies.

An effect of hyperthermia treatment is apoptosis of tumor cells, which have a chaotic vascular structure resulting in poorly perfused regions in which cells are very sensitive to heat. Hyperthermia further makes the tumor cells more susceptible to both radiation and certain cancer drugs. There are a variety of reasons for this. Among others, heat naturally increases blood flow and therefore increases drug delivery to the tumor cells, and also increases the toxicity of some drugs. The effect of radiation is amplified as a result of improved oxygenation due to increases in blood flow. In our setting, we are dealing with local hyperthermia where the aim is to focus the energy noninvasively only at the tumor location. This is done by creating a constructive interference at the tumor location using nonionizing electromagnetic radiation (microwaves) and thereby aiming at heating the tumor to $42 - 43$°C, but even lower temperatures can be beneficial.

Figure 11.1: *Head and neck hyperthermia applicator* HYPERcollar *developed at the Hyperthermia Group of Erasmus MC (Daniel den Hoed Cancer Center), Rotterdam, the Netherlands.*

Cylindrical applicators, such as one shown in Fig. 11.1, which shows the *HYPERcollar* applicator developed for head and neck hyperthermia treatment at the Hyperthermia Group of Erasmus MC (Daniel den Hoed Cancer Center) in Rotterdam, the Netherlands [132, 133], feature a number of dipole antennae arranged on the circumference of the applicator, whose amplitudes and phase shifts can be controlled to create the desired electric field inducing heat. Here the aim is to avoiding cold spots within the tumor and hotspots in healthy tissue to maximize tumor heating and limit pain and tissue damage. The water bolus between the patient's skin and the applicator prevents heating of the skin and more importantly, from an engineering point of view, it is an efficient transfer medium for the electromagnetic waves into the tissue: it reduced reflections and allows for smaller antenna sizes. In treatment planning, it is therefore highly relevant to determine the therapeutically optimal antenna settings, given the patient geometry. This leads to a large-scale nonlinear, nonconvex PDE-constrained optimization problem, the PDE being the thermal model shown in Eqn. 11.1, which is known as Pennes's bioheat equation [134], or a variant thereof. Another important aspect of treatment planning is simulating the temperature distribution within the human body given the antenna parameters, which has been successfully demonstrated to accurately predict phenomena occurring during treatment [150, 119]. Simulations are helpful in order to determine the correct doses and help to overcome the difficulty of temperature monitoring during treatment. Other benefits include assistance in developing new applicators and training staff.

In this chapter, we focus on the simulation only. The thermal model is given by the parabolic partial differential equation

$$\rho C_{\mathrm{p}} \frac{\partial u}{\partial t} = \nabla \cdot (k \, \nabla u) - \rho_{\mathrm{b}} W(u) C_{\mathrm{b}} (u - T_{\mathrm{b}}) + \rho Q + \tfrac{\sigma}{2} \|\mathbf{E}\|^2, \tag{11.1}$$

which is the simplest thermal model. On the boundary we impose Dirichlet boundary conditions to model constant skin temperature, Neumann boundary conditions, which account for constant heat flux through the skin surface, or convective boundary conditions [120]. In Eqn. 11.1, u is the temperature field, for which the equation is solved, ρ is density, C_{p} is the specific heat capacity, k is thermal conductivity, $W(u)$ is the temperature-dependent blood perfusion rate, T_{b} is the arterial blood temperature (the subscript "b" indicates blood properties), Q is a metabolic heat source, σ is electric conductivity, and $\mathbf{E}$ is the electric field generated by the antenna. The electric field has to be calculated (by solving Maxwell's equations) before computing the thermal distribution. More elaborate models include, e.g., temperature-dependent material parame-

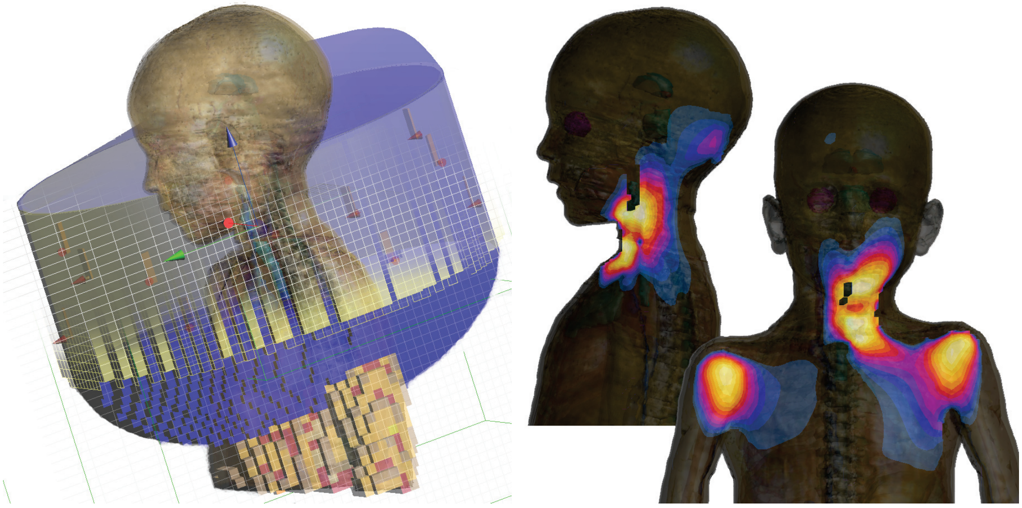


Figure 11.2: *Model of a boy and the HYPERcollar applicator. The left image shows the model and FD discretization, the right images show simulation results. Lighter colors correspond to higher temperatures.*

ters or tensorial heat conductivity to account for the directivity of blood flow [169].

We use a finite volume method for discretizing Eqn. 11.1. The blood perfusion is modeled as a piecewise linear function of the temperature. As a discretized version of Eqn. 11.1 we obtain the stencil expression

$$\begin{aligned} u_{i,j,k}^{(n+1)} &= u_{i,j,k}^{(n)}\left(a_{i,j,k}u_{i,j,k}^{(n)} + b_{i,j,k}\right) + c_{i,j,k} \\ &\quad + d_{i,j,k}u_{i-1,j,k}^{(n)} + e_{i,j,k}u_{i+1,j,k}^{(n)} \\ &\quad + f_{i,j,k}u_{i,j-1,k}^{(n)} + g_{i,j,k}u_{i,j+1,k}^{(n)} \\ &\quad + h_{i,j,k}u_{i,j,k-1}^{(n)} + l_{i,j,k}u_{i,j,k+1}^{(n)}. \end{aligned} \tag{11.2}$$

Note that the coefficients $a, \dots, l$ depend on their location in space. The PDE in Eqn. 11.1 is solved with a simple explicit Euler time integration scheme. Each evaluation of the stencil expression amounts to 16 floating point operations.

11.1.1 Benchmark Results

Performance results for the hyperthermia stencil from Eqn. 11.2 on the AMD Opteron Magny Cours and on the NVIDIA C2050 Fermi GPU are shown in Fig. 11.3. The performance numbers and scalings are given

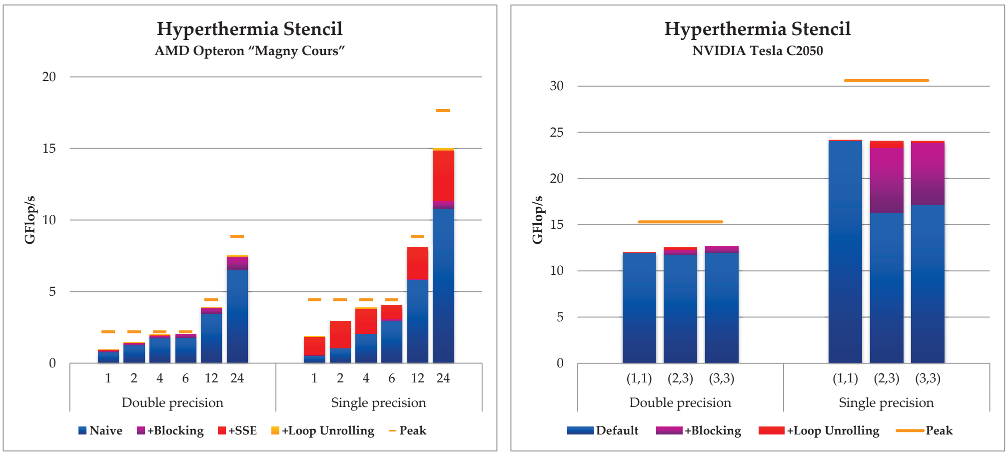


Figure 11.3: *Performance results for the Hyperthermia stencil on the AMD Opteron Magny Cours and on the NVIDIA Fermi GPU.*

for both double (left half of the graphs) and single (right half) precision data types, as the data type can be changed in the solver's user interface. Usually, single precision is sufficient in practice. The performance results were obtained for a problem domain of 200^3 grid points.

The orange bars mark the maximum reachable performance for the stencil's arithmetic performance (0.16 Flop/Byte for double precision and 0.33 Flop/Byte for single precision — or 1.33 Flops per transferred data element, accounting for write allocate traffic). The graphs show that in both cases (single and double precision) on the AMD Opteron around 85% of the optimal performance was reached when using all available NUMA domains. Using only one NUMA domain (one to six threads in the figure), around 90% of the maximum performance is reached. Note that using more than 4 threads on one NUMA domain does not increase performance. The two remaining threads could be used to stream data into the caches and thereby to potentially increase the performance by perfectly overlapping computation and communication.

For double precision, blocking gives a moderate performance increase of 16% over the base line with 24 threads. In the single precision case, vectorization and blocking help to increase the performance by around 40%. This suggests that padding the grids correctly so that no unaligned vector loads have to be carried out yields a major performance gain.

Similarly, on the GPU around 80% of the attainable performance was reached. Interestingly, switching indexing modes did not change the performance. The default thread block sizes (200 in the 1D indexing case and $16 \times 4 \times 4$ in the other cases) gave no relevant performance increase, meaning that the default choice was already a good choice for the thread block size. Indeed, for both indexing modes, the auto-tuner changed the default configuration only slightly to $16 \times 2 \times 6$ (and alternatively to $72 \times 2 \times 2$, which performed equally well). In the single precision case, however, the thread block size was increased to 40–50 for both indexing modes, so that in both single and double precision modes about the equal amount of contiguous memory was accessed. Making explicit use of the GPU's shared memory might increase the performance further.

11.2 Anelastic Wave Propagation

The Anelastic Wave Propagation code AWP-ODC of the Southern California Earthquake Center (SCEC), which was developed by Olsen, Day, Cui, and Dalguer [49] is a scientific modeling code for simulating both dynamic rupture and earthquake wave propagation. It has been used to conduct numerous significant simulations at the SCEC. It provides the capability of simulating the largest earthquakes expected around the San Andreas Fault in southern California at high shaking frequencies (up to 2 Hz in the simulation described below), which allow scientists to understand seismic risks, and thereby help mitigate life and property losses, and let scientists gain new insight about ground motion levels to be expected for a great earthquake along the fault.

The largest simulations done was a simulation of a magnitude-8 earthquake rupturing the entire San Andreas Fault from central California to the Mexican border, a fault length of 545 km. The result is shown in Fig. 11.4. A full description and discussion of the simulation can be found in [49]. The simulation required a discretization of a $810 \times 405 \times 85\ \text{km}^3$ volume with a 40 m-resolution mesh, resulting in $4.4 \cdot 10^{11}$ voxels. The simulation was executed in a 24 hour production run — corresponding to 6 minutes of wave propagation — on around $223,000$ nodes of Jaguar, a Cray XT5 supercomputer, operated at the Oak Ridge National Laboratory in Oak Ridge, Tennessee, USA. On Jaguar, the code achieved a sustained performance of 220 TFlop/s. Simultaneously with this simulation, the code (Fortran, parallelized with MPI) was shown to be highly scalable.

The model's governing elastodynamic equations is the following sys-

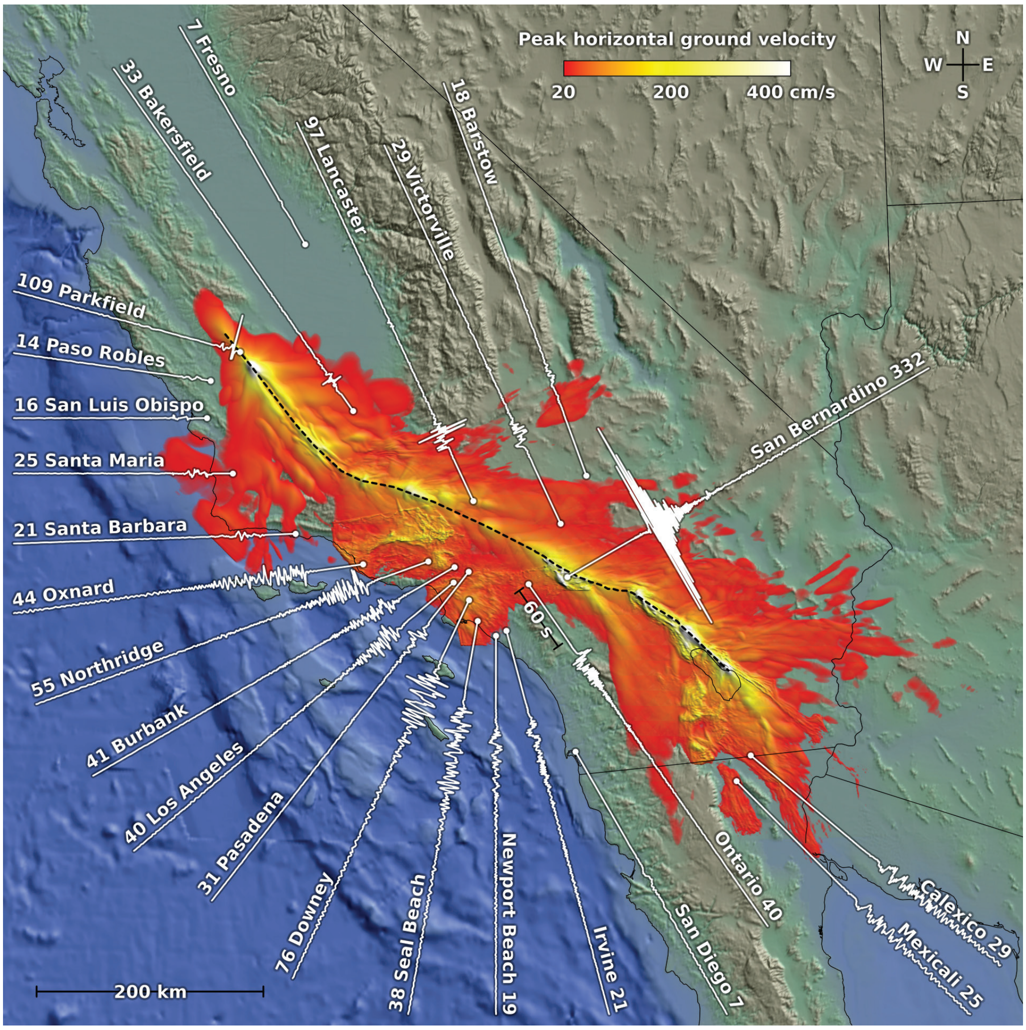


Figure 11.4: *Peak ground velocities derived from a magnitude-8 earthquake simulation along the southern California San Andreas Fault with seismograms and peak velocities for selected locations.*
Image courtesy: *Southern California Earthquake Center, Y. Cui, K. B. Olsen, T. H. Jordan, K. Lee, J. Zhou, P. Small, D. Roten, G. Ely, D. K. Panda, A. Chourasia, J. Levensque, S. M. Day, and P. Maechling* [49].

tem of PDEs [49, 50]:

$$\begin{aligned} \frac{\partial \dot{\mathbf{u}}}{\partial t} &= \rho^{-1}\nabla \cdot \boldsymbol{\sigma} \\ \frac{\partial \boldsymbol{\sigma}}{\partial t} &= \lambda(\nabla \cdot \dot{\mathbf{u}})\mathbf{I} + \mu(\nabla\dot{\mathbf{u}} + \nabla\dot{\mathbf{u}}^{\mathsf{T}}). \end{aligned} \tag{11.3}$$

The dependent variables are the velocity vector field $\dot{\mathbf{u}} = (\dot{u}_x, \dot{u}_y, \dot{u}_z)$ and the stress tensor $\boldsymbol{\sigma} = \begin{pmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{zx} & \sigma_{zy} & \sigma_{zz} \end{pmatrix}$. λ and μ are the Lamé coefficients, ρ is the density, and $\mathbf{I}$ is the identity tensor.

The equations are discretized in the finite difference method — according to [49] the best trade-off between accuracy (fourth-order discretizations of differential operators in space and second-order discretizations in time are used throughout the code), computational efficiency (stencil codes map well to parallel architectures without large overhead such as graph partitioning required for finite element codes), and ease of implementation. The velocity-stress wave equations are solved with an explicit scheme on a staggered grid with equidistant mesh points. A split-node approach is used to model dynamic fault ruptures [50]. As absorbing boundary conditions, the code uses perfectly matched layers or dampening within a *sponge layer*.

11.2.1 Benchmark Results

We selected 4 kernels (corresponding to 4 Fortran routines in the original code) for the benchmarks, which are the 3D stencil routines in which most of the compute time is spent. Table 11.1 shows an overview of the kernels. The corresponding stencil specifications can be found in Appendix C, which are translations to the PATUS stencil specification DSL from the original Fortran code. The discretization of Eqns. 11.3 can be found in [50] and [49].

The stencils are all fourth-order discretizations of the spatial differential operators. The kernels listed in Table 11.1 are to be understood as prototypes; e.g., the actual code contains three variants of *uxx1* — which calculates the velocity field $\dot{u}_x$ in x-dimension, — namely two more for the y and z-dimensions, which are almost identical to *uxx1*. We therefore expect a similar performance and scaling behavior for these related kernels. Similarly, there are two other variants of *xy1*. *xy1* computes the σ_{xy} component of the stress tensor, the other variants compute σ_{xz} and σ_{yz}. Note that the stress tensor $\boldsymbol{\sigma}$ is symmetric. Both *xyz1* and *xyzq* compute

Name	Description	Flops/Stencil	Arith. Int.
uxx1	Velocity in x-dimension	20	0.70 Flop/B
xy1	Stress tensor component σ_{xy}	16	0.65 Flop/B
xyz1	Stress tensor components σ_{xx}, σ_{yy}, σ_{zz}	90	1.58 Flop/B
xyzq	Stress tensor components σ_{xx}, σ_{yy}, σ_{zz} in viscous mode	129	1.22 Flop/B

Table 11.1: *Summary of the AWP kernels which were used in the performance benchmarks.*

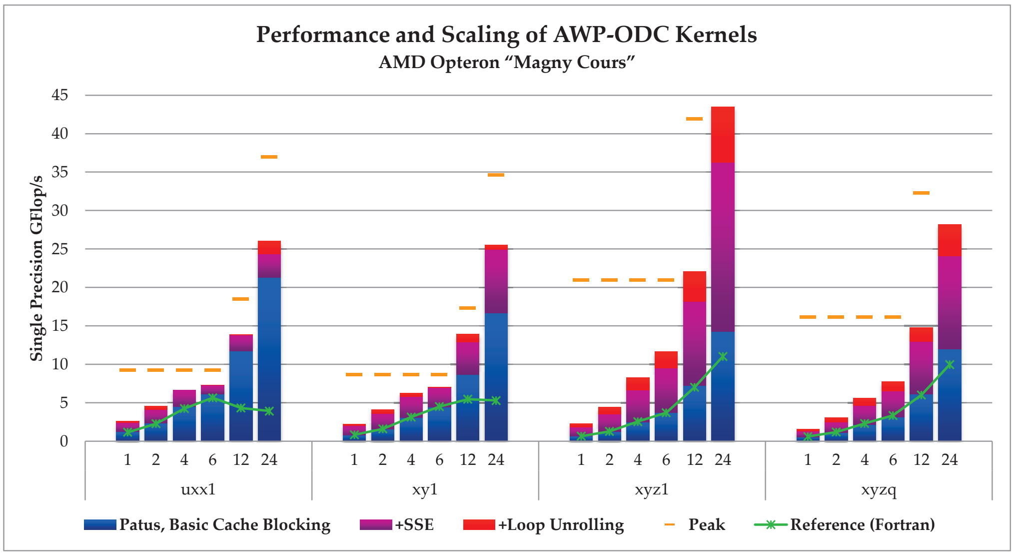


Figure 11.5: *AWP kernel benchmarks on the AMD Opteron Magny Cours.*

the remaining components of the stress tensor, σ_{xx}, σ_{yy}, σ_{zz}, all in one kernel.

The performance benchmarks were done only in single precision, since the original application only uses that precision mode. Again, the orange markers show the maximum attainable performance for the lower arithmetic intensity kernels. On one die (6 threads) the *uxx1* and *xy1* kernels reach around 80% of the peak and around 70% if all 24 threads are used. The theoretical maximum is quite high for the *xyz** kernels, and the reason why only a fraction (around 40%–50%) of the maximum was achieved lies in the arithmetic operations: the kernels contain many divisions (18 in both cases), which are notorious for incurring pipeline stalls. In this case, many of the divisions could be removed by pre-computing and storing the inverses of the Lamé parameters. This manual optimiza-

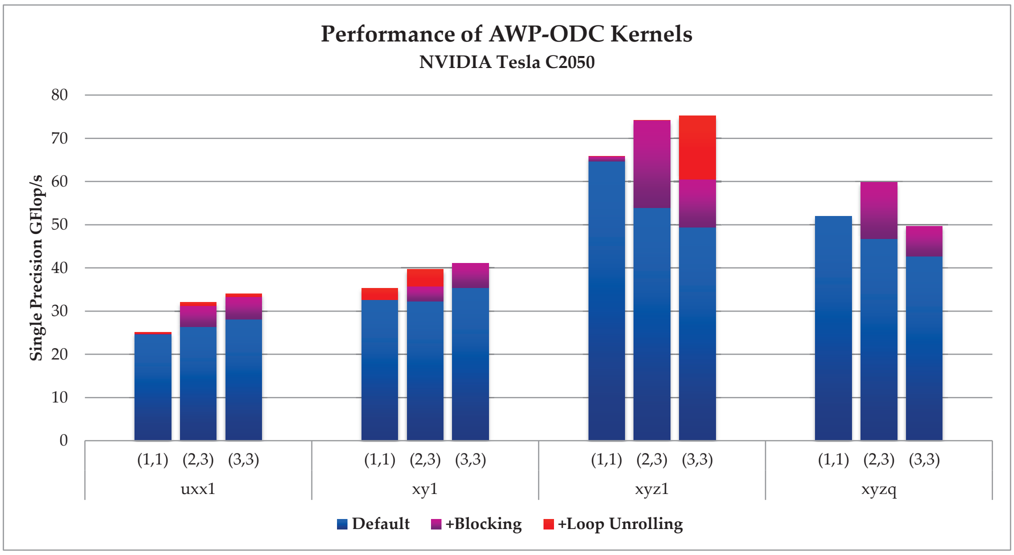


Figure 11.6: *AWP kernel benchmarks on the NVIDIA Tesla C2050 Fermi GPU.*

tion was indeed carried out in an optimized version of the AWP-ODC code.

A possible way for PATUS to remove divisions automatically is to rewrite arithmetic expressions by virtue of the equality $\frac{\alpha}{a} + \frac{\beta}{b} = \frac{\alpha b + \beta a}{ab}$. This transformation removes one division, but introduces three multiplications. For similar transformation of larger sums $\sum_i \frac{\alpha_i}{a_i}$ there is also a tuning opportunity: finding the right trade-off between the number of divisions removed and the number of multiplications added.

The green line shows the performance of the reference Fortran code, which was parallelized by inserting an OpenMP sentinel above the outer most spatial loop. No NUMA optimization was done, which is evident from the scaling behavior of the reference *uxx1* and *xy1* kernels. The arithmetic intensities of the *xyz** kernels are higher; hence the NUMA effect is mitigated to some extent by the relatively high number of floating point operations, and the performance can increase further when going to 2 and 4 NUMA domains (12 and 24 threads, respectively).

The blue bars show the performances of auto-tuned blocked codes, including the NUMA optimization, and relying on the compiler (GNU gfortran/gcc 4.5.2 with the `-O3` optimization flag, for both the reference codes and the generated PATUS codes) to do the vectorization. With the NUMA optimization enabled, the performance scales almost linearly up to 24 threads. Fig. 11.5 shows that explicit use of SSE intrinsics and padding for optimal vector alignment results in a significant performance boost, in particular for the *xyz1* kernel, where explicit vectorization gave a performance increase of 150%. Activating and tuning for loop unrolling gave another slight gain in performance.

Overall, PATUS achieved speedups between 2.8× and 6.6× when 24 threads were used on the AMD Opteron Magny Cours with the NUMA optimization and a cache blocking Strategy.

The GPU performance results are shown in Fig. 11.6. Again, the results are for single precision stencils, and the three indexing modes were used: one-dimensional thread blocks and grids, three-dimensional thread blocks and a two-dimensional grid, and both three-dimensional thread blocks and grids, supported as of CUDA 4.0 on Fermi GPUs. For both the *uxx1* and *xy1* kernels, the default thread block size of $16 \times 4 \times 4$ threads was an adequate choice, and tuning the thread block sizes increased the performance only slightly. In both cases, the fully 3D indexing mode outperformed the $(2, 3)$-dimensional indexing by a tight margin due to the simplified index calculation code. Surprisingly, the fully 3D indexing delivered worse performance in the *xyz** kernels. Loop unrolling could compensate for the loss in performance for the "xyz1" kernel — the only case in which loop unrolling really showed a significant benefit. Also in this case we hope to be able to increase the performance in the future by making use of the GPU's shared memory.

Part IV

Implementation Aspects

Chapter 12

Patus Architecture Overview

> The distinctive characteristic of the
> Analytical Engine, and that which has
> rendered it possible to endow mechanism
> with such extensive faculties as bid fair to
> make this engine the executive right-hand of
> abstract algebra, is the introduction into it of
> the principle which Jacquard devised for
> regulating, by means of punched cards, the
> most complicated patterns in the fabrication
> of brocaded stuffs.
>
> — Ada Lovelace (1815–1852)

PATUS is built from four core components as shown in the high-level overview in Fig. 12.1: the parsers for the two input files, the stencil definition and the Strategy, the actual code generator, and the auto-tuner. PATUS is written in Java and can be therefore run on any major current operating system including Linux, MacOS, and Microsoft Windows. PATUS uses Coco/R [116] as generator for the stencil specification and Strategy parsers, and also for the parser used to interface with the computer algebra system Maxima [20], which is used as a powerful expression simplifier. The Cetus framework ([157, 11]), a compiler infrastructure for source-to-source transformations, provides Java classes for the internal representations for both the Strategies and the generated code, i.e., the

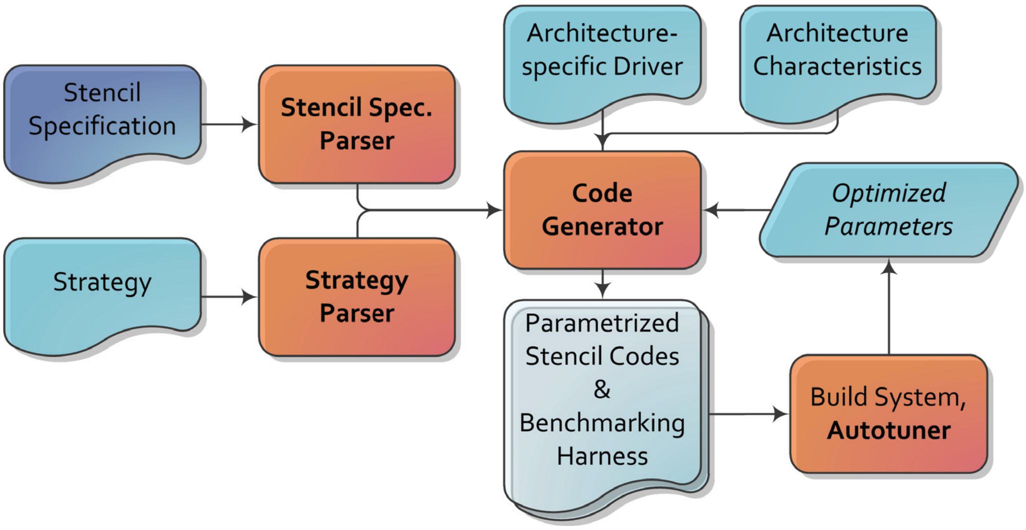


Figure 12.1: *A high-level overview over architecture of the* PATUS *framework.*

Strategy parse tree and the abstract syntax tree of the generated code, and Cetus also provides the mechanism for unparsing the internal representation of the generated code.

The internal representation of the stencil specification consists of the domain size and number-of-iterations attributes and a graph representation of the actual stencil parts described in the stencil `operation`. This will be discussed in detail in section 12.1. The Strategy is transformed to an abstract syntax tree that is used as a template by the code generator. These structures are passed as input to the code generator, along with an additional configuration describing the characteristics of the hardware and the programming model used to program the architecture and specifies the code generation back-end to use. The code generator produces C code for variants of the stencil kernel and also creates an initialization routine that implements a NUMA-aware data initialization based on the parallelization scheme used in the kernel routine.

Along with an implementation for the stencil kernel, the code generator also creates a benchmarking harness from an architecture- and programming model-specific template into which the dynamic memory allocations, the grid initializations, and the kernel invocation are substituted. The benchmarking harness expects the problem-specific parameters related to the domain size (specified in the stencil specification), the Strategy-specific `auto` parameters, as well as internal code generation parameters (currently loop unrolling factors) to be provided to the benchmarking executable as command line arguments.

Currently, the post-code generation compilation and auto-tuning processes have yet to be initiated manually; as of yet, the auto-tuner is still a decoupled part of the system. In consequence, also the feedback loop, returning the set of parameters for which the generated stencil kernel performs well to the code generator in order to substitute them into the parametrized code, has not been implemented yet. Note that it might not even be desirable to do so in a fully automated fashion, since the code generator and the auto-tuning system (and consequently the benchmarking executable) might run on different systems. Also, the auto-tuning subsystem can be used independently for tuning existing parametrized codes not generated by PATUS.

12.1 Parsing and Internal Representation

12.1.1 Data Structures: The Stencil Representation

The stencil specification parser turns a stencil specification into an object structure for which the class diagram is shown in Fig. 12.2. The `StencilCalculation` encapsulates the entire specification including the domain size and the number of iterations, a list of arguments, which will be reflected in the function signature of the generated stencil kernel function, and the actual stencil structure represented by the `StencilBundle`. As an `operation` in the stencil specification can have not only one, but multiple stencil expressions that are computed within a single stencil kernel, the `StencilBundle` contains a list of `Stencil` objects, which are both a graph representation of the geometrical structure of stencil and an arithmetic expression (a Cetus `Expression` object). The graph structure is represented by sets of `StencilNodes`, one set of input nodes, i.e., the nodes whose values are read, and one set of output nodes, the nodes to which values are written in the computation.

Stencil nodes represent a position in the grid relative to the center point, both in space and time. Additionally, they capture which grid is accessed by accessing a stencil node's value. In a mathematical notation, let

$$\mathcal{N} := \mathbb{Z}^d \times \mathbb{Z} \times \mathbb{N}$$

be the index space consisting of a d-dimensional spatial component, a temporal component and a natural-numbered counting component. Then,

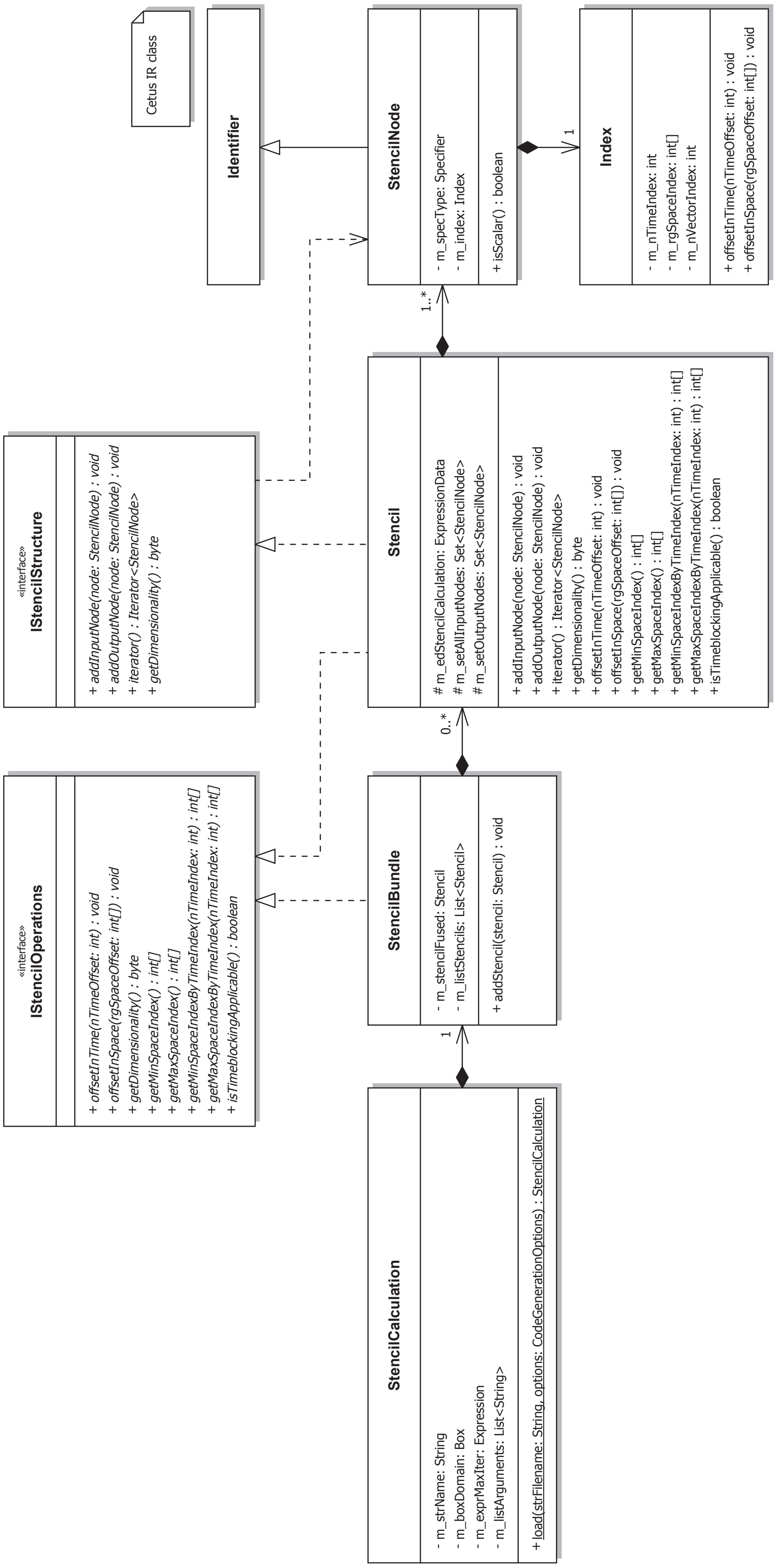


Figure 12.2: *UML class diagram of the internal representation of the stencil specification.*

a *stencil node* is an element of $\mathcal{N}$. Furthermore, let

$$\begin{aligned} \alpha : \mathcal{N} &\twoheadrightarrow \mathbb{Z}^d, \\ \tau : \mathcal{N} &\twoheadrightarrow \mathbb{Z}, \\ \iota : \mathcal{N} &\twoheadrightarrow \mathbb{N} \end{aligned}$$

be the canonical projections onto the spatial, temporal, and counting component, respectively. The spatial component defines the spatial location of the stencil node relative to the center node, the temporal component specifies to which temporal instance of the grid, relative to the current time step, the stencil node belongs, and the counting component distinguishes between different grids. For instance, a stencil for the divergence operator, which maps a vector field to a function, in a d-dimensional setting requires a vector-valued input grid with d components, or, equivalently, d input grids, which are numbered by the counting component.

Definition 12.1. *A* stencil *is denoted by a triple* $(\Sigma^i, \Sigma^o, \Phi)$, *where* $\Sigma^i, \Sigma^o \subseteq \mathcal{N}$ *are finite subsets of* $\mathcal{N}$*;* Σ^i *is the set of* input stencil nodes *and* Σ^o *is the set of* output stencil nodes. *The mapping*

$$\Phi : 2^{\Sigma^i} \longrightarrow \left\{ f : \mathbb{R}^{\left|2^{\Sigma^i}\right|} \to \mathbb{R} \right\} \times 2^{\Sigma^o}$$

assigns a real-valued function in the input stencil nodes (the actual stencil expression), and a set of output stencil nodes (the nodes to which the value is assigned after evaluation of the function) to a subset of input stencil nodes.

In the UML class diagram in Fig. 12.2, Φ corresponds to the stencil expression `m_edStencilExpression` in the `Stencil` class.

Each `Stencil` instance in the list of stencils in the `StencilBundle` represents an object $\left(\Sigma^i_\ell, \Sigma^o_\ell, \Phi_\ell\right), \ell = 1, \ldots, k$. The *fused* stencil, `m_stencilFused`, contained in the `StencilBundle` is a stencil structure synthesized from all the stencils within the specification, $\Sigma^\bullet_{\text{fused}} := \bigcup_{\ell=1}^k \Sigma^\bullet_\ell$. Reasoning about which grids are used or transferred is done based on the fused stencil.

The stencil node n with spatial component $0 \in \mathbb{Z}^d$, i.e., $\alpha(n) = 0$ is called the *center node*; it corresponds to a grid access at `[x,y,z]` in the stencil specification of a 3-dimensional stencil. For each output stencil node $n^o \in \Sigma^o$ we require that the temporal index is larger than $\max\limits_{n^i \in \Sigma^i}\{\tau(n^i)\}$. (Note that if this requirement is not fulfilled, the method would not be an explicit method and would require solving a system of equations.)

Internally, the stencil node components are normalized by shifting the spatial and temporal components in Σ^i and Σ^o such that the spatial indices of the output stencil nodes are $0 \in \mathbb{Z}^d$, and the largest temporal component is 0, i.e.,

$$\begin{aligned} \alpha(n) &= 0 \quad \forall n \in \Sigma^o, \\ \max_{n \in \Sigma^o} \tau(n) &= 0. \end{aligned}$$

The notion of stencil nodes, in conjunction with a Strategy, enables us to map grids, or parts of grids, to arrays which are stored physically in main memory or temporarily in some local memory. In section 13.1 we will discuss in detail how this is done. Also, stencil nodes provide a mechanism to determine the size of an input subdomain required to calculate a result subdomain of a certain size. The fused stencil holds the information for all the subdomain that need to be kept in (local) memory at a given point in time.

12.1.2 Strategies

The Coco/R-based Strategy parser creates a traditional parse tree composed of Cetus IR objects. To encapsulate the special, PATUS Strategy-specific loop structures, additional IR classes have been added. Specifically, `RangeIterators` represent traditional *for* loops, and `SubdomainIterators` represent loops that iterate a smaller subdomain over a parent domain.

Both `RangeIterators` and `SubdomainIterators` extend the class `Loop` which encapsulates the common attributes as shown in Fig. 12.3: the number of threads, chunk size (i.e., the number of consecutive iterates assigned to one thread), and the level of parallelism on which the loop was defined. `RangeIterators` extend the list of attributes to the loop index variable, start and end expressions and the step by which the index variable is incremented in each iteration. A `SubdomainIterator` has two special identifier objects representing the parent domain and the subdomain being iterated within the parent domain. A `SubdomainIdentifier` inherits from the Cetus IR class `Identifier` and adds specific attributes: spatial and temporal offsets as defined in the Strategy and a list of indices through which an array of subdomain identifiers in the Strategy is accessed. This is not to be confused with a stencil node's counting index; a Strategy subdomain is a placeholder for all the stencil grids (defined in the stencil specification) simultaneously.

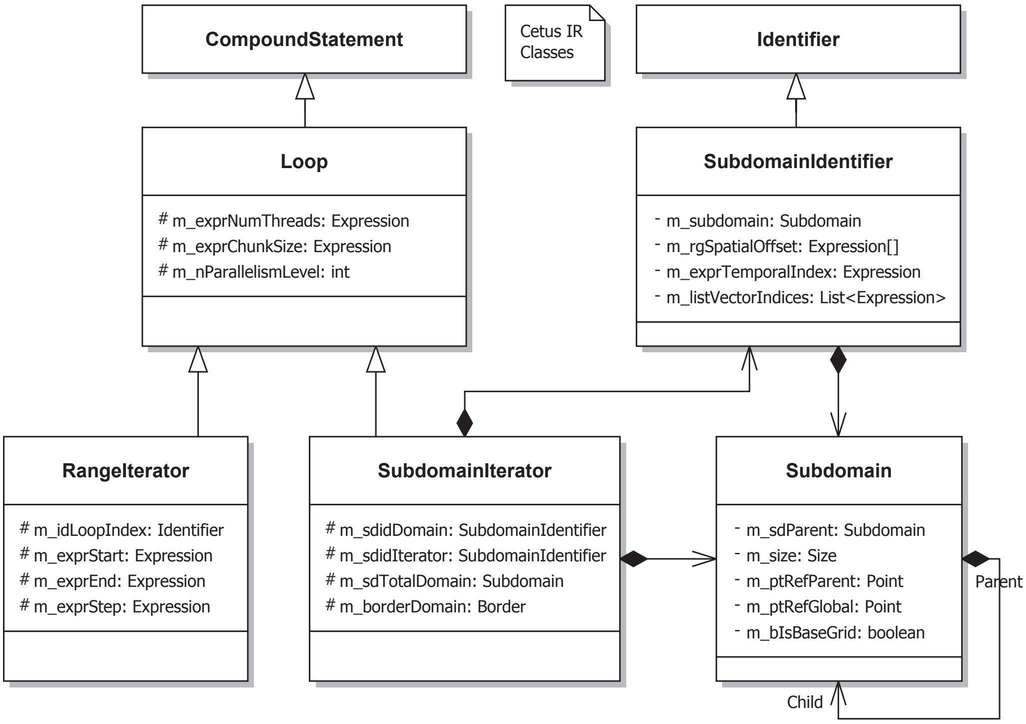


Figure 12.3: *UML class diagram for Strategy loop IR classes.*

For representing the stencil calls and arithmetic expressions, standard Cetus IR class instances are used. The internal representation of the Strategy is used as a template by the code generator into which the concrete stencil computation and the concrete grids are substituted: special Strategy-specific IR objects are gradually replaced by Cetus IR objects that have a C representation. For instance, Strategy-type loops are converted to C `for` loops, and grid-stencil node pairs are converted to linearly indexed grid arrays.

The Strategy parser exploits the fact that the stencil representation is already constructed when the Strategy is being parsed. Hence, stencil properties (e.g., dimensionality, minimum and maximum spatial stencil node coordinates) and aspects of the Strategy that depend on stencil properties, such as vectors of length d specifying Strategy subdomain sizes, where d is the dimensionality of the stencil, or vector subscripts depending on d, are instantiated concretely. For instance, if d is known to be 3, the Strategy subscript expression `(1 ...)` becomes `(1, 1, 1)` at the time of parsing, or a variable `b` of type `dim` becomes `(b_x, b_y, b_z)`. Thus, Strategy subscript expressions are resolved at parse time, and the internal representation only holds the concrete instantiations.

12.2 The Code Generator

The objective of the code generator is to translate the Strategy into which the stencil has been substituted, into the final C code. In particular, it transform Strategy loops into C loops and parallelizes them according to the specification in the Strategy, and it unrolls and vectorizes the inner-most loop containing the stencil calculation if desired; it determines which arrays to use based on the Strategy structure and the grids defined in the stencil specification and calculates the indices for the array accesses. The code generator classes are contained in the package `ch.unibas.cs.hpwc.patus.codegen`.

Fig. 12.4 shows the UML class diagram of the code generation package. `CodeGeneratorMain` is the main entry point of the PATUS code generation module. The method `generateCode` parses both the stencil and Strategy files and invokes the actual code generator `CodeGenerator`. The `generate` method of the latter first “resolves” the Strategy `parallel` keyword: by means of the `ThreadCodeGenerator` an internal representation for the per-thread code of the kernel function is created based on the internal representation of the Strategy. The `ThreadCodeGenerator` class provides methods for parallelizing range iterators and subdomain iterators. The resulting AST is processed by the `SingleThreadCodeGenerator`, which invokes the specialized sub-code generators:

- The class `SubdomainIteratorCodeGenerator` translates Strategy subdomain iterators to C loop constructs. For an iterator over a d-dimensional subdomain, it creates a loop nest with d loops, inferring the loop bounds from the surrounding subdomain iterator or from the problem domain. The inner-most loop is both unrolled and vectorized if these transformations are enabled in the code generation configuration. More details on unrolling and vectorization are given in Chapters 14.1 and 14.3. The actual code generation is done recursively in the inner `CodeGenerator` class.

- The `LoopCodeGenerator` is the counterpart for range iterators, which are simply expanded to C `for` loops. No unrolling or vectorization is done.

- The `StencilCalculationCodeGenerator` substitutes stencil nodes constituting the stencils by accesses to the arrays that contain the grid values. The actual code generation is done within the inner `Code-`

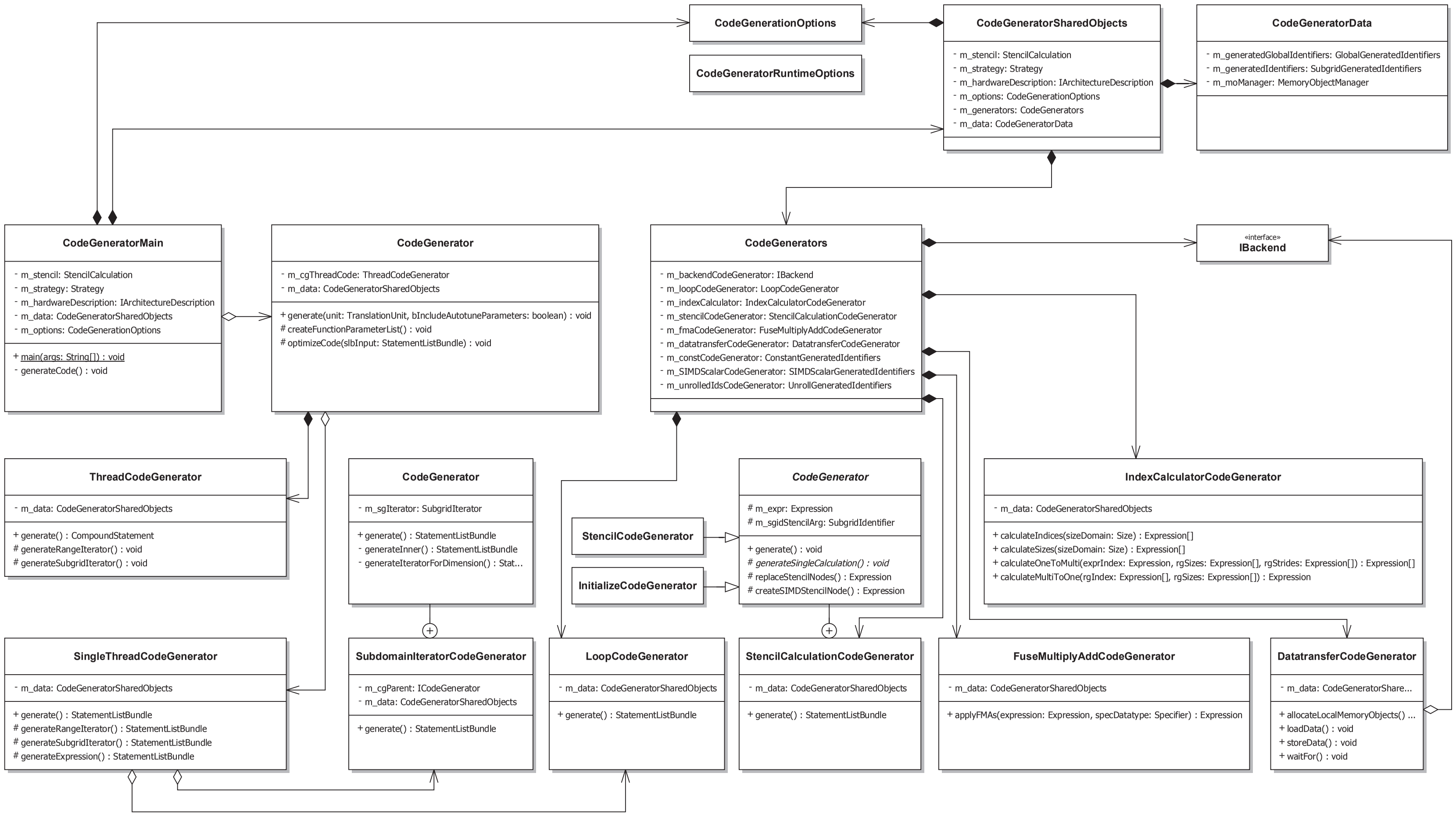


Figure 12.4: *UML class diagram of the code generator classes.*

`Generator` class, which has two specializations, `StencilCodeGenerator`, which creates the code used in the stencil kernel function, and `InitializeCodeGenerator`, which is responsible for the initialization of the grid arrays in the initialization function thus providing the NUMA-aware initialization. The code generator takes care of stencil node offsets due to loop unrolling and vectorization.

- The actual array indices are calculated by the `IndexCalculatorCodeGenerator`. In the generated code, the index calculations are placed just before the stencil evaluation, i.e., within the inner-most loop. We count on the compiler to perform loop-invariant code motion and pull the bulk of the index calculation out of the loop. All the compilers which have been tested (GNU 4.4, Intel icc 11.1, Microsoft cl 16) were able to perform this transformation. The details how the indices are calculated are given in Chapter 13.2.
- The `FuseMultiplyAddCodeGenerator` creates fused multiply adds for architectures which have a corresponding intrinsic by replacing multiply-add combinations in the internal representation, and
- The `DatatransferCodeGenerator` creates the code responsible for copying data to and from memory on other levels of the memory hierarchy, e.g., from global to shared memory on a GPU. The actual implementation depends on the architecture and is therefore provided by an instance of a back-end code generator `IBackend`.

The code generation options are controlled by the settings in `CodeGenerationOptions` and `CodeGeneratorRuntimeOptions`. The former contains the settings for the global options (e.g., whether or not to vectorize and in which way, how to unroll the inner-most loops, whether to create a function interface that is compatible with Fortran, etc.). The latter specifies the local options for the current code generation pass. For instance, if the user wants the code generator to unroll loops, there are most likely multiple unrolling configurations, and the local options specify the unrolling configuration which is currently generated. An instance of the `CodeGeneratorRuntimeOptions` class is passed to the `generate` methods of the code generators; due to space limits the arguments to the code generator methods were omitted in the UML class diagram in Fig. 12.4.

All the code generator objects receive an instance of `CodeGeneratorSharedObjects` when they are constructed. Such an object encapsulates the “common knowledge” of the code generators, in particular the stencil representation, the internal representation of the Strategy, and the hardware architecture description. Identifiers created during code generation are stored in `CodeGeneratorData`.

12.3 Code Generation Back-Ends

The code generation back-ends are responsible for providing the details to effect the parallelization in the respective programming model, for implementing data movement functions if needed or desired, for converting arithmetic operations and standard function calls into intrinsics and for mapping the standard floating point data types to native data types if necessary. For instance, if explicit vectorization is turned on and the vectorization is done using SSE, an addition is no longer written using the "+" operator, but using the `_mm_addpd` intrinsic function instead, and as arguments this function expects two `__m128d`s, so a `double` — or rather a SIMD vector of 2 `double`s — has to be converted to the SSE data type `__m128d`.

Also, the code generation back-ends provide implementations of functions not specific to the compute kernel: the benchmarking harness needs to allocate memory for the grids and free it again, or it needs to create metrics expressions for the stencil, such as the number of Flops per stencil evaluation or the expression computing the performance number.

The code generation back-ends consist of multiple interwoven parts. The Java back-end code generator is chosen by the architecture description. The `backend` attribute of the `codegenerator` tag (cf. Listing 12.1) defines the back-end ID for which the `BackendFactory` will return an instance of the corresponding back-end implementation.

Listing 12.1: *An excerpt of a* PATUS *architecture description.*

```
<?xml version="1.0"?>
<architectureTypes>
  <!-- Intel x86_64 architecture with SSE,
    OpenMP parallelization -->
  <architectureType name="Intel x86_64 SSE">
    <codegenerator backend="OpenMP" src-suffix="c" .../>
    ...
    <intrinsics>
      <!-- intrinsics for double precision arithmetic
        operations -->
      <intrinsic baseName="plus" name="_mm_add_pd"
        datatype="__m128d"/>
```

Listing 12.1: *An excerpt of a* PATUS *architecture description.* (cont.)

```
        ...
      </intrinsics>
      <includes>
        <include file="omp.h"/>
        <include file="xmmintrin.h"/>
        <include file="emmintrin.h"/>
      </includes>
      <build harness-template-dir="arch/CPU_OpenMP" .../>
    </architectureType>
  </architectureTypes>
```

Conversely, the Java back-end implementation (in the package `ch.unibas.cs.hpwc.patus.codegen.backend`; a UML class diagram is shown in Fig. 12.5) is backed by the architecture description, which provides back-end details such as a mapping between arithmetic operators and their replacing intrinsic functions, and a mapping between primitive data types and their native representation as shown in Listing 12.1. The `baseNames` in the `intrinsic` tags correspond to the methods of `IArithmetic`. The default implementation in the class `AbstractArithmeticImpl` converts unary and binary arithmetic operators to function calls with functions named as specified in the architecture description. If certain operators require special treatment (e.g., there is no SSE intrinsic to negate the SSE vector entries), the corresponding `IArithmetic` method can be overwritten in the concrete back-end implementation, which will then be used instead of the default.

Apart from providing architecture-specific bindings for use within the stencil kernel function, the back-end generator also provides mechanisms used to create the benchmarking harness. Details are described in Chapter 12.4.

The `IBackend` interface is composed of logically separated sub-interfaces:

- `IParallel` defines code generation methods related to parallelization, such as synchronization constructs.
- `IDataTransfer` provides code generation related to data movement between memories in the memory hierarchy.
- `IIndexing` defines how subdomain indices are calculated. In the CPU multicore realm, each thread has a one-dimensional ID. In CUDA, on the other hand, there are two levels of indices: thread

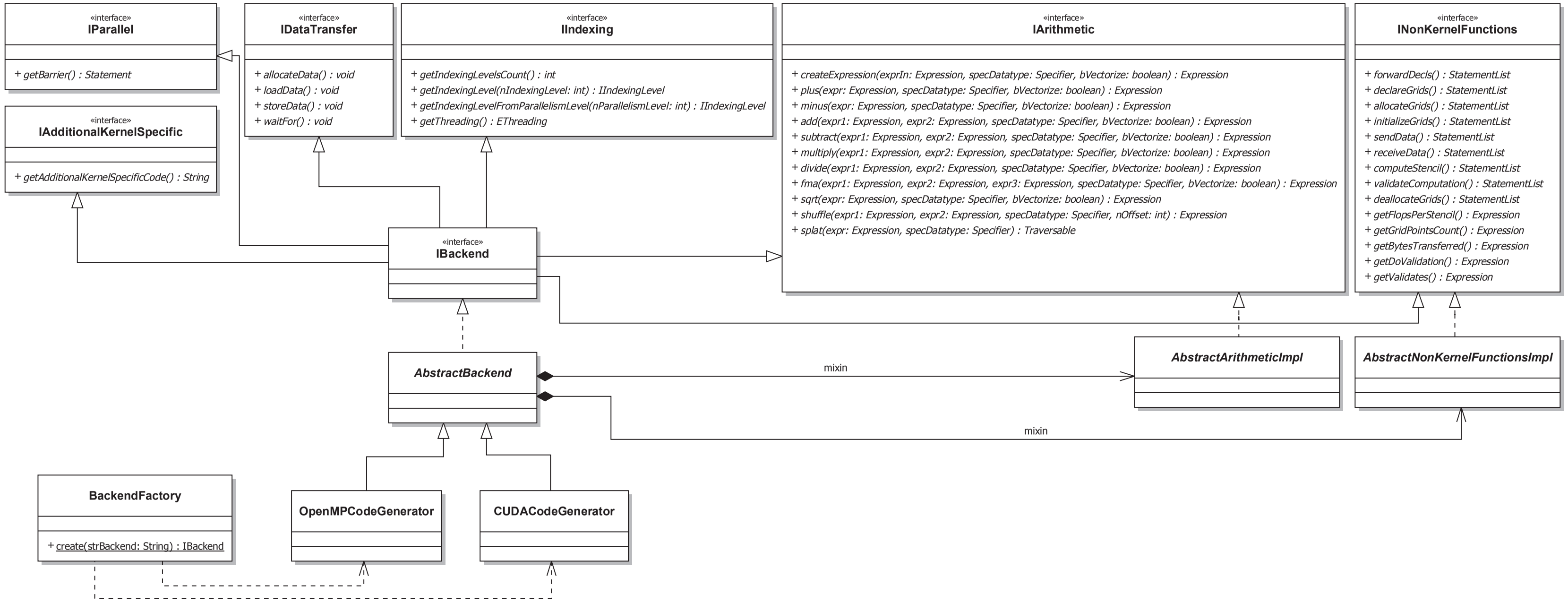


Figure 12.5: *UML class diagram of the code generator back-ends.*

and thread block indices, which can be one-, two-, or three-dimensional. `IIndexing` takes care of how to map these programming model-specific peculiarities.

- `IArithmetic` provides the mapping between arithmetic operators and their architecture-specific native nature as intrinsic as discussed above.
- `INonKernelFunctions` deals with the architecture-specific functionalities related to the benchmarking harness rather than the actual stencil kernel.

`AbstractBackend` provides a base for concrete back-end implementations and mixes in the methods of `AbstractArithmeticImpl` and `AbstractNonKernelFunctionsImpl`. Currently, an OpenMP back-end for shared memory CPU systems and a C for CUDA back-end for NVIDIA graphics processing units are implemented as concrete back-end classes.

12.4 Benchmarking Harness

The benchmarking harness is generated from an architecture-specific template consisting of a set of C (or C for CUDA) files and a Makefile within a directory. The architecture description file specifies the directory in which the code generator will look for the template files to process in the `harness-template-dir` attribute of the `build` tag in Listing 12.1. The code generator (`ch.unibas.cs.hpwc.patus.codegen.benchmark.BenchmarkHarness`) looks for `patus` pragmas and identifiers starting with `PATUS_` in the C files and replaces them with actual code, depending on the stencil specification.

A simple benchmarking harness template for OpenMP is shown in Listing 12.2.

Listing 12.2: *Sample OpenMP benchmarking harness template.*

```
#pragma patus forward_decls
int main (int argc, char** argv)
{
  // prepare grids
  #pragma patus declare_grids
  #pragma patus allocate_grids

```

Listing 12.2: *Sample OpenMP benchmarking harness template.* (cont.)

```
  // initialize
  #pragma omp parallel
  {
    #pragma patus initialize_grids
  }

  long nFlopsPerStencil = PATUS_FLOPS_PER_STENCIL;
  long nGridPointsCount = 5 * PATUS_GRID_POINTS_COUNT;
  long nBytesTransferred = 5 * PATUS_BYTES_TRANSFERRED;

  // run the benchmark
  tic ();
  #pragma omp parallel private(i)
  for (i = 0; i < 5; i++) {
    #pragma patus compute_stencil
    #pragma omp barrier
  }
  toc(nFlopsPerStencil,nGridPointsCount,nBytesTransferred);

  // free memory
  #pragma patus deallocate_grids
  return 0;
}
```

The pragmas create forward declarations for the functions contained in the kernel file, declare the grid arrays, allocate, initialize, and free them. The pragma `compute_stencil` is replaced by the call to the kernel function.

In the code generator, both the `patus` pragmas and the `PATUS_*` identifiers are mapped to `INonKernelFunctions` methods in the Java part, which are invoked while scanning the template files. The `INonKernelFunctions` interface defines the functions corresponding to the pragmas and identifiers shown in Listing 12.2, but should additional stencil- or Strategy-dependent code substitutions for the benchmarking harness be necessary for an architecture, methods named as the pragmas or identifiers can simply be added to the corresponding implementation of the back-end code generator (the implementation of `INonKernelFunctions`), which are called by reflection by the framework[*].

---

[*] The underscores in the pragmas and identifiers are removed and the names are converted to camel-case to meet Java conventions. Thus, e.g., the method in the

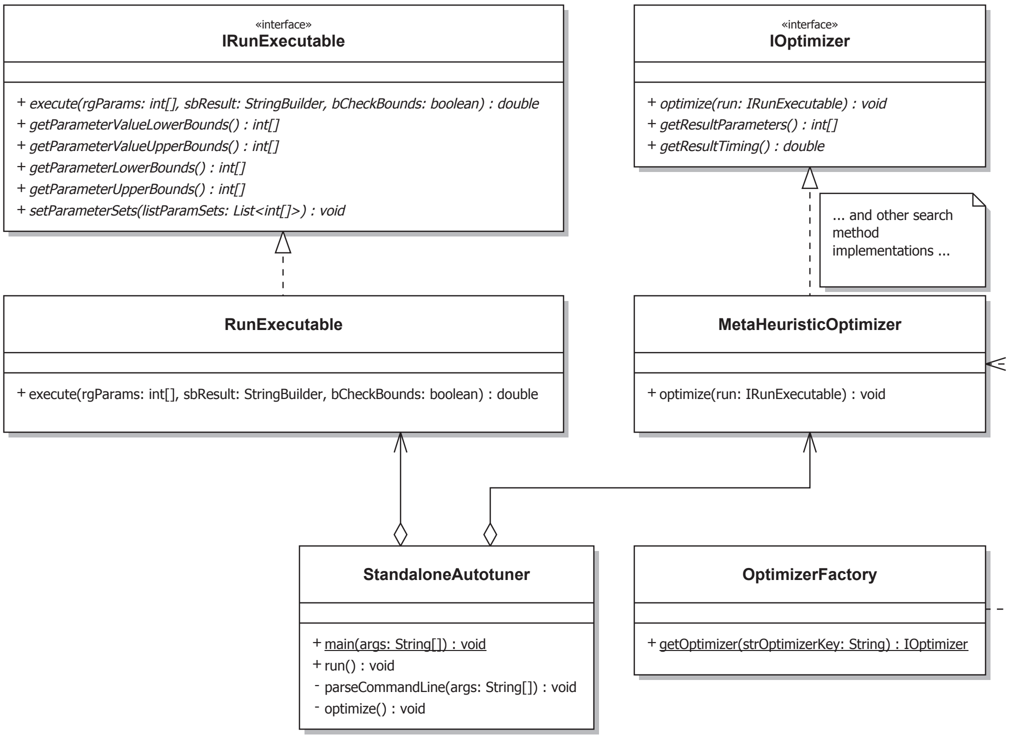


Figure 12.6: *UML class diagram of the auto-tuner subsystem.*

Optionally, the code generator can generate code validating the result of the generated kernel function against a naïve sequential version of the stencil. If validation is turned on, the grid declarations and initializations are duplicated to accommodate both the result computed by the optimized kernel and the reference solution. The validation code, which is inserted directly into the benchmarking harness source, computes the reference solution based on a naïve sequential Strategy, which is constructed internally. Then, it compares the values of the inner grid points of the result returned by the kernel function to the values of the reference grids. More precisely, the relative error is computed, and if it exceeds a certain predefined threshold, the computation implemented in the kernel function is assumed to be faulty, and the validation fails.

12.5 The Auto-Tuner

The implementation of the PATUS auto-tuner module is contained in the Java package `ch.unibas.cs.hpwc.patus.autotuner`. Architecturally, the main parts of the auto-tuner are two Java interfaces, `IOptimizer` and `IRunExecutable`, and the `OptimizerFactory`. The latter instantiates an object implementing the `IOptimizer` interface, i.e., the class implementing a search method. The optimizer is started by invoking `IOptimizer`'s only method, `optimize`, and passing it an instance of a class implementing `IRunExecutable`. The interface does the evaluation of the objective function, i.e., in the context of auto-tuning, runs the benchmarking executable. The default implementation simply starts the executable and reads its output, which must include the time measurement. The time measurement is taken as the value of the objective function for the given set of parameters (objective function arguments). Additional program output is saved (at least for the current best evaluation), which in case of the default benchmarking harness includes some metrics of the computation (number of grid points, Flops per stencil, total Flops), and the performance in GFlop/s.

The `RunExecutable` class, which is the default implementation for `I-RunExecutable`, separates the parameter space for the actual program execution from the space of decision variables, which the optimizer classes see. The actual parameter space for the program execution might be an arbitrary list of integers per parameter. The `StandaloneAutotuner`, which is instantiated when the PATUS auto-tuner is invoked from the command line, accepts parameter ranges that are parts of arithmetic or geometric sequences, or a list of arbitrary integers (cf. Appendix A.2). Each of these ranges is mapped onto a set of consecutive integers, which is what the search methods see.

`RunExecutable` caches execution times (assuming that the variation in execution time is negligible) and returns the cached value if the executable has been started with the same parameters in a previous run to save time. Also, before executing, constraints are checked, and if a constraint is violated a penalty value is returned. Constraints are comparison expressions containing references to the benchmark executable parameters (cf. Appendix A.2). At runtime, the concrete parameter values are substituted for the references, and Cetus's simplifier is used to

---

`IBackend` implementation corresponding to the `compute_stencil` pragma should be `computeStencil`.

check whether the comparison expression is satisfied.

Chapter 13

Generating Code: Instantiating Strategies

> Those who may have the patience to study a moderate quantity of rather dry details will find ample compensation…
>
> — Ada Lovelace (1815–1852)

13.1 Grids and Iterators

In Chapters 12.1.2 and 12.2 we discussed how strategies are represented internally in PATUS and how C code is generated using a Strategy as a template. In this Chapter, we want to give some more details how the stencil specification and the Strategy play together. In particular, the mechanism is presented, how a subdomain iterator together with a stencil node induces the notion of a *memory object*: a (possibly local) array which holds the data for computation or the result of the computation.

The notion of stencil nodes enables us to map grids, or parts of grids, to arrays which are stored physically in main memory or some local memory. In Chapter 6 we discussed how portions of the grids can be laid out in memory, e.g., in cache. It has been argued that it is not necessary to bring an entire cube-shaped subdomain into memory, but rather, e.g., for a 7-point stencil in 3D with stencil nodes $(0,0,-1;-1;\bullet)$, $(0,0,0;-1;\bullet)$, $(\pm 1,0,0;-1;\bullet)$, $(0,\pm 1,0;-1;\bullet)$, $(0,0,1;-1;\bullet)$, 3 *input planes* or data points are required to produce one *output plane*. Assume that the planes are orthogonal to the z-axis. Then, these 3 planes, P_{-1}, P_0, P_1, contain the stencil nodes

$$\begin{aligned}
P_{-1} &\ni \{(0,0,-1;-1;\bullet)\},\\
P_0 &\ni \{(0,0,0;-1;\bullet),(\pm 1,0,0;-1;\bullet),(0,\pm 1,0;-1;\bullet)\},\\
P_1 &\ni \{(0,0,1;-1;\bullet)\}.
\end{aligned}$$

Conversely, if we know that we want to cut the domain into planes, we can infer the planes required from the set of stencil nodes, and therefore the space in memory required and also the memory layout. We call a portion of a grid stored in an array in some memory a *memory object*. In general, we define a memory object as follows:

Definition 13.1. *Let $n \in \Sigma^\bullet$ be a stencil node and let $A : \mathbb{Z}^d \longrightarrow \mathbb{Z}^d$ be a projection. The* memory object *for n under A is the equivalence class $\Sigma^\bullet/A := \{A(\alpha(n)) : n \in \Sigma^\bullet\}$.*

Formally, we denote the d-dimensional subdomain iterator

```
for subdomain v(s) in u(:;t) ...
```

by a d-dimensional vector s defining the size of the subdomain `v` being iterated within its parent subdomain `u`, as $v = (s_1, \dots, s_d) \in \mathbb{N}^d$ with $s_1, \dots, s_d \geqslant 1$.

Definition 13.2. *Let $v = (s_1, \dots, s_d)$ be a subdomain iterator. The* projection mask *$\pi \in \mathbb{Z}^{d\times d}$ associated with v is the diagonal $d \times d$ matrix over $\mathbb{Z}$ defined by*

$$\pi_{ii} := \begin{cases} 1 & \text{if } s_i = 1, \\ 0 & \text{otherwise.} \end{cases}$$

In this way, every subdomain iterator gives rise to a projection, which, as per Definition 13.1, can be applied to a stencil node, and we obtain a memory object; in fact we obtain the memory object containing that stencil node when iterating over the parent domain. The idea behind Definition 13.2 is the following. In 3D, we assume that a subdomain iterator is either a plane (a slice of width 1, no matter how it is oriented, but orthogonal to some coordinate axis — if exactly one coordinate s_i is set to 1), a

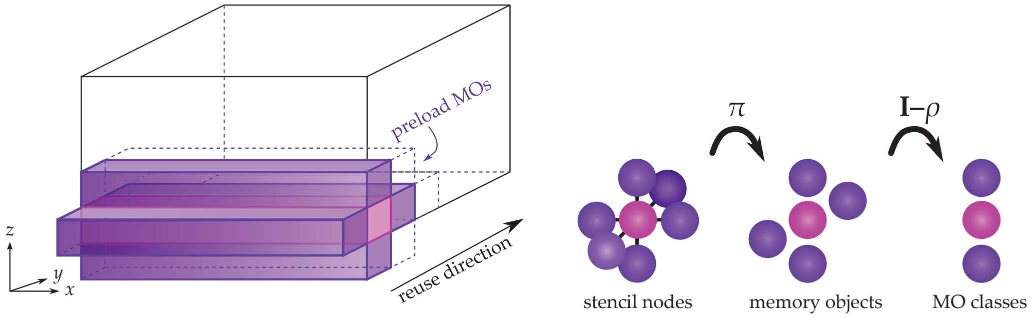


Figure 13.1: *Memory objects (MOs) and memory object classes obtained by projecting stencil nodes using the projections π and $\mathbf{I}-\rho$ induced by the subdomain iterator.*

line (a stick with exactly two coordinates set to 1), or a point (if all 3 coordinates are set to 1). All other configurations are assumed to be boxes which contain as least as many points such that a stencil can be evaluated using the points within the box.

Thus, if, e.g., $v = (s_1, s_2, 1)$ is a plane subdomain iterator (with plane orthogonal to the z-axis), the associated projection mask is the projection $\begin{pmatrix} 0 & & \\ & 0 & \\ & & 1 \end{pmatrix}$. Applying the projection to the stencil nodes of the 3D 7-point stencil mentioned above, we retrieve three memory objects, corresponding to the P_{-1}, P_0, P_1 from above. The four stencil nodes $(\pm 1, 0, 0; -1; \bullet)$, $(0, \pm 1, 0; -1; \bullet)$, being projected to $(0, 0, 0; -1; \bullet)$, are "masked out," hence the name of the projection.

Fig. 13.1 shows an example of a subdomain iterator $v = (s_1, 1, 1)$. The projection mask is $\pi = \begin{pmatrix} 0 & & \\ & 1 & \\ & & 1 \end{pmatrix}$, hence there are five memory objects (MOs) $(*, \pm 1, 0)$, $(*, 0, 0)$, $(*, 0, \pm 1)$ as shown in the figure. The stencil nodes $(0, 0, 0; -1; \bullet)$ and $(\pm 1, 0, 0; -1; \bullet)$ are collapsed into the single memory object $(*, 0, 0)$. (Although the memory object space $\Sigma^\bullet/\pi$ is 2-dimensional, we keep the coordinate which was projected to zero in the notation for clarity and mark it by an asterisk.)

The way subdomain iterators are constructed, we can identify the *reuse direction*, the direction in which data from memory objects into which data was loaded previously, can be reused. The reuse direction is the first direction with a unit step progression. In Fig. 13.1, the x-direction is fully spanned by the subdomain iterator, and the first unit step direction is along the y-axis. Hence, in this example, the reuse direction is the y-direction. This motivates the definition of the reuse mask:

Definition 13.3. *The* reuse mask $\rho \in \mathbb{Z}^{d \times d}$ *for the subdomain iterator v is the* *diagonal $d \times d$ matrix over $\mathbb{Z}$ defined by*

$$\rho_{ii} := \begin{cases} 1 & \text{if } \pi_{ii} = 1 \text{ and } \pi_{jj} = 0 \text{ for } j < i, \\ 0 & \text{otherwise,} \end{cases}$$

i.e., ρ has exactly one non-zero entry at the first non-zero position in π.

Applying the projection $(\mathbf{I} - \rho) \circ \pi$ to the set of stencil nodes $\Sigma^{\bullet}$ gives rise to memory object equivalence classes

$$\Sigma^{\bullet} / \left((\mathbf{I} - \rho) \circ \pi \right)$$

within which memory objects can be reused cyclically. $\mathbf{I}$ is the identity matrix.

The structure labeled "MO classes" in Fig. 13.1 show the three classes into which memory objects are divided, $(*, *, \pm 1)$ and $(*, *, 0)$ after applying the projection $\mathbf{I} - \rho = \begin{pmatrix} 1 & & \\ & 1 & \\ & & 1 \end{pmatrix} - \begin{pmatrix} 0 & & \\ & 1 & \\ & & 0 \end{pmatrix} = \begin{pmatrix} 1 & & \\ & 0 & \\ & & 1 \end{pmatrix}$ to the memory objects $(*, \pm 1, 0)$, $(*, 0, 0)$, $(*, 0, \pm 1)$. The lower and upper classes $(*, *, \pm 1)$ symbolized with a purple bullet contain one memory object each, which means that there is no data reuse from the respective memory objects when the subdomain iterator is advanced by one step (which is in y-direction). The middle class symbolized by the pink bullet, however, contains three memory objects $(*, 0, 0)$, $(*, \pm 1, 0)$. When the iterator progresses one step, two of the memory objects within the class can be reused, namely $(*, 0, 0)$ and $(*, 1, 0)$, and only one, the one with the oldest data, has to be loaded newly. Of course, the same theory also applies to other subdomain iterator shapes.

For overlapping computation and data transfers, the reuse direction gives rise to *preload memory objects*, memory objects which are pre-loaded with data before they are consumed in the next iteration step. In Fig. 13.1 they are drawn in dotted lines and labeled accordingly. The set of preload memory objects can be found by advancing the spatial coordinates of the stencil nodes by 1 in the reuse direction and subtracting the original set from the union:

$$\text{Preload} = \Big(\left(\{ n + (\rho \mathbb{1}; 0; 0) : n \in \Sigma^{\bullet} \} \cup \Sigma^{\bullet} \right) \setminus \Sigma^{\bullet} \Big) \Big/ \pi.$$

Here, $\mathbb{1}$ is the vector $(1, \ldots, 1)^{\intercal} \in \mathbb{Z}^{d}$.

13.2 Index Calculations

Another concern in the code generator is calculating indices into subdomains and into memory objects. We declare all the arrays used for the computation as one-dimensional arrays, thus a the d-dimensional indexing in the stencil specification and the Strategy has to be converted to one linear index. Internally, subdomains keep their d-dimensional indexing, but when assigning subdomains to parallel entities (threads or thread blocks), the hardware- or programming model-specific indexing has to be converted to the d-dimensional indexing of the subdomains. In OpenMP we have one level of one-dimensional indexing: each thread is assigned a linear index. In CUDA, however, there are two levels of indexing, one for the thread blocks and one for the threads. Moreover, thread block indices are natively two-dimensional prior to version 4 of CUDA (but the programmer could discard the second dimension), and threads are indexed by 1, 2, or 3-dimensional indices. Hence, a two-level $(2, 3)$-dimensional index has ultimately to be converted to a d-dimensional index. In CUDA 4.0 the grid dimensionality was extended to 3 dimensions for Fermi GPUs, so the source is a $(3, 3)$-dimensional index. In this chapter, a general framework for converting between multi-level multi-dimensional indices is established.

A linearized index i is calculated from a d-dimensional index $(i_1, \ldots, i_d)$ in a domain sized $(N_1, \ldots, N_d)$ by means of the formula

$$i = i_1 + n_1(i_2 + n_2(\cdots(i_{d-1} + n_{d-1}i_d)\cdots)). \tag{13.1}$$

We do not need multi-level indices to calculate the linear memory object index, since we always know the global coordinates.

However, we do need multi-level indices to compute a subdomain target index from a hardware index, such as a thread ID. Let D be the dimensionality of the target index (e.g., 3 for 3-dimensional stencil computations). Assume there are L indexing levels. Let $d^{(\ell)}$ be the minimum of the index dimensionality of the indexing level ℓ, $\ell = 1, 2, \ldots, L$, and the target dimensionality D. By taking the minimum, excess hardware indices are simply ignored.

Example 13.1: *NVIDIA GPUs*

On a pre-Fermi NVIDIA GPU with two indexing levels, thread blocks and the grid of blocks, threads within a block can be addressed

Example 13.1: *NVIDIA GPUs* (cont.)

with 3-dimensional indices, and the blocks in a grid can be addressed with 2-dimensional indices. So for this example, $L = 2$ and $d^{(1)} = 3, d^{(2)} = 2$.

By $\left(i_1^{(\ell)}, i_2^{(\ell)}, \ldots, i_{d^{(\ell)}}^{(\ell)}\right) \in \mathbb{N}^{d^{(\ell)}}$ we denote an actual index for indexing level ℓ, and by $\left(n_1^{(\ell)}, n_2^{(\ell)}, \ldots, n_{d^{(\ell)}}^{(\ell)}\right)$ the block size, i.e., the bounds for the indices, $0 \leqslant i_j^{(\ell)} < n_j^{(\ell)}$, for $j = 1, 2, \ldots, d^{(\ell)}$. Also, denote the full domain size by $(N_1, N_2, \ldots, N_D)$.

We create the D-dimensional target index from the hierarchical source indices, whose dimensions are dictated by the hardware, in two steps. First, we expand the source indices $\left(i_1^{(\ell)}, i_2^{(\ell)}, \ldots, i_{d^{(\ell)}}^{(j)}\right)$, $\ell = 1, \ldots, L$, to indices of dimension D. For the source indices where $\ell < L$, we append zeros so that the index has the desired dimensionality. For the last indexing level ($\ell = L$), we emulate the possibly missing dimensions by extrapolating the last dimension. The block sizes are expanded similarly. Secondly, we calculate the total global index from the expanded indices.

In detail, for all indexing levels $\ell = 1, \ldots, L-1$, but the last, we extend the dimensionalities to the target dimensionality D of the indices by defining

$$\tilde{i}_j^{\ell} := \begin{cases} i_j^{(\ell)} & \text{if } 1 \leqslant j \leqslant d^{(\ell)}, \\ 0 & \text{if } d^{(\ell)} + 1 \leqslant j \leqslant D, \end{cases}$$

$$\tilde{n}_j^{(\ell)} := \begin{cases} n_j^{(\ell)} & \text{if } 1 \leqslant j \leqslant d^{(\ell)}, \\ 1 & \text{if } d^{(\ell)} + 1 \leqslant j \leqslant D. \end{cases}$$

We assume that the block sizes $\left(n_1^{(\ell)}, n_2^{(\ell)}, \ldots, n_{d^{(\ell)}}^{(\ell)}\right)$ are given (dictated by the hardware) for $\ell = 1, \ldots, L-1$. For the block sizes in the last indexing level L, we calculate how many level $L-1$ blocks are needed in each dimension by

$$\begin{aligned} \tilde{n}_j^{(L)} &= \left\lceil N_j \Big/ \prod_{\ell=1}^{L-1} \tilde{n}_j^{(\ell)} \right\rceil \qquad \text{for } j = 1, \ldots, d^{(L)} - 1 \\ \tilde{n}_{d^{(L)}}^{(L)} &= \prod_{k=d^{(L)}}^{D} \left\lceil N_k \Big/ \prod_{\ell=1}^{L-1} \tilde{n}_k^{(\ell)} \right\rceil . \end{aligned} \tag{13.2}$$

Up to the dimensionality minus 1 of the last indexing level, the number of $(L-1)$-level blocks is just the domain size divided by the size of the $(L-1)$-level block. The size of an $(L-1)$-level block is the product over all levels of the number of blocks. In the last dimension we emulate the dimensions that are possibly missing in the last indexing level, hence the product over the last $(d^{(L)})$ and possibly missing dimensions $(d^{(L)}, \ldots, D)$.

The index $\left(\tilde{i}_1^{(L)}, \tilde{i}_2^{(L)}, \ldots, \tilde{i}_D^{(L)}\right)$ on the last indexing level is calculated for $j = D, D-1, \ldots, d^{(L)}+1, d^{(L)}$ in decrementing order, by the recursion

$$\tilde{i}_j^{(L)} = \left\lfloor \left(i_{d^{(L)}}^{(L)} - \sum_{k=j+1}^{D} \tilde{i}_k^{(L)} \sigma_{k-1} \right) \Big/ \sigma_{j-1} \right\rfloor, \tag{13.3}$$

with strides (i.e., the total number of level L blocks in the grid in the emulated directions $\leqslant j$)

$$\sigma_j := \begin{cases} 1 & \text{for } 0 \leqslant j \leqslant d^{(L)} - 1, \\ \sigma_{j-1} \cdot \left\lceil N_j \Big/ \prod_{\ell=1}^{L-1} \tilde{n}_j^{(\ell)} \right\rceil & \text{for } d^{(L)} \leqslant j \leqslant D \end{cases} \tag{13.4}$$

For $1 \leqslant j \leqslant d^{(L)} - 1$, we set $\tilde{i}_j^{(L)} = i_j^{(L)}$.

Note that the factor in the definition of σ_j corresponds to the number of blocks being multiplied together in Eqn. 13.2. Also note that Eqn. 13.3 solves the equation

$$i_{d^{(L)}}^{(L)} = \sum_{k=d^{(L)}}^{D} \tilde{i}_k^{(L)} \sigma_{k-1}$$

iteratively for the unknowns $\tilde{i}_{d^{(L)}}^{(L)}, \tilde{i}_{d^{(L)}+1}^{(L)}, \ldots \tilde{i}_{D-1}^{(L)}, \tilde{i}_D^{(L)}$. This equation corresponds to the calculation of a linearized index (Eqn. 13.1), with the block sizes $n_\bullet$ written as strides $\sigma_\bullet$. (Indeed, set $n_j = \left\lceil N_j \big/ \prod_{\ell=1}^{L-1} \tilde{n}_j^{(\ell)} \right\rceil$ and cf. Eqn. 13.4.) In this case, however, the linearized index (the left hand side) is given, and the coordinates of the multi-dimensional index are sought.

Now, the total global index $\left(\hat{i}_1, \hat{i}_2, \ldots, \hat{i}_D\right)$ is calculated using the expanded indices and block sizes by

$$\hat{i}_j := \tilde{i}_j^{(1)} + \tilde{n}_j^{(1)} \left(\tilde{i}_j^{(2)} + \tilde{n}_j^{(2)} \left(\cdots \left(\tilde{i}_j^{(L-1)} + \tilde{n}_j^{(L-1)} \cdot \tilde{i}_j^{(L)} \right) \cdots \right) \right) \quad (j = 1, \ldots, D).$$

Example 13.2: *Calculating a 3D index from a 1D OpenMP thread ID.*

The target dimensionality is $D = 3$, and we have one indexing level, hence $L = 1$. Also, $d^{(1)} = \min(1,3) = 1$. $i_1^{(1)} = \texttt{thdid}$ and $n_1^{(1)} = \texttt{numthds}$.

$$\tilde{n}_1^{(1)} = \prod_{k=1}^{3} \lceil N_k/1 \rceil = N_1 N_2 N_3$$

$$\sigma_0 = 1, \quad \sigma_1 = \sigma_0 \lceil N_1/1 \rceil = N_1, \quad \sigma_2 = \sigma_1 \lceil N_2/1 \rceil = N_1 N_2$$

$$z = \tilde{i}_3^{(1)} = \left\lfloor \left(i_1^{(1)} - 0 \right) \Big/ \sigma_2 \right\rfloor = \lfloor \texttt{thdid}/\sigma_2 \rfloor$$

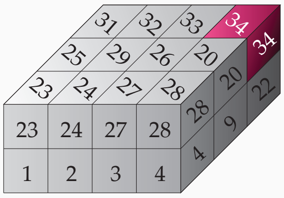
$$y = \tilde{i}_2^{(1)} = \left\lfloor \left(i_1^{(1)} - \tilde{i}_3^{(1)} \sigma_2 \right) \Big/ \sigma_1 \right\rfloor = \left\lfloor \texttt{thdid} - \tilde{i}_3^{(1)} \sigma_2 \Big/ N_1 \right\rfloor$$

$$x = \tilde{i}_1^{(1)} = \left\lfloor \left(i_1^{(1)} - \tilde{i}_2^{(1)} \sigma_1 - \tilde{i}_3^{(1)} \sigma_2 \right) \Big/ \sigma_0 \right\rfloor = \texttt{thdid} - \tilde{i}_2^{(1)} \sigma_1 \tilde{i}_3^{(1)} \sigma_2.$$

Example 13.3: *Some concrete numbers...*

In Example 13.2, set $N_1 = 4$, $N_2 = 3$, $N_3 = 2$. Then, $\sigma_0 = 1$, $\sigma_1 = N_1 = 4$, $\sigma_2 = N_1 N_2 = 12$. In the figure we can see that the thread with ID 23 is mapped to the coordinates $(3, 2, 1)$. Indeed, from the formulae in Example 13.2 we have

$$z = \lfloor 23/12 \rfloor = 1$$
$$y = \lfloor (23 - 1 \cdot 12)/4 \rfloor = 2$$
$$x = 23 - 1 \cdot 12 - 2 \cdot 4 = 3.$$



Example 13.4: *1D indexing on a CUDA GPU for blocks and grids. 3D target.*

Here, again $D = 3$ and $L = 2$, but we set $d^{(1)} = d^{(2)} = 1$.
Level 1: $i_1^{(1)} = \texttt{threadIdx.x}$, $n_1^{(1)} = \texttt{blockDim.x}$.
Level 2: $i_1^{(2)} = \texttt{blockIdx.x}$, $n_1^{(2)} = \texttt{gridDim.x}$.

Example 13.4: *1D indexing on a CUDA GPU for blocks and grids. 3D target.* (cont.)

To compute the global index (x, y, z), we calculate

$$\begin{aligned}\sigma_0 &= 1,\\ \sigma_1 &= \sigma_0 \cdot \lceil N_1/\texttt{blockDim.x} \rceil,\\ \sigma_2 &= \sigma_1 \cdot \lceil N_2/1 \rceil = \sigma_1 N_2\end{aligned}$$

$$\begin{aligned}\tilde{i}_3^{(2)} &= \left\lfloor i_1^{(2)} \middle/ \sigma_2 \right\rfloor = \lfloor \texttt{blockIdx.x}/\sigma_2 \rfloor\\ \tilde{i}_2^{(2)} &= \left\lfloor \left(i_2^{(2)} - \tilde{i}_3^{(2)} \cdot \sigma_2 \right) \middle/ \sigma_1 \right\rfloor\\ \tilde{i}_1^{(2)} &= i_2^{(2)} - \tilde{i}_2^{(2)} \cdot \sigma_1 - \tilde{i}_3^{(2)} \cdot \sigma_2\end{aligned}$$

$$\begin{aligned}x &= \texttt{threadIdx.x} + \texttt{blockDim.x} \cdot \tilde{i}_1^{(2)}\\ y &= 0 + 1 \cdot \tilde{i}_2^{(2)} = \tilde{i}_2^{(2)}\\ z &= 0 + 1 \cdot \tilde{i}_3^{(2)} = \tilde{i}_3^{(2)}\end{aligned}$$

Example 13.5: *CUDA GPU with 3D thread blocks and 2D grids. 3D target.*

The target dimensionality is specified to be 3, so $D = 3$. We have two levels of indices, threads and blocks, hence $L = 2$. Furthermore, we have $d^{(1)} = \min(3, 3) = 2$ and $d^{(2)} = \min(2, 3) = 2$.

Level 1:
$\left(i_1^{(1)}, i_2^{(1)}, i_3^{(1)}\right) = (\texttt{threadIdx.x}, \texttt{threadIdx.y}, \texttt{threadIdx.z})$,
$\left(n_1^{(1)}, n_2^{(1)}, n_3^{(1)}\right) = (\texttt{blockDim.x}, \texttt{blockDim.y}, \texttt{blockDim.z})$.

Level 2:
$\left(i_1^{(2)}, i_2^{(2)}\right) = (\texttt{blockIdx.x}, \texttt{blockIdx.y})$,
$\left(n_1^{(2)}, n_2^{(2)}\right) = (\texttt{gridDim.x}, \texttt{gridDim.y})$.

The global index (x, y, z) is calculated as follows:

$$\begin{aligned}\sigma_1 &= 1\\ \sigma_2 &= \sigma_1 \cdot \lceil N_2/\texttt{blockDim.y} \rceil\end{aligned}$$

Example 13.5: *CUDA GPU with 3D thread blocks and 2D grids. 3D target.* (cont.)

$$\begin{aligned}
\tilde{i}_3^{(2)} &= \left\lfloor i_2^{(2)} \middle/ \sigma_2 \right\rfloor = \lfloor \texttt{blockIdx.y}/\sigma_2 \rfloor \\
\tilde{i}_2^{(2)} &= \left\lfloor \left(i_2^{(2)} - \tilde{i}_3^{(2)} \cdot \sigma_2 \right) \middle/ \sigma_1 \right\rfloor = i_2^{(2)} - \tilde{i}_3^{(2)} \cdot \sigma_2
\end{aligned}$$

$$\begin{aligned}
x &= \texttt{threadIdx.x} + \texttt{blockDim.x} \cdot \texttt{blockIdx.x} \\
y &= \texttt{threadIdx.y} + \texttt{blockDim.y} \cdot \tilde{i}_2^{(2)} \\
z &= \texttt{threadIdx.z} + \texttt{blockDim.z} \cdot \tilde{i}_3^{(2)}
\end{aligned}$$

Chapter 14

Internal Code Optimizations

> Still it is a very important case as regards the engine, and suggests ideas peculiar to itself, which we should regret to pass wholly without allusion. Nothing could be more interesting than to follow out, in every detail, the solution by the engine of such a case as the above; but the time, space and labour this would necessitate, could only suit a very extensive work.
>
> — Ada Lovelace (1815–1852)

In this chapter, code transformations are described which are *not* controlled by Strategies. They are complementary optimizations, which the user can control on the PATUS command line when the code is generated. Currently, these code optimizations comprise loop unrolling and (explicit) vectorization.

In contrast to the optimization concepts provided by Strategies, internal code optimizations are not parameterizable in the same way as, e.g., cache block sizes. Parameters defined in a Strategy can be incorporated symbolically into the generated code and do not need to be known at code generation time, whereas the internal code optimizations alter the structure of the generated code. Hence parameters to internal code optimizations need to be known at code generation time. The optimizations described here generate multiple code variants, from which one is selected when the program is run. In the file containing the generated kernels there will indeed by one function for each loop unrolling variant, provided that different loop unrolling configurations are to be generated, along with one *master function* selecting one of the variants based on a parameter.

14.1 Loop Unrolling

Assuming that the stencil on the grid points can be evaluated in any order (e.g., using the Jacobi iteration, the loops sweeping over the spatial domain are fully permutable), we can do loop unrolling not only in one, the inner-most, loop, but also unroll the enclosing loops, while collapsing the inner loops and pull the loop bodies, the actual computation, into the inner-most loop in an unroll-and-jam [29] kind of way. Hence, all d loops sweeping over a d-dimensional block of data are unrolled simultaneously, thus providing another level of blocking. This transformation is also referred to as *register blocking* [52]. The idea is illustrated in Listing 14.1 for $d = 2$ and unrolling factors of 2 for each of the loops in the nest.

Listing 14.1: *Unrolling a loop nest.*

```
// original loop nest
for i = imin .. imax
  for j = jmin .. jmax
    S(i,j)

// unrolled loop nest
for i = imin .. imax by 2
  for j = jmin .. jmax by 2
      S(i,j)
      S(i,j+1)
      S(i+1,j)
      S(i+1,j+1)
```

Special treatment is required when the number of iterations is not divisible by the unrolling factor. In that case, cleanup loops have to be added which handle the remaining iterations. Algorithm 14.1 shows the method PATUS uses to unroll loop nests. Creating the unrolled and the cleanup loops is done recursively. Unrolling is only applied to the innermost subdomain iterator of a Strategy containing a stencil call.

Algorithm 14.1: *Creating an unrolled loop nest in* PATUS.

```
function UNROLL($it$, $config$, $dim$, $unroll$)
    $AST \leftarrow \{\}$
    if $dim > 0$ then                                              ▷ create unrolled loop
        $loop^{\text{unrolled}} \leftarrow$ createLoopHead ($it$, $dim$, $unroll_{dim}$)
        $loop^{\text{unrolled}}$.setBody (unroll ($it$, $config \cup \{unroll_{dim}\}$, $dim - 1$))
        $AST$.add ($loop^{\text{unrolled}}$)
        if $unroll_{dim} > 1$ then                                 ▷ create cleanup loop
            $loop^{\text{cleanup}} \leftarrow$ createLoopHead ($it$, $dim$, 1)
            $loop^{\text{cleanup}}$.setBody (unroll ($it$, $config \cup \{1\}$, $dim - 1$))
            $AST$.add ($loop^{\text{cleanup}}$)
        end if
    else
        $AST$.add (generateUnrolledLoopBody ($it$, $config$))
    end if
    return $AST$
end function
```

The function UNROLL expects three parameters, the subdomain iterator *it*, which we want to unroll, a configuration *config*, which encodes which loops being unrolled in the current state of the recursion, and for which loops cleanup loops are being generated. The parameter *dim* specifies which dimension of the subdomain iterator *it* is to be processed, and *unroll* is the desired unrolling configuration for the loop nest. For instance, if *it* is a 3-dimensional subdomain iterator and we want to unroll twice in the first dimension and four times in the second and third, *unroll* would be the vector $(2, 4, 4)$. The initial call to the function is

$$\text{UNROLL}(it, \{\}, d, \textit{desired unrolling configuration}),$$

where *d* is the dimensionality of the block the subdomain iterator *it* iterates through, and an empty set is passed as configuration. An empty local AST is created, to which the loop head of the unrolled loop and the cleanup loop are added, and which is returned by the function. The bodies of the unrolled and cleanup loops are the recursively generated ASTs by UNROLL with *dim* decremented, or the actual computation if *dim* has reached 0 at the leafs of the recursion tree.

The listing in Example 14.1 shows the code generated by UNROLL for a 3-dimensional subdomain iterator v with $unroll = (4, 4, 4)$. Every `for` loop in which the loop index is incremented by 4 is therefore an unrolled loop, all the loops with one increments are cleanup loops. Note that in the unrolled loops the original upper loop bounds were reduced by 3 (which is $unroll_{dim} - 1$) so that the upper bounds are not overshot, and that the initialization of the loop index in the cleanup loops is omitted so that the cleanup loops pick up where the unrolled loops left off.

Example 14.1: *Loop nests produced by unrolling.*

This listing shows an inner-most loop nest over points p = (`p_idx_x`, `p_idx_y`, `p_idx_z`) in a 3D subdomain v = (`v_idx_x`, `v_idx_y`, `v_idx_z`):

```
for (p_idx_z = v_idx_z; p_idx_z < v_idx_z_max - 3;
  p_idx_z += 4)
{
  for (p_idx_y = v_idx_y; p_idx_y < v_idx_y_max - 3;
    p_idx_y += 4)
  {
    for (p_idx_x = v_idx_x; p_idx_x < v_idx_x_max - 3;
      p_idx_x += 4)
    {
      // (4, 4, 4)-unrolled stencil code
    }
    for ( ; p_idx_x < v_idx_x_max; p_idx_x += 1)
      // (1, 4, 4)-unrolled stencil code
  }
  for ( ; p_idx_y < v_idx_y_max; p_idx_y += 1) {
    for (p_idx_x = v_idx_x; p_idx_x < v_idx_x_max - 3;
      p_idx_x += 4)
    {
      // (4, 1, 4)-unrolled stencil code
    }
    for ( ; p_idx_x < v_idx_x_max; p_idx_x += 1)
      // (1, 1, 4)-unrolled stencil code
  }
}
for ( ; p_idx_z < v_idx_z_max; p_idx_z += 1) {
  for (p_idx_y=v_idx_y; p_idx_y < v_idx_y_max - 3;
    p_idx_y += 4)
  {
    for (p_idx_x = v_idx_x; p_idx_x < v_idx_x_max - 3;
      p_idx_x += 4)
```

Example 14.1: *Loop nests produced by unrolling.* (cont.)

```
    {
      // (4, 4, 1)-unrolled stencil code
    }
    for ( ; p_idx_x < v_idx_x_max; p_idx_x += 1)
      // (1, 4, 1)-unrolled stencil code
  }
  for ( ; p_idx_y<v_idx_y_max; p_idx_y += 1)  {
    for (p_idx_x = v_idx_x; p_idx_x < v_idx_x_max - 3;
      p_idx_x += 4)
    {
      // (4, 1, 1)-unrolled stencil code
    }
    for ( ; p_idx_x < v_idx_x_max; p_idx_x += 1)
      // (1, 1, 1)-unrolled stencil code
  }
}
```

14.2 Dealing With Multiple Code Variants

Having code generation options which create a multitude of code variants is managed in the internal representation through statement list *bundles*: each element of the bundle corresponds to one configuration in the product space of the code generation options. This allows us to do only one code generation pass through the Strategy tree instead of as many as there are code variants. Adding a statement to a bundle means adding the statement to all the statement lists within the bundle. Adding a statement specific to one configuration of a code generation option means adding that statement to only the elements in the bundle with the matching configuration of that code generation option.

Suppose there are two code generation options generating multiple code variants, loop unrolling and software prefetching, for which a number of different implementations are provided*. Then, the stencil call within the inner-most spatial loop affects loop unrolling, but not software prefetching. For each loop unrolling configuration, the same unrolled statements are added to each of the statement lists within the bundle, which are tagged with the respective loop unrolling configuration,

---

*This example is just for illustration; currently no software prefetching has been implemented in Patus.

but regardless of the configuration of the lists software prefetching option tag, i.e., it is added to as many statement lists as there are software prefetching configurations.

14.3 Vectorization

Stencil computations naturally exhibit a large amount of data parallelism: the same arithmetic operations are performed on each point of the input grids (again assuming that the spatial loops are fully permutable). In particular, if we are free to traverse the grid in any order, we may choose to group consecutive elements in unit stride direction, and the computation becomes highly vectorizable. (In a red-black Gauss-Seidel scheme which computes only every other element within a sweep, vectorization obviously becomes less straightforward, though.)

For instance, when vectorized, the scalar assignment statement of a 1D 3-point stencil

$$u[x;t+1] \leftarrow u[x-1;t] + u[x;t] + u[x+1;t]$$

becomes

$$u[x:x+3;t+1] \leftarrow u[x-1:x+2;t] + u[x:x+3;t] + u[x+1:x+4;t]$$

if the length of a SIMD vector is 4 as is the case for the SSE single precision floating point data type. Unlike the general Fortran vector notation, when making use of low-level optimizations such as explicitly generating SSE code, the length of the vectors is dictated by the hardware and fixed.

In practice, there are at least two issues which make vectorization more complicated than illustrated above. Firstly, the number of grid points to compute may not be a multiple of the SIMD vector length. Thus, the excess grid points need to be treated specially in an epilogue loop, which does the computation in a scalar fashion. Secondly, the hardware may impose alignment restrictions. In SSE, if a vector read or written is not aligned at 16 Byte boundaries, which is the width of SSE vectors, there is a performance penalty. In fact, there are two specialized SSE instructions, `movaps`, which loads only aligned single precision data into an SSE register, and `movups`, which can also load unaligned data. `movaps` is fast, but the processor throws an exception if the alignment restriction is violated. `movups` has several cycles more latency, but it works in any case.

On other hardware, such as the Cell Broadband Engine Architecture, unaligned moves are not possible, i.e., if not correctly aligned data is tried to be brought into a register, always an exception will be thrown.

In the above example, if $x \equiv 0 \mod 4$, $u[x : x + 3; t]$ is a SIMD vector aligned at vector boundaries, but both $u[x - 1 : x + 2; t]$ and $u[x + 1 : x + 4; t]$ are not. These vectors could be loaded with a slower unaligned move instruction. *Shuffle* or *select* instructions, respectively, are another way around this: using such instructions, a new vector can be created from two input operands and a mask that specifies which bits of the two inputs are copied to which bits of the output. PATUS uses this approach and expects a recipe for using shuffle intrinsics in the code generation back-end for the respective architecture. For mixed-type computations for data types with SIMD vector lengths $\lambda_1, \dots, \lambda_k$, we require that the alignment restriction requirements be replaced by $\Lambda := \operatorname{lcm}(\lambda_1, \dots, \lambda_k)$.

In multiple dimensions, as the data is stored linearly in memory, it makes sense to do the vectorization only in the unit stride dimension. Alignment considerations, however, apply to any dimension: in 3D, assuming that x is the unit stride dimension and that z is the dimension varying slowest in memory, whether or not $u[x, y + \delta, z; t]$ is aligned at SIMD vector boundaries, does not depend on δ alone, but on the number of in-memory grid points in the x direction, X. In particular, if X is divisible by the SIMD vector length, then $u[x, y + \delta, z; t]$ is aligned for any $\delta \in \mathbb{Z}$. Note that in that case $u[x, y, z + \delta; t]$ is aligned as well. Therefore, PATUS assumes that X is a multiple of the SIMD vector length, thus shuffle instructions or unaligned moves are not necessary in the y and z dimensions. This requirement is not as restrictive as it may seem; it does not require that the number of *computed* grid points in the unit stride dimension is a multiple of the SIMD vector length, only that the number of elements in memory meets the condition. This can be achieved by padding the grid array in the unit stride direction[†]. Yet, compilers have to err on the conservative side and cannot make such assumptions. Even when a simple stencil code is automatically vectorized by a compiler, the performance benefit is not drastic. In fact, Intel's C compiler, `icc` v11.1, refuses to vectorize the code when compiled with the `-O3` optimizing option. It does vectorize when optimization is set to `-fast`, but the performance gain is negligible.

[†] The benchmarking harness does not do the padding automatically. If SIMD code is generated and the condition is not met, a warning is displayed, and the benchmarking executable terminates.

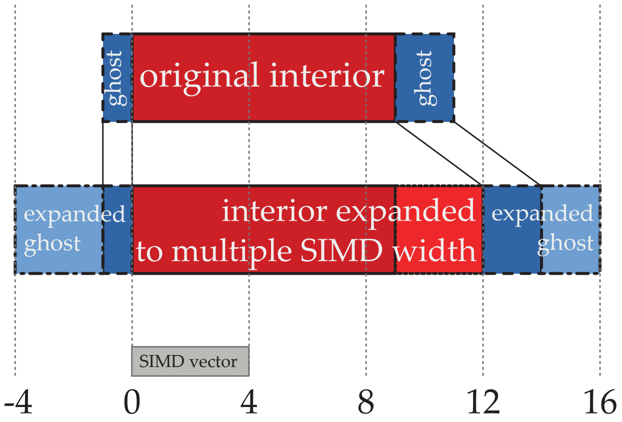


Figure 14.1: *Alignments in native SIMD data type vectorization mode.*

PATUS supports two modes of vectorization. Either native SIMD data types are used throughout the kernel, which also implies that the stencil kernel function expects grid arrays typed as native SIMD vectors (e.g., `__m128` in case of SSE). The other mode allows grids to be declared as arrays of "regular" data types, which are then re-interpreted as SIMD vectors for the calculation.

Native SIMD Data Types

In this mode, a SIMD vector is treated as an entity, and the inner most loop in the generated code containing the stencil expression iterates over vectors of grid points. Using native SIMD data types automatically ensures that data is aligned at SIMD vector boundaries. However, the PATUS code generator expects that certain alignment properties are met in unit stride direction (which have to be fulfilled by the code calling the generated stencil kernel): as shown in Fig. 14.1, the start of the interior area must be aligned at a SIMD vector boundary, and the size of the interior of the domain has to be divisible by the SIMD vector width. The domain boundaries and ghost zones have to be padded to full multiples of SIMD vector lengths. If parallelization along the unit stride direction is done, PATUS will automatically take care of dividing the domain along SIMD vector boundaries and of adding the appropriate ghost zones.

This version of vectorization carries less overhead; there are no cleanup loops which are required to deal with non-native SIMD data types. However, code vectorized in this fashion is not trivial to integrate into existing codes. On the command line, native SIMD data type vectorization can be turned on by setting the option `--use-native-simd-datatypes=yes`. It is turned off by default. Also, the benchmark results presented in Chapter 10 were *not* done in this mode.

Non-Native SIMD Data Types

In non-native SIMD data type mode, grids are declared as arrays of regular floating point data types (`float`, `double`) and passed as as such to the stencil kernel function. In this mode we do not make any assumptions on the alignment of inner nodes as restrictive as in native SIMD mode. However, we make the assumption that the number of grid points in the full grid in unit stride direction is divisible by the SIMD vector length. This requirement allows to forgo shuffle or unaligned move instructions when accessing neighboring points in non-unit stride directions as discussed above.

Non-native SIMD data type vectorization is associated and tightly integrated with loop unrolling. It only affects the inner-most loop directly containing the stencil call, though. This loop is split into three loops:

- The *prologue loop* does scalar stencil computations for as many iterations until the output grid node is aligned at a vector boundary. The number of iterations in the prologue loop is determined by the address of the first output grid node to which data is written to and the SIMD vector width. We assume that all the grids passed to the stencil kernel are aligned at vector boundaries. An example is shown in Example 14.2. Assuming a SIMD vector is 16 Bytes long and `idx0` points to the first computed grid point in `u`, `prologend` calculates the end index for the prologue loop.

- The *main loop* is the loop which does the bulk of the computation in a vectorized fashion, and which also is unrolled if the loop unrolling configuration asks to unroll the inner-most loop. In the listing in Example 14.2, the loop is unrolled twice, and hence the index of the main loop is incremented by twice the SSE single precision vector length after each iteration.

- The *epilogue loop* is what corresponds to the cleanup loop for loop unrolling. It processes the remaining iterates in a scalar manner. If the main loop was unrolled, the first iterates of the epilogue loop could be vectorized, but this was omitted in favor of less loop control overhead.

Example 14.2: *Non-native SIMD data type vectorization.*

This listing shows the prologue, main, and epilogue loops in non-native SIMD data type vectorization.

```
prologend = min(
  v_idx_x + (16 - (((uintptr_t) &u[idx0]) + 15) & 15) /
    sizeof(float),
  v_idx_x_max);

// prolog loop
for (p_idx_x = v_idx_x; p_idx_x < prologend; p_idx_x++) {
  // scalar stencil code
}

// main loop
for ( ; p_idx_x < v_idx_x_max - 7; p_idx_x += 8) {
  // vectorized and unrolled stencil code
}

// epilog loop
for ( ; p_idx_x < v_idx_x_max; p_idx_x++) {
  // scalar stencil code
}
```

The length of data vectors per data type is specified by the architecture description. It also provides a mapping from arithmetic operation-data type pairs to vector intrinsics replacing the arithmetic operation. Additionally, if there is no intrinsic available for a certain type of operation, the code generator back-end can also be modified so as to provide an architecture-specific implementation for that operation. In this way, flexibility is guaranteed that the vectorization can be mapped to any architecture supporting a SIMD mode and to any implementation of the SIMD mode, such as SSE or AVX.

Listing 14.2 is an extract from an architecture description and shows how scalar data types, `float` and `double`, are mapped to their respective SSE data types, `__m128` and `__m128d`, of SIMD vector length 4 and 2, respectively; and it shows how arithmetic operations (addition, subtraction, multiplication, division, and a function — exemplified by the square root extraction) are mapped onto the double precision SSE intrinsics `_mm_add_pd`, `_mm_sub_pd`, `_mm_mul_pd`, `_mm_div_pd`, `_mm_sqrt_pd`.

Listing 14.2: *Mapping primitive types and operators to SSE equivalents.*

```
<?xml version="1.0"?>
<architectureTypes>
  <!-- Intel x86_64 architecture with SSE,
    OpenMP parallelization -->
  <architectureType name="Intel x86_64 SSE">
    <datatypes>
      <datatype basetype="float" name="__m128"
        simdVectorLength="4"/>
      <datatype basetype="double" name="__m128d"
        simdVectorLength="2"/>
    </datatypes>
    <intrinsics>
      <intrinsic baseName="plus" name="_mm_add_pd"
        datatype="__m128d"/>
      <intrinsic baseName="minus" name="_mm_sub_pd" .../>
      <intrinsic baseName="multiply" name="_mm_mul_pd" .../>
      <intrinsic baseName="divide" name="_mm_div_pd" .../>
      <intrinsic baseName="sqrt" name="_mm_sqrt_pd" .../>
      ...
    </intrinsics>
    ...
  </architectureType>
</architectureTypes>
```

As with any form of parallel execution involving floating point arithmetic, it cannot be guaranteed that the result of the computation is the same as if the computation had been carried out sequentially. This is due to the fact that floating point arithmetic breaks the associativity law, i.e., the sequence of operations has an influence on the result. In particular, Patus's stencil specification parser interprets non-bracketed expressions (e.g., $a_1 + a_2 + a_3 + \ldots$) in such a way that a balanced expression tree is created, i.e., has as minimum depth. For instance, the expression $a_1 + a_2 + a_3 + a_4$ would be interpreted as $((a_1 + a_2) + (a_3 + a_4))$. This is done in the hope that as many temporary subexpressions as possible can be calculated independently and therefore that instruction level parallelism is maximized. As binary expressions are translated from this representation to vector intrinsics, the shape of the expression tree dictates the sequence in which the vector intrinsics are executed, which might differ from the sequence which a vectorizing compiler would generate or from the sequence the expression is executed using scalar values.

14.4 NUMA-Awareness

NUMA-Awareness of the stencil code is achieved by providing a data initialization function along with the actual stencil compute kernel function, which is also parallelized in the same way as the compute kernel and initializes all the grids used by the stencil kernel. Using the same parallelization as the stencil function ensures that the data used for computation later on is made local to the CPU cores as detailed in Chapter 6.4.2 — as long as the system implements a first touch policy. The user should invoke this initialization function directly after allocating the data. If possible, the initialization routine can also be modified to reflect the actual application-specific initial conditions. This can have a beneficial impact on the cache performance. If the data is initialized with the real initial conditions separately and later, the cache data locality established in the initialization function provided by PATUS might be destroyed (in particular, if the real initialization is done sequentially).

Part V

Conclusions & Outlook

Chapter 15

Conclusion and Outlook

> While parallel machines of yesteryear may have been constructed of enchanted metals and powered by unicorn tears, today's parallel machines are as common — and nearly as inexpensive — as hamburgers.
>
> Steve Teixeira, April 19, 2010

In this thesis, we presented PATUS, a software framework for generating stencil codes from a high-level, portable specification in a domain specific language (DSL). By using this approach, we expressed our belief that domain specific languages are an effective means to bridge programmer productivity, reusability, and the ability to deliver performance, albeit sacrificing generality of the language.

Developments in hardware have driven us to a point at which it becomes troublesome to write programs which can fully exploit a machine's compute power. Parallelization, sophisticated program transformations, and hardware-specific optimizations are required, which, however, are error-prone and at odds with the need for productivity: carrying out these transformations manually not only render the code unmaintainable, but also tie it to the particular platform for which it has been crafted, i.e., the code becomes non-portable. Recently, the comeback of hardware accelerators, which show a tremendous performance potential for certain types of code structures and therefore can be highly attractive, has added to complicate the picture. Typically they require their own programming model; thus heterogeneous hardware has to be addressed in a heterogeneous coding style.

Traditionally, the burden of transforming a portable code (e.g., written in ANSI C or Fortran, which are still the prevalent languages in scientific computing) into a code adapted to the target hardware platform was laid on the compiler. Yet, we have come to accept that sophisticated transformations, in particular parallelization, which might require new, specifically tailored algorithms, are beyond the compiler's reach. Hence, we believe that concentrating on a specific domain for which we possess the knowledge of applicable, aggressive transformation and optimization techniques, which might not be generally applicable, and encapsulating these domains in concise domain specific languages, is a way to go forward in maintaining both productivity and performance.

While complementary ongoing research focuses on the development of new parallel languages, experience with these new languages — such as Chapel — has shown that embracing productivity and generality comes, at least to some extent, at the expense of performance. For instance, in an experiment conducted with the students of the high performance computing course at the University of Basel, we could observe that Chapel, although it was given good productivity grades by the students, still lags behind other programming models — in particular the DSL approach — in terms of performance [27].

Domain specific languages are robust enough to support portability across hardware platforms, therefore enabling code reusability: once a port of the back-end to the target platform exists, any implementation in the respective domain specific language automatically runs on the new hardware. Thus, when the underlying hardware changes, or when a new architecture emerges, domain specific languages are a way to support it without having to rewrite the entire application.

We implemented this DSL idea exemplarily for the class of stencil computations, the core computation of the *structured grid motif* in the Berkeley terminology. Especially in scientific computing, but also in consumer codes, stencil computations are an important class of computations. Indeed, the two applications presented in this thesis, which are based on stencil computations, are targeted at medically, scientifically, and socially relevant questions. Stencil computations arise in finite difference-type PDE solvers, and despite the finite element method being en vogue, finite differences still enjoy considerable popularity for the simplicity of the mathematical method and the simplicity of transferring it to computer code.

We deliberately separated the algorithmic implementation from the computation from the pointwise specification of the stencil structure, as they are orthogonal concepts. In PATUS, an algorithmic implementation is called a *Strategy*. Strategies are independent of both the stencil specification and of the hardware platform; to guarantee performance portability, auto-tuning is the methodology of choice: given a benchmark executable created from the stencil specification and the Strategy, we employ a search method to find the Strategy parameter combination (and even possibly a Strategy from a predefined set) which delivers the best performance on the hardware platform under consideration. The auto-tuning methodology has been adopted successfully in a number of scientific libraries, and we believe that auto-tuners will play an increasingly important role, complementing and supporting compilers, or in the future of increasingly complex and diverse hardware architectures. Using this combined DSL and auto-tuning approach, we were able to present some promising results in terms of performance for a broad range of stencils on three different hardware platforms.

Looking Forward

The work described in this thesis is but a beginning, and a lot remains to be done. In the near future, technicalities, such as CUDA-specific optimizations (taking care of hardware particularities, e.g., alignment of data, making use of shared memory and registers) or a distributed memory back-end need to be addressed, and extending the code generator to fully support temporally blocked methods.

Boundary conditions are a concern as they are one of the defining aspects of stencil codes. So far, PATUS can handle boundaries only by defining specific boundary stencils, which then are applied to the the boundary data after doing the inner sweep, or by incorporating the boundary calculation into coefficient data. The two real-world application codes incidentally met this assumption. However, multi-time step methods clearly cannot be applied in the former approach. Hence, when going forward to supporting such methods through strategies, boundary handling within the stencil kernel becomes essential, albeit non-trivial.

The auto-tuner should support tuning multiple stencils or multiple problem sizes at once by tuning over distributions rather than a single stencil and problem size. Also, we need to make advances in automating parts of the tool that have to be done manually for the time being, such as devising a method to generate an auto-tuner configuration script based on the Strategy.

Programming more complex Strategies could prove to error-prone, and errors might be hard to detect. Sketching, i.e., leaving hard-to-get-right expressions unspecified and rely on a SAT solver for finding the correct completions based on a reference implementation is an appealing option to be explored, as it was successfully applied to stencil computations in [148], although without the idea of separating the actual stencil expression from the algorithm.

For production use of the tool, tighter integration of PATUS into existing tool chains might be desirable. We chose to define our own domain specific language for stencil specifications. This more light weight approach has the advantage that we have the guarantee that the input is indeed as stencil, rendering the need for additional code analysis detecting stencil expressions unnecessary. It also guarantees that we are given exact control over the supported computation structures. This facilitates code generator features such as vectorization. However, the internal representation of a stencil is *not* tied to the stencil specification language. Therefore, PATUS can be extended modularly to support other stencil specification syntaxes by providing the corresponding parser; the actual code generator, the Strategy machinery, and the back-ends do not have to be touched. For instance, a parser could be added which accepts annotated Fortran as input format. This, of course, changes nothing in the philosophy of the approach: the stencil is still specified in a domain specific language, now *embedded* into a host language (Fortran).

Thinking further in this direction, we also may want to consider composability of DSLs, realizing that real applications have a variety of components which could be captured by (a variety of) DSLs. Language virtualization is an appealing concept which could provide this. However, the performance aspect currently ties the target language to a selection of low-level, *close-to-the-metal* languages. Language virtualization *does* allow a code generation approach by lifting the program to an internal AST-like representation, but an effective mechanism must be found for the interoperability between the host language and the generated code.

In a broader context, as parallel computing becomes the norm rather than the exception, the disciplines of software engineering and high performance computing will have to inspire each other. For instance, today, high performance computing applications are typically monolithic pieces of software; the typical established reuse and composability principles in scientific computing are through library calls, and even when using parallel libraries technical difficulties need to be overcome: threaded libraries are parallelized with OpenMP, Intel's Threading Building Blocks, Cilk++, or some other threading library, and contain their own resource management. Thus, when an application utilizes multiple threaded libraries, the performance can suffer from destructive interference of the libraries caused by over-subscription of the hardware resources. The problem really lies in the way how operating systems virtualize hardware resources. Two or more software threads sharing a single hardware thread force frequent context switches and abet cache trashing, thus degrading performance. Therefore it is desirable that libraries rely on a common resource management infrastructure such as Lithe [128], which advocates that the caller allocates resources for the callee and provides a mechanism precluding over-subscription of the hardware resources. It proposes the use of actual hardware threads ("harts") rather than virtual software threads.

Altering scientific questions to be answered by a simulation might entail non-trivial modifications of the simulation software. If a simulation could be pieced together from black box components which are accessible through a well-defined interface, they would become faster to build and more maintainable. Efforts in the direction of composable scientific software are addressed by the Common Component Architecture Forum [1]. In consumer and business informatics, such component systems are known in the form of the service-oriented architecture (SOA) architectural pattern. For instance, web services are a concrete instantiation of SOA. The interface of a web service is exposed through a clearly defined description in an interface definition language: the web service description language (WDSL), which eases the composability of web services without the need of knowing their internals.

Other inter-disciplinary cooperations must include tight integration of the hardware- and software design processes. One of the problems encountered when microprocessor manufacturers and software developers each go their own way in optimizing their designs is that processors are optimized based on fixed benchmark codes, or even fixed benchmark binaries, and software is optimized for a given, existing hardware platform. A path is taken in the hardware/software design space that successively builds on results of the complementary domain, and is prone to steer towards a local optimum.

With this difficulty in mind, one of the ideas for implementing an exa-scale system by the end of the decade is an approach which brings both areas together: hardware/software co-design. The hope is that, based on design interaction, a path can be taken towards a global rather than merely a local optimum and that the development cycle can be substantially accelerated.

In the past, such approaches have been taken. A prominent example is the *Gravity Pipe* — short: *GRAPE* — computer, a hardware specifically designed for computing stellar dynamics calculations. The project was started in 1989 and won the Gordon Bell Prize in the *special-purpose machines* category in 1995 and in the *price performance* category in 1999 [147]. However, *GRAPE* was overtaken by the brute-force improvements of clock frequency scaling in the commodity market.

Now that frequency scaling has come to a halt, the idea for special-purpose machines has become attractive again. Rather than limiting problems that can be solved on current machines, the relevant question to answer relevant scientific issues — such as climate change — is what machine is needed to address that scientific problem. Many of today's outstanding scientific questions call for substantially increased compute requirements so that they can be answered. *Greenflash* [168] is a project that works in the direction of a special-purpose hardware platform for climate simulations. A study has been conducted for an application-targeted exa-scale machine implemented by using mainstream embedded design processes. The study concluded that in this particular case a custom system consisting of Tensilica XTensa processors with small cores, tailored to the computational requirements following an embedded design philosophy, has both a considerably lower cost and is significantly more power efficient than AMD Opteron-based or a Blue Gene/P system with comparable application-specific performance characteristics.

Processor performance emulation based on the target application prior to manufacturing the actual hardware allows efficient design process iterations. However, emulating a system, especially a large parallel system, in software is too slow to be practical. As a remedy, in [167] a research accelerator for multi-processors ("RAMP") is proposed. RAMP is a system which emulates microprocessors based on FPGAs. This reduces the slowdown compared to the actual system to a feasible range of 1–2 orders of magnitude. It therefore permits performance assessment of the entire application rather than of simplified kernels. It also allows an effective *co-tuning* approach: decisions about the hardware design can be made based on performance data from (auto-) tuned code instead of fixed code instances.

For a broader context than one specific target application, motifs can be helpful in the co-design process. The intention behind TORCH [91] is to create a kernel reference testbed to drive hardware/software co-design. To this end, it provides a high-level description of algorithmic instances within motifs, reference implementations, generators for scalable input data sets, and verification mechanisms. As a basis for hardware and software optimization research, it aims at being agnostic of hardware and software instances and of algorithms developed and tuned for these ecosystems.

Whether scientific simulations, which are to run on large supercomputer systems, or personal computing is targeted, the current hardware trends dictate that parallel computing permeate all disciplines in computer science. Parallelism has grown from the niche reserved to scientific computing and supercomputing specialists to a crosscutting concern; parallelism has become a mainstream matter. Gradually, parallel constructs have begun to seep into standard APIs, e.g., in the form of Java's concurrency package or as new features of Microsoft's Visual Studio in the form of the parallel .NET extensions or parallel libraries, and taking advantage of new language constructs to ease parallel programming, such as lambda expressions defined in the upcoming C++0x standard [158]. Simultaneously, new languages with inherent support for concurrency are being developed.

The ubiquity of parallelism obviously has consequences for education as well. Instead of being taught in advanced classes in the computer science curriculum, parallel programming needs to be integrated as one of the pillars in computer science and treated as self-evidently in the undergraduate program, which is for instance proposed by the NSF-TCPP [156]. It is encouraging to see that these ideas are gradually being adopted worldwide.

Thus, there is hope that advances in other fields of computer science, especially in software and algorithm engineering, can stimulate parallel computing and vice versa, and that the general computer science community can ingest the lessons learned by the community of parallel computing.

Bibliography

[1] The Common Component Architecture Forum. `http://www.cca-forum.org/`. Accessed August 2011. [cited at p. 235]

[2] LANL Advanced Computing Laboratory. POOMA. `http://acts.nersc.gov/pooma/`. Accessed July 2011. [cited at p. 52]

[3] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA Projects. *J. Phys. Conf. Ser.*, 180(1):1–5, July 2009. [cited at p. 51]

[4] Y. Ajima, S. Sumimoto, and T. Shimizu. Tofu: A 6D Mesh/Torus Interconnect for Exascale Computers. *Computer*, 42:36–40, November 2009. [cited at p. 15]

[5] J. Allen. *Dependence analysis for subscripted variables and its application to program transformations*. PhD thesis, Rice University, Houston, TX, USA, 1983. [cited at p. 36]

[6] G. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proc. AFIPS Spring Joint Computer Conference 1967*, pages 483–485, New York, NY, USA, 1967. ACM. [cited at p. 18]

[7] J. Ansel, Y. Won, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe. Language and Compiler Support for Auto-Tuning Variable-Accuracy Algorithms. Technical Report MIT-CSAIL-TR-2010-032, Massachusetts Institute of Technology, Cambridge, MA, July 2010. [cited at p. 44]

[8] Applied Numerical Algorithms Group (ANAG), Lawrence Berkeley National Laboratory, Berkeley, CA. Chombo Website. `http://seesar.lbl.gov/anag/chombo`. Accessed July 2011. [cited at p. 87]

[9] K. Asanović, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick. The Landscape of Parallel Computing Research: a View from Berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, December 2006. [cited at p. 4, 13, 45, 46, 50, 51]

[10] D. Bacon, S. Graham, and O. Sharp. Compiler Transformations for High-Performance Computing. *ACM Comput. Surv.*, 26:345–420, December 1994. [cited at p. 33, 34]

[11] H. Bae, L. Bachega, C. Dave, S. Lee, S. Lee, S. Min, R. Eigenmann, and S. Midkiff. Cetus: A Source-to-Source Compiler Infrastructure for Multicores. In *Proc. Int'l Workshop on Compilers for Parallel Computing (CPC 2009)*, 2009. [cited at p. 189]

[12] G. Ballard, J. Demmel, and I. Dumitriu. Minimizing Communication for Eigenproblems and the Singular Value Decomposition. *CoRR*, abs/1011.3077:1–44, 2010. [cited at p. 31]

[13] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Communication-optimal Parallel and Sequential Cholesky Decomposition. *SIAM J. Sci. Comp.*, 32(6):3495–3523, 2010. [cited at p. 31]

[14] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA, USA, 1988. [cited at p. 36]

[15] J. Barnes and P. Hut. A hierarchical $\mathcal{O}(N \log N)$ force-calculation algorithm. *Nature*, 324(6096):446–449, December 1986. [cited at p. 48]

[16] A. Basumallik, S. Min, and R. Eigenmann. Programming Distributed Memory Sytems Using OpenMP. In *Proc. IEEE Int'l Parallel & Distributed Processing Symposium (IPDPS 2007)*, pages 1–8. IEEE Computer Society, 2007. [cited at p. 27]

[17] N. Bell and M. Garland. Efficient Sparse Matrix-Vector Multiplication on CUDA. Technical report, NVIDIA Corporation, Santa Clara, CA, USA, December 2008. [cited at p. 47]

[18] J. Berenger. A Perfectly Matched Layer for the Absorption of Electromagnetic Waves. *J. Comput. Phys.*, 114:185–200, October 1994. [cited at p. 69]

[19] M. Berger and J. Oliger. Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations. *J. of Comput. Phys.*, 53:484–512, 1984. [cited at p. 49]

[20] L. Beshenov. Maxima, a Computer Algebra System. `http://maxima.sourceforge.net/`. Accessed July 2011. [cited at p. 189]

[21] A. Bhatele, P. Jetley, H. Gahvari, L. Wesolowski, W.D. Gropp, and L.V. Kalé. Architectural Constraints to Attain 1 Exaflop/s for Three Scientific Application Classes. In *Proc. IEEE Int'l Parallel & Distributed Processing Symposium (IPDPS 2011)*, May 2011. [cited at p. 9]

[22] C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Comput. Surv.*, 35:268–308, September 2003. [cited at p. 130, 137]

[23] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2008)*, 2008. [cited at p. 24, 87]

[24] U. Brüning. Exascale Computing - What is required for HW? Presentation, Int'l Supercomputing Conference (ISC 2011), 2011. [cited at p. 15]

[25] W. Burger and M. Burge. *Digital image processing: an algorithmic introduction using Java*. Springer, 2008. [cited at p. 66]

[26] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proc. SIGPLAN Symposium on Compiler Construction (SIGPLAN 1986)*, pages 162–175. ACM, 1986. [cited at p. 36]

[27] H. Burkhart, M. Christen, M. Rietmann, M. Sathe, and O. Schenk. Run, Stencil, Run! A Comparison of Modern Parallel Programming Paradigms. In *PARS-Mitteilungen 2011*, 2011. [cited at p. 232]

[28] H. Burkhart, R. Frank, and G. Hächler. ALWAN: A Skeleton Programming Language. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models*, volume 1061 of *Lecture Notes in Computer Science*, pages 407–410. Springer Berlin / Heidelberg, 1996. [cited at p. 28]

[29] S. Carr and Y. Guan. Unroll-and-jam using uniformly generated sets. In *Proc. ACM/IEEE Int'l Symposium on Microarchitecture (MICRO 30)*, pages 349–357. IEEE Computer Society, 1997. [cited at p. 218]

[30] B. Catanzaro, S. Kamil, Y. Lee, K. Asanović, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox. SEJITS: Getting Productivity and Performance with Selective Embedded JIT Specialization. In *Proc. First Workshop on Programmable Models for Emerging Architecture (PMEA) at the ACM Int'l Conference on Parallel Architectures and Compilation Techniques (PACT 2009)*, 2009. [cited at p. 44]

[31] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language Virtualization for Heterogeneous Parallel Computing. In *Proc. ACM Int'l Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA 2010)*, pages 835–847, 2010. [cited at p. 43, 44]

[32] B. Chamberlain, D. Callahan, and H. Zima. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.*, 21:291–312, August 2007. [cited at p. 30]

[33] B. Chamberlain, S. Choi, E. Lewis, L. Snyder, W. Weathersby, and C. Lin. The Case for High-Level Parallel Programming in ZPL. *IEEE Comput. Sci. Eng.*, 5:76–86, July 1998. [cited at p. 30, 31]

[34] P. Charles, P. Cheng, C. Donawa, J. Dolby, P. Gallop, C. Grothoff, A. Kielstra, and F. Pizlo. Report on the Experimental Language X10. Technical report, IBM, 2006. [cited at p. 31]

[35] R. Chau. Integrated CMOS Tri-Gate Transistors. `http://www.intel.com/technology/silicon/integrated_cmos.htm`. Accessed August 2011. [cited at p. 11, 13]

[36] M. Christen, N. Keen, T. Ligocki, L. Oliker, J. Shalf, B. Van Straalen, and S. Williams. Automatic Thread-Level Parallelization in the Chombo AMR Library. Technical report, Lawrence Berkeley National Laboratory, Berkeley CA, USA / University of Basel, Switzerland, 2011. [cited at p. 87]

[37] M. Christen, O. Schenk, and H. Burkhart. Automatic Code Generation and Tuning for Stencil Kernels on Modern Microarchitectures. In *Proc. Int'l Supercomputing Conference (ISC 2011)*, volume 26, pages 205–210, 2011. [cited at p. 61]

[38] M. Christen, O. Schenk, and H. Burkhart. PATUS: A Code Generation and Autotuning Framework For Parallel Iterative Stencil Computations on Modern Microarchitectures. In *Proc. IEEE Int'l Parallel & Distributed Processing Symposium (IPDPS 2011)*, pages 1–12, 2011. [cited at p. 61]

[39] M. Christen, O. Schenk, and H. Burkhart. PATUS: A Code Generation and Auto-Tuning Framework For Parallel Stencil Computations. In *Proc. Cetus Users and Compiler Infastructure Workshop*, pages 1–5, 2011. [cited at p. 61]

[40] M. Christen, O. Schenk, P. Messmer, E. Neufeld, and H. Burkhart. Accelerating Stencil-Based Computations by Increased Temporal Locality on Modern Multi- and Many-Core Architectures. In *High performance*

and hardware aware computing: Proceedings of the First International Workshop on New Frontiers in High-performance and Hardware-aware Computing (HipHaC'08), pages 47–54. Universitätsverlag Karlsruhe, 2008. [cited at p. 97, 99, 119]

[41] M. Christen, O. Schenk, E. Neufeld, P. Messmer, and H. Burkhart. Parallel Data-Locality Aware Stencil Computations on Modern Micro-Architectures. In *Proc. IEEE Int'l Parallel & Distributed Processing Symposium (IPDPS 2009)*, pages 1–10, May 2009. [cited at p. 97, 99, 119]

[42] M. Christen, O. Schenk, E. Neufeld, M. Paulides, and H. Burkhart. Many-core Stencil Computations in Hyperthermia Applications. In J. Dongarra, D. Bader, and J. Kurzak, editors, *Scientific Computing with Multicore and Accelerators*, pages 255–277. CRC Press, 2010. [cited at p. 97, 99]

[43] P. Colella. Defining Software Requirements for Scientific Computing. Presentation, 2004. [cited at p. 45]

[44] P. Colella, D. Graves, T. Ligocki, D. Martin, D. Modiano, D. Serafini, and B. Van Straalen. Chombo Software Package for AMR Applications: Design Document. `http://davis.lbl.gov/apdec/designdocuments/chombodesign.pdf`. Accessed July 2011. [cited at p. 52, 87]

[45] UPC Consortium. UPC Specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Laboratory, 2005. [cited at p. 29]

[46] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor. *IEEE Micro*, 30:16–29, 2010. [cited at p. 151]

[47] J. Cooley and J. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comput.*, 19(90):297–301, April 1965. [cited at p. 47]

[48] Standard Performance Evaluation Corporation. Standard performance evaluation corporation. `http://www.spec.org`. Accessed July 2011. [cited at p. 45]

[49] Y. Cui, K. Olsen, T. Jordan, K. Lee, J. Zhou, P. Small, D. Roten, G. Ely, D. Panda, A. Chourasia, J. Levesque, S. Day, and P. Maechling. Scalable Earthquake Simulation on Petascale Supercomputers. In *Proc. ACM/IEEE Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC 2010)*, pages 1–20, Washington, DC, USA, 2010. IEEE Computer Society. [cited at p. 180, 181, 182]

[50] L. Dalguer. Staggered-Grid Split-Node Method for Spontaneous Rupture Simulation. *J. Geophys. Res.*, 112(B02302):1–15, 2007. [cited at p. 182]

[51] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors. *SIAM Review*, 51(1):129–159, 2009. [cited at p. 97]

[52] K. Datta, S. Williams, V. Volkov, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Auto-tuning Stencil Computations on Diverse Multicore Architectures. In *Parallel and Distributed Computing*. IN-TECH Publishers, 2009. [cited at p. 218]

[53] K. Datta, S. Williams, V. Volkov, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Auto-tuning Stencil Computations on Multicore and Accelerators. In J. Dongarra, D. Bader, and J. Kurzak, editors, *Scientific Computing with Multicore and Accelerators*, pages 219–253. CRC Press, 2010. [cited at p. 84, 109]

[54] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51:107–113, January 2008. [cited at p. 50]

[55] D. Delling, A. Goldberg, A. Nowatzyk, , and R. Werneck. PHAST: Hardware-Accelerated Shortest Path Trees. In *Proc. IEEE Int'l Parallel & Distributed Processing Symposium (IPDPS 2011)*, pages 1–11, 2011. [cited at p. 56, 57]

[56] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Communication-Optimal Parallel and Sequential QR and LU Factorizations. Technical Report EECS-2008-89, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2008. [cited at p. 31]

[57] M. Dewhirst, Z. Vujaskovic, E. Jones, and D. Thrall. Re-setting the Biological Rationale for Thermal Therapy. *Int. J. Hyperthermia*, 21:779–790, 2005. [cited at p. 175]

[58] E. Dijkstra. A note on two problems in connexion with Graphs. *Numer. Math.*, 1(1):269–271, December 1959. [cited at p. 55]

[59] N. Edmonds, A. Breuer, D. Gregor, and A. Lumsdaine. Single-Source Shortest Paths with the Parallel Boost Graph Library. In *The Ninth DIMACS Implementation Challenge: The Shortest Path Problem*, Piscataway, NJ, November 2006. [cited at p. 56, 57]

[60] K. Fatahalian, D. Horn, T. Knight, L. Leem, M. Houston, J. Park, M. Erez, M. Ren, A. Aiken, W. Dally, and P. Hanrahan. Sequoia: Programming the Memory Hierarchy. In *Proc. ACM/IEEE Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC 2006)*, pages 1–13. ACM, 2006. [cited at p. 32]

[61] L. Flynn. Intel Halts Development of 2 New Microprocessors. *The New York Times*, May, 8 2004. `http://www.nytimes.com/2004/05/08/business/08chip.html`. [cited at p. 11]

[62] Center for Discrete Mathematics & Theoretical Computer Science. 9th DIMACS Implementation Challenge — Shortest Paths. `http://www.dis.uniroma1.it/~challenge9/index.shtml`, 2006. Accessed August 2011. [cited at p. 56]

[63] International Technology Roadmap for Semiconductors. 2010 Tables, Lithography. `http://www.itrs.net/Links/2010ITRS/2010Update/ToPost/2010Tables_LITHO_FOCUS_D_ITRS.xls`, 2011. [cited at p. 11]

[64] Consortium for small-scale modelling. Core Documentation of the COSMO-Model. `http://cosmo-model.cscs.ch/content/model/documentation/core/default.htm`. Accessed July 2011. [cited at p. 64, 158]

[65] M. Frigo and S. Johnson. The Design and Implementation of FFTW3. *P. IEEE — Special issue on Program Generation, Optimization, and Platform Adaptation*, 93(2):216–231, 2005. [cited at p. 52, 77, 128]

[66] M. Frigo and V. Strumpen. Cache oblivious stencil computations. In *Proc. ACM Int'l Conference on Supercomputing (ICS 2005)*, pages 361–366, 2005. [cited at p. 85, 101, 102]

[67] M. Frigo and V. Strumpen. The Cache Complexity of Multithreaded Cache Oblivious Algorithms. In *Proc. ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2006)*, pages 271–280. ACM, 2006. [cited at p. 85, 102]

[68] J. Gablonsky. *Modifications of the DIRECT Algorithm*. PhD thesis, North Carolina State University, 2001. [cited at p. 137]

[69] M. Gardner. The Fantastic Combinations of John Conway's New Solitaire Game "Life". *Scientific American*, 223:120–123, October 1970. [cited at p. 67]

[70] K. Gatlin. Power Your App with the Programming Model and Compiler Optimizations of Visual C++. `http://msdn.microsoft.com/en-us/magazine/cc163855.aspx#S4`, 2005. [cited at p. 33]

[71] G. Goff, K. Kennedy, and C. Tseng. Practical Dependence Testing. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1991)*, pages 15–29. ACM, 1991. [cited at p. 36]

[72] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2003. [cited at p. 53, 56]

[73] L. Greengard and V. Rokhlin. A Fast Algorithm for Particle Simulations. *J. Comput. Phys.*, 73:325–348, December 1987. [cited at p. 48]

[74] M. Griebl, C. Lengauer, and S. Wetzel. Code Generation in the Polytope Model. In *Proc. ACM Int'l Conference on Parallel Architectures and Compilation Techniques (PACT 1998)*, pages 106–111. IEEE Computer Society, 1998. [cited at p. 39]

[75] L. Grigori, J. Demmel, and H. Xiang. Communication avoiding Gaussian elimination. In *Proc. ACM/IEEE Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC 2008)*, pages 1–12, 2008. [cited at p. 31]

[76] M. Gschwind. Chip multiprocessing and the cell broadband engine. In *Proc. ACM Conference on Computing Frontiers (CF 2006)*, pages 1–8. ACM, 2006. [cited at p. 13, 99]

[77] Richard Günther. FreePOOMA — Parallel Object Oriented Methods and Applications. `http://www.nongnu.org/freepooma/`. Accessed July 2011. [cited at p. 52]

[78] John L. Gustafson. Reevaluating Amdahl's Law. *Commun. ACM*, 31:532–533, May 1988. [cited at p. 22]

[79] G. Hächler and H. Burkhart. Implementing the ALWAN Communication and Data Distribution Library Using PVM. In A. Bode, J. Dongarra, T. Ludwig, and V. Sunderam, editors, *Parallel Virtual Machine – EuroPVM '96*, volume 1156 of *Lecture Notes in Computer Science*, pages 243–250. Springer Berlin / Heidelberg, 1996. [cited at p. 28]

[80] M. Hall, J. Chame, C. Chen, J. Shin, Gabe Rudy, and Malik Khan. Loop Transformation Recipes for Code Generation and Auto-Tuning. In Guang Gao, Lori Pollock, John Cavazos, and Xiaoming Li, editors, *Languages and Compilers for Parallel Computing*, volume 5898 of *Lecture Notes in Computer Science*, pages 50–64. Springer Berlin / Heidelberg, 2010. [cited at p. 87, 127, 129]

[81] Mark D. Hill and Michael R. Marty. Amdahl's Law in the Multicore Era. *Computer*, 41:33–38, July 2008. [cited at p. 20, 21]

[82] R. Hooke and T. A. Jeeves. Direct Search Solution of Numerical and Statistical Problems. *J. ACM*, 8(2):212–229, April 1961. [cited at p. 133]

[83] IBM. X10: Performance and Productivity at Scale. `http://www.x10-lang.org/`. Accessed July 2011. [cited at p. 30]

[84] Intel. High-κ and Metal Gate Research. `http://www.intel.com/technology/silicon/high-k.htm`. Accessed August 2011. [cited at p. 11]

[85] Intel. Intel® Advanced Vector Extensions Programming Reference. `http://software.intel.com/file/35247/`. Accessed June 2011. [cited at p. 41]

[86] Intel. Interprocedural Optimization (IPO) Overview. `http://software.intel.com/sites/products/documentation/studio/composer/en-us/2009/compiler_c/optaps/common/optaps_ipo_mult.htm`. Accessed August 2011. [cited at p. 33]

[87] Intel. Moore's law — 40th anniversary. `http://www.intel.com/pressroom/kits/events/moores_law_40th/`, 2005. Accessed August 2011. [cited at p. 12]

[88] Intel. Petascale to Exascale — Extending Intel's Commitment. `http://download.intel.com/pressroom/archive/reference/ISC_2010_Skaugen_keynote.pdf`, 2010. Presentation, Accessed August 2011. [cited at p. 11]

[89] G. Jin, J. Mellor-Crummey, and R. Fowler. Increasing Temporal Locality with Skewing and Recursive Blocking. In *Proc. ACM/IEEE Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC 2001)*, pages 43–43. ACM, 2001. [cited at p. 94]

[90] D. Jones, C. Perttunen, and B. Stuckman. Lipschitzian Optimization Without the Lipschitz Constant. *J. Optim. Theory Appl.*, 79:157–181, October 1993. [cited at p. 137]

[91] A. Kaiser, S. Williams, K. Madduri, K. Ibrahim, D. Bailey, J. Demmel, and E. Strohmaier. TORCH Computational Reference Kernels: A Testbed for Computer Science Research. Technical Report UCB/EECS-2010-144, EECS Department, University of California, Berkeley, December 2010. [cited at p. 45, 237]

[92] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An Auto-tuning Framework For Parallel Multicore Stencil Computations. In *Proc. IEEE Int'l Parallel & Distributed Processing Symposium (IPDPS 2010)*, pages 1–12, April 2010. [cited at p. 52, 84, 85]

[93] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002. [cited at p. 34, 38]

[94] D. Keyes. Exaflop/s: The Why and the How. *C. R. Mec.*, 339:70–77, 2011. [cited at p. 9]

[95] Khronos OpenCL Working Group. *The OpenCL Specification*, December, 8 2008. [cited at p. 28, 122]

[96] K. Kusano, M. Sato, T. Hosomi, and Y. Seo. The Omni OpenMP Compiler on the Distributed Shared Memory of Cenju-4. In R. Eigenmann and M. Voss, editors, *OpenMP Shared Memory Parallel Programming*, volume 2104 of *Lecture Notes in Computer Science*, pages 20–30. Springer Berlin / Heidelberg, 2001. [cited at p. 27]

[97] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *Proc. ACM/IEEE Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC 2010)*, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society. [cited at p. 27]

[98] R. Lewis, V. Torczon, and M. Trosset. Direct search methods: then and now. *J. Comput. Appl. Math.*, 124(1-2):191–207, 2000. [cited at p. 131]

[99] Z. Li and Y. Song. Automatic Tiling of Iterative Stencil Loops. *ACM Trans. Program. Lang. Syst.*, 26(6):975–1028, November 2004. [cited at p. 52, 87]

[100] Z. Li and P. Yew. Some results on exact data dependence analysis. In *Proc.: Selected Papers of the Second Workshop on Languages and Compilers for Parallel Computing*, pages 374–401, London, UK, 1990. Pitman Publishing. [cited at p. 36]

[101] Z. Li, P. Yew, and C. Zhu. Data dependence analysis on multi-dimensional array references. In *Proc. ACM Int'l Conference on Supercomputing (ICS 1989)*, pages 215–224. ACM, 1989. [cited at p. 36]

[102] Y. Lin, C. Dimitrakopoulos, K. Jenkins, D. Farmer, H. Chiu, A. Grill, and P. Avouris. 100-GHz Transistors from Wafer-Scale Epitaxial Graphene. *Science*, 327(5966):662, 2010. [cited at p. 11]

[103] C. Mack. Seeing Double. *IEEE Spectrum*, 45:47–51, November 2008. [cited at p. 10]

[104] K. Madduri, D. Bader, J. Berry, and J. Crobak. Parallel Shortest Path Algorithms for Solving Large-Scale Instances. In *The Ninth DIMACS Implementation Challenge: The Shortest Path Problem*, Piscataway, NJ, November 2006. [cited at p. 56, 57]

[105] D. Maydan, J. Hennessy, and M. Lam. Efficient and exact data dependence analysis. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1991)*, pages 1–14. ACM, 1991. [cited at p. 36]

[106] J. McCalpin. STREAM: Sustainable Memory Bandwidth in High-Performance Computers. `http://www.cs.virginia.edu/stream/`. Accessed July 2011. [cited at p. 73, 149, 151, 153]

[107] J. McCalpin and D. Wonnacott. Time skewing: A value-based approach to optimizing for memory locality. Technical Report DCS-TR-379, Department of Computer Science, Rutgers University, 1998. [cited at p. 94]

[108] P. McKenney and M. Michael. Is Parallel Programming Hard, And If So, Why? `http://www.cs.pdx.edu/pdfs/tr0902.pdf`. Accessed July 2011. [cited at p. 42]

[109] K. Meffert. JGAP — Java Genetic Algorithm Package. `http://jgap.sourceforge.net/`. Accessed July 2011. [cited at p. 138]

[110] J. Meng and K. Skadron. A Performance Study for Iterative Stencil Loops on GPUs with Ghost Zone Optimizations. *Int. J. Parallel Prog.*, 39:115–142, February 2011. 10.1007/s10766-010-0142-5. [cited at p. 86, 99]

[111] Message Passing Interface Forum. The Message Passing Interface (MPI) standard. `http://www.mcs.anl.gov/research/projects/mpi/`. Accessed July 2011. [cited at p. 26]

[112] D. Miles, B. Leback, and D. Norton. Optimizing Application Performance on x64 Processor-based Systems with PGI Compilers and Tools. Technical report, The Portland Group, 2008. [cited at p. 33]

[113] F. Miller, A. Vandome, and J. McBrewster. *AltiVec*. Alpha Press, 2010. [cited at p. 41]

[114] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick. Minimizing Communication in Sparse Matrix Solvers. In *Proc. ACM/IEEE Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC 2009)*, pages 1–12, 2009. [cited at p. 31]

[115] G. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 1965. [cited at p. 10]

[116] H. Mössenböck, M. Löberbauer, and A. Wöß. The Compiler Generator Coco/R. `http://www.ssw.uni-linz.ac.at/coco`. Accessed July 2011. [cited at p. 189]

[117] M. Müller-Hannemann and S. Schirra, editors. *Algorithm Engineering: Bridging the Gap between Algorithm Theory and Practice*. Springer-Verlag, Berlin, Heidelberg, 2010. [cited at p. 54]

[118] J. A. Nelder and R. Mead. A simplex method for function minimization. *Comput. J.*, 7:308–313, 1965. [cited at p. 131, 136]

[119] E. Neufeld. *High Resolution Hyperthermia Treatment Planning*. PhD thesis, ETH Zurich, August 2008. [cited at p. 177]

[120] E. Neufeld, N. Chavannes, T. Samaras, and N. Kuster. Novel conformal technique to reduce staircasing artifacts at material boundaries for FDTD modeling of the bioheat equation. *Phys. Med. Biol.*, 52(15):4371, 2007. [cited at p. 98, 177]

[121] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs. In *Proc. ACM/IEEE Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC 2010)*, pages 1–13, Washington, DC, USA, 2010. IEEE Computer Society. [cited at p. 97, 119]

[122] R. Numrich and J. Reid. Co-array Fortran for Parallel Programming. *SIGPLAN Fortran Forum*, 17:1–31, August 1998. [cited at p. 29]

[123] NVIDIA. NVIDIA CUDA™ — NVIDIA CUDA C Programming Guide. `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf`. Accessed July 2011. [cited at p. 27, 155]

[124] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`. Accessed July 2011. [cited at p. 154]

[125] M. Odersky. Scala. `http://www.scala-lang.org`. Accessed July 2011. [cited at p. 44]

[126] K. Olukotun, P. Hanrahan, Hassan Chafi, N. Bronson, A. Sujeeth, and D. Makarov. Delite. `http://ppl.stanford.edu/wiki/index.php/Delite`. Accessed July 2011. [cited at p. 44]

[127] OpenMP Architecture Review Board. The OpenMP® API Specification for Parallel Programming. `http://www.openmp.org`. Accessed July 2011. [cited at p. 26]

[128] H. Pan, B. Hindman, and K. Asanović. Lithe: Enabling Efficient Composition of Parallel Libraries. In *Proc. USENIX Workshop on Hot Topics in Parallelism (HotPar'09)*, pages 1–11. USENIX Association, 2009. [cited at p. 235]

[129] Z. Pan and R. Eigenmann. PEAK – A Fast and Effective Performance Tuning System via Compiler Optimization Orchestration. *ACM Trans. Program. Lang. Syst.*, 30:1–43, May 2008. [cited at p. 33, 128, 132]

[130] C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization*. Dover Publications, Inc., 1998. [cited at p. 130]

[131] D. Patterson and J. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*, chapter 7.1: Roofline: A Simple Performance Model, pages 667 – 675. Morgan Kaufmann, 4 edition, November 2008. [cited at p. 71]

[132] M. Paulides, J. Bakker, E. Neufeld, J. van der Zee, P. Jansen, P. Levendag, and G. van Rhoon. The HYPERcollar: A novel applicator for hyperthermia in the head and neck. *Int. J. Hyperther.*, 23:567 – 576, 2007. [cited at p. 176]

[133] M. Paulides, J. Bakker, A. Zwamborn, and G. van Rhoon. A head and neck hyperthermia applicator: Theoretical antenna array design. *International Journal of Hyperthermia*, 23(1):59–67, 2007. [cited at p. 176]

[134] H. H. Pennes. Analysis of Tissue and Arterial Blood Temperatures in the Resting Human Forearm. *J. Appl. Physiol.*, 1(2):93–122, 1948. [cited at p. 177]

[135] D. Playner and K. Hawick. Auto-Generation of Parallel Finite-Differencing Code for MPI, TBB and CUDA. In *Proc. Int'l Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2011)*, pages 1–8, 2011. [cited at p. 86]

[136] M. Powell. An efficient method for finding the minimum of a function of several variables without calculating derivatives. *Comput. J.*, 7(2):155–162, January 1964. [cited at p. 135]

[137] C. Price, B. Robertson, and M. Reale. A hybrid Hooke and Jeeves — Direct Method for Non-Smooth Optimization. *Adv. Model. Optim.*, 11, 2009. [cited at p. 133]

[138] W. Pugh. The Omega Test: a Fast and Practical Integer Programming Algorithm for Dependence Analysis. In *Proc. ACM/IEEE Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC 1991)*, pages 4–13. ACM, 1991. [cited at p. 36]

[139] M. Püschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *P. IEEE — Special issue on Program Generation, Optimization, and Platform Adaptation*, 93(2):232–275, 2005. [cited at p. 52, 128]

[140] M. Ren, J. Y. Park, M. Houston, A. Aiken, and W. Dally. A Tuning Framework for Software-Managed Memory Hierarchies. In *Proc. ACM Int'l Conference on Parallel Architectures and Compilation Techniques (PACT 2008)*, pages 280–291, 2008. [cited at p. 32]

[141] L. Renganarayanan, D. Kim, S. Rajopadhye, and M. Strout. Parameterized Tiled Loops for Free. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2007)*, pages 405–414, New York, NY, USA, 2007. ACM. [cited at p. 87]

[142] G. Rivera and C. Tseng. Tiling optimizations for 3D scientific computations. In *Proc. ACM/IEEE Conference on Supercomputing (SC 2000)*, 2000. [cited at p. 90]

[143] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Pub. Co., 1996. [cited at p. 69]

[144] Palash Sarkar. A brief history of cellular automata. *ACM Comput. Surv.*, 32:80–107, March 2000. [cited at p. 66]

[145] V. Sarkar. Language and Virtual Machine Challenges for Large-scale Parallel Systems. Presentation, `http://www.research.ibm.com/vee04/Sarkar.pdf`. Accessed July 2011. [cited at p. 30]

[146] N. Savage. First Graphene Integrated Circuit. *IEEE Spectrum*, 48, June 2011. [cited at p. 11]

[147] Supercomputing Conference Series. Gordon Bell Prize Winners. `http://www.sc2000.org/bell/pastawrd.htm`. Accessed July 2011. [cited at p. 236]

[148] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Sketching stencils. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2007)*, pages 167–178. ACM, 2007. [cited at p. 234]

[149] Y. Song and Z. Li. New Tiling Techniques to Improve Cache Temporal Locality. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1999)*, 1999. [cited at p. 87]

[150] G. Sreenivasa, J. Gellermann, B. Rau, J. Nadobny, P. Schlag, P. Deuflhard, R. Felix, and P. Wust. Clinical use of the hyperthermia treatment planning system HyperPlan to predict effectiveness and toxicity. *Int. J. Radiat. Oncol. Biol. Phys.*, 55:407–419, February 2003. [cited at p. 177]

[151] J. Strachan, D. Strukov, J. Borghetti, J. Yang, G. Medeiros-Ribeiro, and R. Williams. The Switching Location of a Bipolar Memristor: Chemical,

Thermal and Structural Mapping. *Nanotechnology*, 22(25):254015, 2011. [cited at p. 11]

[152] R. Strzodka, M. Shaheen, and D. Pajak. Time skewing made simple. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2011)*, pages 295–296, New York, NY, USA, 2011. ACM. [cited at p. 86, 94]

[153] R. Strzodka, M. Shaheen, D. Pajak, and H. Seidel. Cache oblivious parallelograms in iterative stencil computations. In *Proc. ACM Int'l Conference on Supercomputing (ICS 2010)*, pages 49–59, New York, NY, USA, 2010. ACM. [cited at p. 86, 102, 103]

[154] M. Stürmer. A framework that supports in writing performance-optimized stencil-based codes. Technical report, Universität Erlangen, Institut für Informatik, 2010. [cited at p. 52, 86, 105]

[155] Y. Tang, R. Chowdhury, B. Kuszmaul, C. Luk, and C. Leiserson. The Pochoir Stencil Compiler. In *Proc. ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2011)*, pages 117–128. ACM, 2011. [cited at p. 52, 85, 102]

[156] TCPP. NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing Core Topics for Undergraduates. `http://www.cs.gsu.edu/~tcpp/curriculum/index.php`. Accessed August 2011. [cited at p. 237]

[157] The Cetus Team. Cetus – A Source-to-Source Compiler Infrastructure for C Programs. `http://cetus.ecn.purdue.edu/`. Accessed July 2011. [cited at p. 24, 189]

[158] S. Teixeira. Visual Studio 2010 Brings Parallelism Mainstream. *Dr. Dobb's*, April 19 2010. [cited at p. 237]

[159] A. Tiwari and J. Hollingsworth. Online Adaptive Code Generation and Tuning. In *Proc. IEEE Int'l Parallel & Distributed Processing Symposium (IPDPS 2011)*, May 2011. [cited at p. 129, 130, 136]

[160] S. Tomov, J. Dongarra, and M. Baboulin. Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems. *Parallel Computing*, 36:232–240, June 2010. [cited at p. 51]

[161] TOP500.org. TOP500 Supercomputer Sites. `http://www.top500.org`. Accessed July 2011. [cited at p. 15, 23, 40, 154]

[162] J. Treibig, G. Wellein, and G. Hager. Efficient Multicore-Aware Parallelization Strategies for Iterative Stencil Computations. *Journal of Computational*

Science, 2(2):130 – 137, 2011. Simulation Software for Supercomputers. [cited at p. 99]

[163] D. Unat, X. Cai, and S. Baden. Mint: Realizing CUDA Performance in 3D Stencil Methods with Annotated C. In *Proc. ACM Int'l Conference on Supercomputing (ICS 2011)*, pages 214–224, New York, NY, USA, 2011. ACM. [cited at p. 52, 86]

[164] J. van der Zee. Heating the Patient: a Promising Approach? *Ann. Oncol.*, 13(8):1173–1184, 2002. [cited at p. 175]

[165] A. van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000. [cited at p. 43]

[166] R. Vuduc, J. Demmel, and K. Yelick. OSKI: A Library of Automatically Tuned Sparse Matrix Kernels. *J. Phys. Conf. Ser.*, 16(1):521, 2005. [cited at p. 51, 128]

[167] J. Wawrzynek, D. Patterson, M. Oskin, S. Lu, C. Kozyrakis, J. Hoe, D. Chiou, and K. Asanović. RAMP: Research Accelerator for Multiple Processors. *IEEE Micro*, 27:46–57, March 2007. [cited at p. 236]

[168] M. Wehner, L. Oliker, and J. Shalf. Towards Ultra-High Resolution Models of Climate and Weather. *Int. J. High Perform. C.*, 22(2):149–165, 2008. [cited at p. 9, 236]

[169] S. Weinbaum and L. M. Jiji. A New Simplified Bioheat Equation for the Effect of Blood Flow on Local Average Tissue Temperature. *J. Biomech. Eng.-T. ASME*, 107(2):131–139, 1985. [cited at p. 178]

[170] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske. Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. In *Proc. IEEE Int'l Computer Software and Applications Conference (COMPSAC 2009)*, pages 579–586, 2009. [cited at p. 86, 99]

[171] C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proc. ACM/IEEE Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC 2009)*, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society. [cited at p. 51]

[172] R. C. Whaley and A. Petitet. Minimizing Development and Maintenance Costs in Supporting Persistently Optimized BLAS. *Softw. Pract. Exper.*, 35:101–121, February 2005. [cited at p. 51, 128]

[173] GCC Wiki. Interprocedural_optimizations. `http://gcc.gnu.org/wiki/Interprocedural_optimizations`, 2008. Accessed August 2011. [cited at p. 33]

[174] Wikipedia. Microprocessor chronology. `http://en.wikipedia.org/wiki/Microprocessor_chronology`, August 17, 2011. Accessed August 2011. [cited at p. 12]

[175] Wikipedia. Transistor count. `http://en.wikipedia.org/wiki/Transistor_count`, June 19, 2011. Accessed August 2011. [cited at p. 12]

[176] S. Williams. *Auto-tuning Performance on Multicore Computers*. PhD thesis, EECS Department, University of California, Berkeley, December 2008. [cited at p. 51, 63, 74, 84, 109, 129]

[177] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. Scientific computing kernels on the Cell processor. *Int. J. Parallel Program.*, 35(3):263–298, 2007. [cited at p. 97]

[178] D. Wohlfeld, F. Lemke, S. Schenk, H. Froening, and U. Bruening. High Density Active Optical Cable — From a New Concept to a Prototype. In *Proc. SPIE Conference on Optoelectronic Interconnects and Component Integration*, volume 7944, pages 1–7, 2011. [cited at p. 15]

[179] M. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Stanford University, Stanford, CA, USA, 1992. [cited at p. 39, 40]

[180] M. Wolf and M. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Trans. Parallel Distrib. Syst.*, 2:452–471, October 1991. [cited at p. 39, 40]

[181] M. Wolf and M. Lam. A data locality optimizing algorithm. *SIGPLAN Not.*, 26:30–44, May 1991. [cited at p. 90]

[182] M. Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge, MA, USA, 1990. [cited at p. 36]

[183] L. Wolsey. *Integer Programming*. John Wiley & Sons, Inc., 1998. [cited at p. 130]

[184] D. Wonnacott. Time skewing for parallel computers. In *Proc. Int'l Workshop on Compilers for Parallel Computing (CPC 1999)*, pages 477–480. Springer-Verlag, 1999. [cited at p. 94]

[185] D. Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *Proc. IEEE Int'l Parallel & Distributed Processing Symposium (IPDPS 2000)*, Cancun, Mexico, 2000. [cited at p. 94]

[186] P. Wust, B. Hildebrandt, G. Sreenivasa, B. Rau, J. Gellermann, H. Riess, R. Felix, and P. Schlag. Hyperthermia in Combined Treatment of Cancer. *Lancet Oncol.*, 3:487–497, 2002. [cited at p. 175]

[187] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A High-Performance Java Dialect. *Concurrency–Pract. Ex.*, 10(11-13):825–836, 1999. [cited at p. 29]

[188] A. Yzelman and R. Bisseling. Cache-Oblivious Sparse Matrix–Vector Multiplication by Using Sparse Matrix Partitioning Methods. *SIAM J. Scientific Computing*, 31(4):3128–3154, 2009. [cited at p. 47]

Appendices

Appendix A

Patus Usage

A.1 Code Generation

The PATUS code generator is invoked by the following command:

```
java -jar patus.jar codegen
  --stencil=<Stencil File>   --strategy=<Strategy File>
  --architecture=<Architecture Description File> ,<Hardware Name>
  [--outdir=<Output Directory> ][--generate=<Target> ]
  [--kernel-file=<Kernel Output File Name> ]
  [--compatibility={ C | Fortran }]
  [--unroll=<UnrollFactor1> ,<UnrollFactor2> ,... ]
  [--use-native-simd-datatypes={ yes | no }]
  [--create-validation={ yes | no }]
  [--validation-tolerance=<Tolerance> ]
  [--debug=<DebugOption1> ,[<DebugOption2> ,[... ,
    [<DebugOptionN> ]...]]
```

`--stencil=`*<Stencil File>*

> Specifies the stencil specification file for which to generate code.

`--strategy=`*<Strategy File>*

> The Strategy file describing the parallelization/optimization strategy.

`--architecture=`*<Architecture Description File>*,*<Hardware Name>*

The architecture description file and the name of the selected architecture (as specified in the `name` attribute of the `architectureType` element).

`--outdir=`<*Output Directory*>

The output directory into which the generated files will be written. Optional; if not specified the generated files will be created in the current directory.

`--generate=`<*Target*>

The target that will be generated. <*Target*> can be one of:

- `benchmark`
  This will generate a full benchmark harness. This is the default setting.
- `kernel`
  This will only generate the kernel file.

`--kernel-file=`<*Kernel Output File Name*>

Specifies the name of the C source file to which the generated kernel is written. The suffix is appended or replaced from the definition in the hardware architecture description.
Defaults to `kernel.c`.

`--compatibility=`{`C` | `Fortran`}

Selects whether the generated code has to be compatible with Fortran (Omits the double pointer output type in the kernel declaration; therefore multiple time steps are not supported.)
Defaults to `C`.

`--unroll=`<*UnrollFactor1*>,<*UnrollFactor2*>,...

A list of unrolling factors applied to the inner most loop nest containing the stencil computation. The unrolling factors are applied to all the dimensions.

`--use-native-simd-datatypes=`{`yes` | `no`}

Specifies whether the native SSE data type is to be used in the kernel signature. If set to `yes`, this also requires that the fields are padded correctly in unit stride direction.
Defaults to `no`.

`--create-validation=`{`yes` | `no`}

Specifies whether to create code that will validate the result. If *<Target>* is not `benchmark`, this option will be ignored.
Defaults to `yes`.

`--validation-tolerance=`*<Tolerance>*

Sets the tolerance for the relative error in the validation. This option is only relevant if validation code is generated (`--create-validation=yes`).
Defaults to `yes`.

`--debug=`*<DebugOption1>*,[*<DebugOption2>*,[...,[*<DebugOptionN>*]...]]

Specifies debug options (as a comma-separated list) that will influence the code generator. Valid debug options (for *<DebugOptionI>*, $I = 1, \ldots, N$) are:

- `print-stencil-indices`
  This will insert a `printf` statement for every stencil calculation with the index into the grid array at which the result is written.
- `print-validation-errors`
  Prints all values if the validation fails. The option is ignored if no validation code is generated.

A.2 Auto-Tuning

The PATUS auto-tuner is invoked on the command line like so:

```
java -jar patus.jar autotune
   <Executable Filename>
   <Param1 >  <Param2 >  . . .  <ParamN >
   [ <Constraint1>  <Constraint2>  . . .  <ConstraintM> ]
   [ -m<Method> ]
```

<Executable Filename> is the path to the file name of the benchmark executable. The benchmark executable must expose the tunable parameters as command line parameters. The PATUS auto-tuner only generates numerical parameter values. If the benchmark executable requires strings, these must be mapped from numerical values internally in the benchmarking program.

The parameters *ParamI*, $I = 1, \dots, N$, define integer parameter ranges and have the following syntax:

<StartValue>`:[[*]`*<Step>*`:]`*<EndValue>*`[!]`

or

<Value1>`[,`*<Value2>*`[,`*<Value3>*`...]][!]`

The first version, when no asterisk $\star$ in the *<Step>* is specified, enumerates all values in

$$\{a := \text{<\textit{StartValue}>} + k \cdot \text{<\textit{Step}>} : k \in \mathbb{N}_0 \text{ and } a \leqslant \text{<\textit{EndValue}>}\}.$$

If no *<Step>* is given, it defaults to 1. If there is an asterisk $\star$ in front of the *<Step>*, the values enumerated are

$$\{a := \text{<\textit{StartValue}>} \cdot \text{<\textit{Step}>}^k : k \in \mathbb{N}_0 \text{ and } a \leqslant \text{<\textit{EndValue}>}\}.$$

In the second version, all the comma-separated values *<Value1>*, *<Value2>*, ... are enumerated.

If the optional `!` is appended to the value range specification, each of the specified values is guaranteed to be used, i.e., an exhaustive search is used for the corresponding parameter.

Example A.1: *Specifying parameter ranges.*

`1:10`	enumerates all the integer numbers between and including 1 and 10.
`2:2:41`	enumerates the integers $2, 4, 6, \dots, 38, 40$.
`1:*2:128`	enumerates some powers of 2, namely 1, 2, 4, 8, 16, 32, 64, 128.

Optional constraints can be specified to restrict the parameter values. The syntax for the constraints is

`C`*<ComparisonExpression>*

where *<ComparisonExpression>* is an expression which can contain arithmetic operators `+`, `-`, `*`, and `/`, as well as comparison operators `<`, `<=`, `==`, `>=`, `>`, `!=`, and variables `$1`, …, `$`N, which correspond to the parameters *<Param1 >*, … *<ParamN >*.

Example A.2: *Constraints examples.*

`C$2<=$1`	forces the second parameter to be less or equal to the first.
`C($2+$1-1)/$1>=24`	forces $\frac{\$2+\$1-1}{\$1} = \left\lceil \frac{\$2}{\$1} \right\rceil \geqslant 24$.

The PATUS auto-tuner supports a range of search methods. The method can be selected by `-m`*<Method>* where *<Method>* is one of

- `ch.unibas.cs.hpwc.patus.autotuner.DiRectOptimizer`
  DIRECT method
- `ch.unibas.cs.hpwc.patus.autotuner.ExhaustiveSearchOptimizer`
  exhaustive search
- `ch.unibas.cs.hpwc.patus.autotuner.GeneralCombinedEliminationOptimizer`
  general combined elimination
- `ch.unibas.cs.hpwc.patus.autotuner.GreedyOptimizer`
  greedy search
- `ch.unibas.cs.hpwc.patus.autotuner.HookeJeevesOptimizer`
  Hooke-Jeeves algorithm
- `ch.unibas.cs.hpwc.patus.autotuner.MetaHeuristicOptimizer`
  genetic algorithm
- `ch.unibas.cs.hpwc.patus.autotuner.RandomSearchOptimizer`
  draws 500 random samples

- `ch.unibas.cs.hpwc.patus.autotuner.SimplexSearchOptimizer`
  simplex search (aka Nelder-Mead method)

These are Java class paths to `IOptimizer` implementations; this allows to extend range of methods easily. Please refer to Chapter 8 for a discussion and comparison of the methods. If no method is specified, the greedy algorithm is used by default.

Appendix B

Patus Grammars

B.1 Stencil DSL Grammar

In the following, the EBNF grammar for the Patus stencil specifications syntax is given. The grayed out identifiers have not yet been specified or implemented and will be added eventually in the future.

⟨*Stencil*⟩ ::= ‘stencil’ ⟨*Identifier*⟩ ‘{’ [⟨*Options*⟩] ⟨*DomainSize*⟩ ⟨*NumIterations*⟩ ⟨*Operation*⟩ ⟨*Boundary*⟩ [⟨*Filter*⟩] [⟨*StoppingCriterion*⟩] ‘}’

⟨*DomainSize*⟩ ::= ‘**domainsize**’ ‘**=**’ ⟨*Box*⟩ ‘**;**’

⟨*NumIterations*⟩ ::= ‘**t_max**’ ‘**=**’ ⟨*IntegerExpr*⟩ ‘**;**’

⟨*Operation*⟩ ::= ‘**operation**’ ⟨*Identifier*⟩ ‘**(**’ ⟨*ParamList*⟩ ‘**)**’ ‘{’ { ⟨*Statement*⟩ } ‘}’

⟨*ParamList*⟩ ::= { ⟨*GridDecl*⟩ | ⟨*ParamDecl*⟩ }

⟨*Statement*⟩ ::= ⟨*LHS*⟩ ‘**=**’ ⟨*StencilExpr*⟩ ‘**;**’

⟨*LHS*⟩ ::= ⟨*StencilNode*⟩ | ⟨*VarDecl*⟩

⟨*StencilExpr*⟩ ::= ⟨*StencilNode*⟩ | ⟨*Identifier*⟩ | ⟨*NumberLiteral*⟩ | ⟨*FunctionCall*⟩ | (⟨*UnaryOperator*⟩ ⟨*StencilExpr*⟩) | (⟨*StencilExpr*⟩ ⟨*BinaryOperator*⟩ ⟨*StencilExpr*⟩) | ‘**(**’ ⟨*StencilExpr*⟩ ‘**)**’

⟨*StencilNode*⟩ ::= ⟨*Identifier*⟩ ‘**[**’ ⟨*SpatialCoords*⟩ [‘**;**’ ⟨*TemporalCoord*⟩] [‘**;**’ ⟨*ArrayIndices*⟩] ‘**]**’

⟨*SpatialCoords*⟩ ::= (‘**x**’ | ‘**y**’ | ‘**z**’ | ‘**u**’ | ‘**v**’ | ‘**w**’ | ‘**x**’ ⟨*IntegerLiteral*⟩) [⟨*Offset*⟩]

⟨*TemporalCoord*⟩ ::= ‘**t**’ [⟨*Offset*⟩

⟨*ArrayIndices*⟩ ::= ⟨*IntegerLiteral*⟩ { ‘**,**’ ⟨*IntegerLiteral*⟩ }

⟨*Offset*⟩ ::= ⟨*UnaryOperator*⟩ ⟨*IntegerLiteral*⟩

⟨*FunctionCall*⟩ ::= ⟨*Identifier*⟩ ‘**(**’ [⟨*StencilExpr*⟩ { ‘**,**’ ⟨*StencilExpr*⟩ }] ‘**)**’

⟨*IntegerExpr*⟩ ::= ⟨*Identifier*⟩ | ⟨*IntegerLiteral*⟩ | ⟨*FunctionCall*⟩ | (⟨*UnaryOperator*⟩ ⟨*IntegerExpr*⟩) | (⟨*IntegerExpr*⟩ ⟨*BinaryOperator*⟩ ⟨*IntegerExpr*⟩) | ‘**(**’ ⟨*IntegerExpr*⟩ ‘**)**’

⟨*VarDecl*⟩ ::= ⟨*Type*⟩ ⟨*Identifier*⟩

⟨*Box*⟩ ::= ‘**(**’ ⟨*Range*⟩ { ‘**,**’ ⟨*Range*⟩ ‘**)**’

⟨*Range*⟩ ::= ⟨*IntegerExpr*⟩ ‘**..**’ ⟨*IntegerExpr*⟩

⟨*GridDecl*⟩ ::= [⟨*Specifier*⟩] ⟨*Type*⟩ ‘**grid**’ ⟨*Identifier*⟩ [‘**(**’ ⟨*Box*⟩ ‘**)**’] [⟨*ArrayDecl*⟩]

⟨*ParamDecl*⟩ ::= ⟨*Type*⟩ ‘**param**’ ⟨*Identifier*⟩ [⟨*ArrayDecl*⟩]

⟨*ArrayDecl*⟩ ::= ‘**[**’ ⟨*IntegerLiteral*⟩ { ‘**,**’ ⟨*IntegerLiteral*⟩ } ‘**]**’

⟨*Specifier*⟩ ::= ‘**const**’

⟨*Type*⟩ ::= ‘**float**’ | ‘**double**’

⟨*UnaryOperator*⟩ ::= ‘**+**’ | ‘**-**’

⟨*BinaryOperator*⟩ ::= ‘**+**’ | ‘**-**’ | ‘*’ | ‘**/**’ | ‘^’

B.2 Strategy DSL Grammar

The following EBNF grammar specifies the PATUS Strategy syntax. Again, as the project matures, the specification might change so that yet missing aspects of parallelization and optimization methods can be specified as PATUS Strategies.

⟨*Strategy*⟩ ::= ‘**strategy**’ ⟨*Identifier*⟩ ‘**(**’ ⟨*ParamList*⟩ ‘**)**’ ⟨*CompoundStatement*⟩

⟨*ParamList*⟩ ::= ⟨*SubdomainParam*⟩ { ‘**,**’ ⟨*AutoTunerParam*⟩ }

⟨*SubdomainParam*⟩ ::= ‘**domain**’ ⟨*Identifier*⟩

⟨*AutoTunerParam*⟩ ::= ‘**auto**’ ⟨*AutoTunerDeclSpec*⟩ ⟨*Identifier*⟩

⟨*AutoTunerDeclSpec*⟩ ::= ‘**int**’ | ‘**dim**’ | (‘**codim**’ ‘**(**’ ⟨*IntegerLiteral*⟩ ‘**)**’)

⟨*Statement*⟩ ::= ⟨*DeclarationStatement*⟩ | ⟨*AssignmentStatement*⟩ | ⟨*CompoundStatement*⟩ | ⟨*IfStatement*⟩ | ⟨*Loop*⟩

⟨*DeclarationStatement*⟩ ::= ⟨*DeclSpec*⟩ ⟨*Identifier*⟩ ‘**;**’

⟨*AssignmentStatement*⟩ ::= ⟨*LValue*⟩ ‘**=**’ ⟨*Expr*⟩ ‘**;**’

⟨*CompoundStatement*⟩ ::= ‘{’ { ⟨*Statement*⟩ } ‘}’

⟨*IfStatement*⟩ ::= ‘**if**’ ‘**(**’ ⟨*ConditionExpr*⟩ ‘**)**’ ⟨*Statement*⟩ [‘**else**’ ⟨*Statement*⟩]

⟨*Loop*⟩ ::= (⟨*RangeIterator*⟩ | ⟨*SubdomainIterator*⟩) [‘**parallel**’ [⟨*IntegerLiteral*⟩] [‘**schedule**’ ⟨*IntegerLiteral*⟩]] ⟨*Statement*⟩

⟨*RangeIterator*⟩ ::= ‘**for**’ ⟨*Identifier*⟩ ‘**=**’ ⟨*Expr*⟩ ‘**..**’ ⟨*Expr*⟩ [‘**by**’ ⟨*Expr*⟩]

⟨*SubdomainIterator*⟩ ::= ‘**for**’ ⟨*SubdomainIteratorDecl*⟩ ‘**in**’ ⟨*Identifier*⟩ ‘**(**’ ⟨*Range*⟩ ‘**;**’ ⟨*Expr*⟩ ‘**)**’

⟨*SubdomainIteratorDecl*⟩ ::= ⟨*PointDecl*⟩ | ⟨*PlaneDecl*⟩ | ⟨*SubdomainDecl*⟩

⟨*PointDecl*⟩ ::= ‘**point**’ ⟨*Identifier*⟩

⟨*PlaneDecl*⟩ ::= ‘**plane**’ ⟨*Identifier*⟩

⟨*SubdomainDecl*⟩ ::= ‘**subdomain**’ ⟨*Identifier*⟩ ‘**(**’ ⟨*Range*⟩ ‘**)**’

⟨*Range*⟩ ::= ⟨*Vector*⟩ { ⟨*UnaryOperator*⟩ ⟨*ScaledBorder*⟩ }

⟨*Vector*⟩ ::= ⟨*Subvector*⟩ [‘**...**’ [‘**,**’ ⟨*Subvector*⟩]]

⟨*Subvector*⟩ ::= (‘**:**’ { ‘**,**’ ⟨*ScalarList*⟩ }) | ⟨*DimensionIdentifier*⟩ | ⟨*DomainSizeExpr*⟩ | ⟨*BracketedVector*⟩ | ⟨*ScalarList*⟩

⟨*ScalarList*⟩ ::= ⟨*ScalarRange*⟩ { ‘**,**’ ⟨*ScalarRange*⟩ }

⟨*DimensionIdentifier*⟩ ::= ⟨*Expr*⟩ [‘**(**’ ⟨*Vector*⟩ ‘**)**’]

⟨*DomainSizeExpr*⟩ ::= ⟨*SizeProperty*⟩ [‘**(**’ ⟨*Vector*⟩ ‘**)**’]

⟨*BracketedVector*⟩ ::= ‘**(**’ ⟨*Vector*⟩ { ‘**,**’ ⟨*Vector*⟩ } ‘**)**’

⟨*ScalarRange*⟩ ::= ⟨*Expr*⟩ [‘**..**’ ⟨*Expr*⟩]

⟨*SizeProperty*⟩ ::= (‘**stencil**’ | ⟨*Identifier*⟩) ‘**.**’ (‘**size**’ | ‘**min**’ | ‘**max**’)

⟨*ScaledBorder*⟩ ::= [⟨*Expr*⟩ ⟨*MultiplicativeOperator*⟩] ⟨*Border*⟩ [⟨*MultiplicativeOperator*⟩ ⟨*Expr*⟩]

⟨*Border*⟩ ::= ⟨*StencilBoxBorder*⟩ | ⟨*LiteralBorder*⟩

⟨*StencilBoxBorder*⟩ ::= ‘**stencil**’ ‘**.**’ ‘**box**’ [‘**(**’ ⟨*Vector*⟩ ‘**)**’]

⟨*LiteralBorder*⟩ ::= ‘**(**’ ⟨*Vector*⟩ ‘**)**’ ‘**,**’ ‘**(**’ ⟨*Vector*⟩ ‘**)**’

⟨*LValue*⟩ ::= ⟨*GridAccess*⟩ | ⟨*Identifier*⟩

⟨*GridAccess*⟩ ::= ⟨*Identifier*⟩ ‘**[**’ ⟨*SpatialIndex*⟩ ‘**;**’ ⟨*Expr*⟩ { ‘**;**’ ⟨*Expr*⟩ } ‘**]**’

⟨*SpatialIndex*⟩ ::= ⟨*Identifier*⟩ | ⟨*Range*⟩

⟨*Expr*⟩ ::= ⟨*Identifier*⟩ | ⟨*NumberLiteral*⟩ | ⟨*FunctionCall*⟩ | (⟨*UnaryOperator*⟩ ⟨*Expr*⟩) | (⟨*Expr*⟩ ⟨*BinaryOperator*⟩ ⟨*Expr*⟩) | ‘**(**’ ⟨*Expr*⟩ ‘**)**’

⟨*FunctionCall*⟩ ::= ⟨*Identifier*⟩ ‘**(**’ [⟨*Expr*⟩ { ‘**,**’ ⟨*Expr*⟩ }] ‘**)**’

⟨*ConditionExpr*⟩ ::= ⟨*ComparisonExpr*⟩ | (⟨*ConditionExpr*⟩
⟨*LogicalOperator*⟩ ⟨*ConditionExpr*⟩)

⟨*ComparisonExpr*⟩ ::= ⟨*Expr*⟩ ⟨*ComparisonOperator*⟩ ⟨*Expr*⟩

⟨*UnaryOperator*⟩ ::= ‘**+**’ | ‘**-**’

⟨*MultiplicativeOperator*⟩ ::= ‘*’

⟨*BinaryOperator*⟩ ::= ‘**+**’ | ‘**-**’ | ‘*’ | ‘**/**’ | ‘**%**’

⟨*LogicalOperator*⟩ ::= ‘||’ | ‘**&&**’

⟨*ComparisonOperator*⟩ ::= ‘<’ | ‘**<=**’ | ‘**==**’ | ‘**>=**’ | ‘>’ | ‘**!=**’

Appendix C

Stencil Specifications

C.1 Basic Differential Operators

C.1.1 Laplacian

```
stencil laplacian
{
  domainsize = (1 .. N, 1 .. N, 1 .. N);
  t_max = 1;

  operation (float grid u, float param alpha, float param beta)
  {
    u[x, y, z; t+1] =
      alpha * u[x, y, z; t] +
      beta  * (
        u[x+1, y, z; t] + u[x-1, y, z; t] +
        u[x, y+1, z; t] + u[x, y-1, z; t] +
        u[x, y, z+1; t] + u[x, y, z-1; t]);
  }
}
```

C.1.2 Divergence

```
stencil divergence
{
  domainsize = (1 .. x_max, 1 .. y_max, 1 .. z_max);
  t_max = 1;

  operation (
    float grid u(0 .. x_max+1, 0 .. y_max+1, 0 .. z_max+1),
```

```
    const float grid ux(0 .. x_max+1, 0 .. y_max+1, 0 .. z_max+1),
    const float grid uy(0 .. x_max+1, 0 .. y_max+1, 0 .. z_max+1),
    const float grid uz(0 .. x_max+1, 0 .. y_max+1, 0 .. z_max+1),
    float param alpha, float param beta, float param gamma)
  {
    u[x, y, z; t] =
      alpha * (ux[x+1, y, z] - ux[x-1, y, z]) +
      beta  * (uy[x, y+1, z] - uy[x, y-1, z]) +
      gamma * (uz[x, y, z+1] - uz[x, y, z-1]);
  }
}
```

C.1.3 Gradient

```
stencil gradient
{
  domainsize = (1 .. x_max, 1 .. y_max, 1 .. z_max);
  t_max = 1;

  operation (
    const float grid u(0 .. x_max+1, 0 .. y_max+1, 0 .. z_max+1),
    float grid ux(0 .. x_max+1, 0 .. y_max+1, 0 .. z_max+1),
    float grid uy(0 .. x_max+1, 0 .. y_max+1, 0 .. z_max+1),
    float grid uz(0 .. x_max+1, 0 .. y_max+1, 0 .. z_max+1),
    float param alpha, float param beta, float param gamma)
  {
    ux[x, y, z; t] = alpha * (u[x+1, y, z] + u[x-1, y, z]);
    uy[x, y, z; t] = beta  * (u[x, y+1, z] + u[x, y-1, z]);
    uz[x, y, z; t] = gamma * (u[x, y, z+1] + u[x, y, z-1]);
  }
}
```

C.2 Wave Equation

```
stencil wave
{
  domainsize = (1 .. N, 1 .. N, 1 .. N);
  t_max = 100;

  operation (float grid u, float param c2dt_h2)
  {
    u[x, y, z; t+1] = 2 * u[x, y, z; t] - u[x, y, z; t-1] +
      c2dt_h2 * (
        -15/2 * u[x, y, z; t] +
        4/3 * (
          u[x+1, y, z; t] + u[x-1, y, z; t] +
```

```
          u[x, y+1, z; t] + u[x, y-1, z; t] +
          u[x, y, z+1; t] + u[x, y, z-1; t]
        )
        -1/12 * (
          u[x+2, y, z; t] + u[x-2, y, z; t] +
          u[x, y+2, z; t] + u[x, y-2, z; t] +
          u[x, y, z+2; t] + u[x, y, z-2; t]
        )
      );
  }
}
```

C.3 COSMO

C.3.1 Upstream

```
stencil upstream_5_3d
{
  domainsize = (1 .. x_max, 1 .. y_max, 1 .. z_max);
  t_max = 1;

  operation (double grid u, double param a)
  {
    u[x, y, z; t+1] = a * (
       -2 * (u[x-3, y, z; t] + u[x, y-3, z; t] + u[x, y, z-3; t]) +
       15 * (u[x-2, y, z; t] + u[x, y-2, z; t] + u[x, y, z-2; t]) +
      -60 * (u[x-1, y, z; t] + u[x, y-1, z; t] + u[x, y, z-1; t]) +
       20 *  u[x,   y, z; t] +
       30 * (u[x+1, y, z; t] + u[x, y+1, z; t] + u[x, y, z+1; t]) +
       -3 * (u[x+2, y, z; t] + u[x, y+2, z; t] + u[x, y, z+2; t]);
  }
}
```

C.3.2 Tricubic Interpolation

```
stencil tricubic_interpolation
{
  domainsize = (1 .. x_max, 1 .. y_max, 1 .. z_max);
  t_max = 1;

  operation (double grid u,
    const double grid a, const double grid b, const double grid c)
  {
    double w1_a = 1.0/6.0*a[x,y,z]*(a[x,y,z]+1.0)*(a[x,y,z]+2.0);
    double w2_a = -0.5*(a[x,y,z]-1.0)*(a[x,y,z]+1.0)*(a[x,y,z]+2.0);
    double w3_a =  0.5*(a[x,y,z]-1.0)*a[x,y,z]*(a[x,y,z]+2.0);
```

```
    double w4_a = -1.0/6.0*(a[x,y,z]-1.0)*a[x,y,z]*(a[x,y,z]+1.0);

    double w1_b = 1.0/6.0*b[x,y,z]*(b[x,y,z]+1.0)*(b[x,y,z]+2.0);
    double w2_b = -0.5*(b[x,y,z]-1.0)*(b[x,y,z]+1.0)*(b[x,y,z]+2.0);
    double w3_b =  0.5*(b[x,y,z]-1.0)*b[x,y,z]*(b[x,y,z]+2.0);
    double w4_b = -1.0/6.0*(b[x,y,z]-1.0)*b[x,y,z]*(b[x,y,z]+1.0);

    double w1_c = 1.0/6.0*c[x,y,z]*(c[x,y,z]+1.0)*(c[x,y,z]+2.0);
    double w2_c = -0.5*(c[x,y,z]-1.0)*(c[x,y,z]+1.0)*(c[x,y,z]+2.0);
    double w3_c =  0.5*(c[x,y,z]-1.0)*c[x,y,z]*(c[x,y,z]+2.0);
    double w4_c = -1.0/6.0*(c[x,y,z]-1.0)*c[x,y,z]*(c[x,y,z]+1.0);

    u[x, y, z; t+1] =
      w1_a * w1_b * w1_c * u[x-1, y-1, z-1; t] +
      w2_a * w1_b * w1_c * u[x,   y-1, z-1; t] +
      w3_a * w1_b * w1_c * u[x+1, y-1, z-1; t] +
      w4_a * w1_b * w1_c * u[x+2, y-1, z-1; t] +
      ... // etc. for all 64 combinations of w?_a * w?_b * w?_c
          // and u[x+d1, y+d2, z+d3; t],  d1,d2,d3=-1,0,1,2
  }
}
```

C.4 Hyperthermia

```
stencil hyperthermia
{
  domainsize = (1 .. x_max, 1 .. y_max, 1 .. z_max);
  t_max = 1;

  operation (
    float grid T(0 .. x_max+1, 0 .. y_max+1, 0 .. z_max+1),
    const float grid c(0 .. x_max+1, 0 .. y_max+1, 0 .. z_max+1)[9])
  {
    T[x, y, z; t+1] =

      // center point
      T[x, y, z; t] * (c[x, y, z; 0] * T[x, y, z; t] + c[x, y, z; 1]) +
      c[x, y, z; 2] +

      // faces
      c[x, y, z; 3] * T[x-1, y, z; t] +
      c[x, y, z; 4] * T[x+1, y, z; t] +
      c[x, y, z; 5] * T[x, y-1, z; t] +
      c[x, y, z; 6] * T[x, y+1, z; t] +
      c[x, y, z; 7] * T[x, y, z-1; t] +
      c[x, y, z; 8] * T[x, y, z+1; t];
```

```
  }
}
```

C.5 Image Processing

C.5.1 Blur Kernel

```
stencil blur
{
  domainsize = (1 .. width, 1 .. height);
  t_max = 1;

  operation (float grid u, float param sigma)
  {
    float f0 = 1 / (2 * sigma ^ 2);
    float s0 = exp ( 0 * f0);
    float s1 = exp (-1 * f0);
    float s2 = exp (-2 * f0);
    float s4 = exp (-4 * f0);
    float s5 = exp (-5 * f0);
    float s8 = exp (-8 * f0);
    float f = 1 / (s0 + 4 * (s1 + s2 + s4 + s8) + 8 * s5);

    u[x, y; t+1] = f * (
      s0 * u[x, y; t] +
      s1 * (u[x - 1, y; t] + u[x + 1, y; t] +
          u[x, y - 1; t] + u[x, y + 1; t]) +
      s2 * (u[x - 1, y - 1; t] + u[x + 1, y - 1; t] +
          u[x - 1, y + 1; t] + u[x + 1, y + 1; t]) +
      s4 * (u[x - 2, y; t] + u[x + 2, y; t] +
          u[x, y - 2; t] + u[x, y + 2; t]) +
      s5 * (
        u[x - 2, y - 1; t] + u[x - 1, y - 2; t] +
          u[x + 1, y - 2; t] + u[x + 2, y - 1; t] +
        u[x - 2, y + 1; t] + u[x - 1, y + 2; t] +
          u[x + 1, y + 2; t] + u[x + 2, y + 1; t]
      ) +
      s8 * (u[x - 2, y - 2; t] + u[x + 2, y - 2; t] +
          u[x - 2, y + 2; t] + u[x + 2, y + 2; t])
    );
  }
}
```

C.5.2 Edge Detection

```
stencil edge
```

```
{
  domainsize = (1 .. width, 1 .. height);
  t_max = 1;

  operation (float grid u)
  {
    u[x, y; t+1] =
      -12 * u[x, y; t] +
        2 *(u[x - 1, y; t] + u[x + 1, y; t] +
            u[x, y - 1; t] + u[x, y + 1; t]) +
        u[x - 1, y - 1; t] + u[x + 1, y - 1; t] +
          u[x - 1, y + 1; t] + u[x + 1, y + 1; t];
  }
}
```

C.6 Cellular Automata

C.6.1 Conway's Game of Life

```
stencil game_of_life
{
  domainsize = (1 .. width, 1 .. height);
  t_max = 1;

  operation (float grid u)
  {
    // some large number
    float C = 100000000000000000000;

    // count the number of live neighbors
    float L =
      u[x - 1, y - 1; t] + u[x, y - 1; t] + u[x + 1, y - 1; t] +
      u[x - 1, y    ; t] +                  u[x + 1, y    ; t] +
      u[x - 1, y + 1; t] + u[x, y + 1; t] + u[x + 1, y + 1; t];

    // apply the rules
    u[x, y; t+1] = 1 / (1 + (u[x, y; t] + L - 3) * (L - 3) * C);
  }
}
```

C.7 Anelastic Wave Propagation

C.7.1 uxx1

```
stencil pmcl3d_uxx1
{
```

```
  domainsize = (nxb .. nxe, nyb .. nye, nzb .. nze);
  t_max = 1;

  operation (
    const float grid d1(−1 .. nxt+2, −1 .. nyt+2, −1 .. nzt+2),
    float grid u1(−1 .. nxt+2, −1 .. nyt+2, −1 .. nzt+2),
    const float grid xx(−1 .. nxt+2, −1 .. nyt+2, −1 .. nzt+2),
    const float grid xy(−1 .. nxt+2, −1 .. nyt+2, −1 .. nzt+2),
    const float grid xz(−1 .. nxt+2, −1 .. nyt+2, −1 .. nzt+2),
    float param dth)
  {
    float c1 = 9./8.;
    float c2 = −1./24.;

    float d = 0.25 *
      (d1[x,y,z] + d1[x,y−1,z] + d1[x,y,z−1] + d1[x,y−1,z−1]);
    u1[x,y,z; t+1] = u1[x,y,z; t] + (dth / d) * (
      c1 * (
        xx[x,y,z] − xx[x−1,y,z] +
        xy[x,y,z] − xy[x,y−1,z] +
        xz[x,y,z] − xz[x,y,z−1]) +
      c2 * (
        xx[x+1,y,z] − xx[x−2,y,z] +
        xy[x,y+1,z] − xy[x,y−2,z] +
        xz[x,y,z+1] − xz[x,y,z−2])
    );
  }
}
```

C.7.2 xy1

```
stencil pmcl3d_xy1
{
  domainsize = (nxb .. nxe, nyb .. nye, nzb .. nze);
  t_max = 1;

  operation (
    const float grid mu(−1 .. nxt+2, −1 .. nyt+2, −1 .. nzt+2),
    float grid xy(−1 .. nxt+2, −1 .. nyt+2, −1 .. nzt+2),
    const float grid u1(−1 .. nxt+2, −1 .. nyt+2, −1 .. nzt+2),
    const float grid v1(−1 .. nxt+2, −1 .. nyt+2, −1 .. nzt+2),
    float param dth)
  {
    float c1 = 9. / 8.;
    float c2 = −1. / 24.;

    float xmu = 2. / (1. / mu[x, y, z] + 1. / mu[x, y, z−1]);
```

```

    xy[x, y, z; t+1] = xy[x, y, z; t] + dth * xmu * (
      c1 * (
        u1[x, y+1, z] - u1[x,   y, z] +
        v1[x, y,   z] - v1[x-1, y, z]
      ) +
      c2 * (
        u1[x,   y+2, z] - u1[x,   y-1, z] +
        v1[x+1, y,   z] - v1[x-2, y,   z]
      )
    );
  }
}
```

C.7.3 xyz1

```
stencil pmcl3d_xyz1
{
  domainsize = (nxb .. nxe, nyb .. nye, nzb .. nze);
  t_max = 1;

  operation (
    const float grid mu(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    const float grid lam(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    const float grid u1(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    const float grid v1(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    const float grid w1(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    float grid xx(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    float grid yy(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    float grid zz(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    float param dth)
  {
    float c1 = 9./8.;
    float c2 = -1./24.;

    float b = 8. / (
      1. / lam[x, y,   z  ] + 1. / lam[x+1, y,   z  ] +
      1. / lam[x, y-1, z  ] + 1. / lam[x+1, y-1, z  ] +
      1. / lam[x, y,   z-1] + 1. / lam[x+1, y,   z-1] +
      1. / lam[x, y-1, z-1] + 1. / lam[x+1, y-1, z-1]
    );

    float a = b + 2. * 8. / (
      1. / mu[x, y,   z  ] + 1. / mu[x+1, y,   z  ] +
      1. / mu[x, y-1, z  ] + 1. / mu[x+1, y-1, z  ] +
      1. / mu[x, y,   z-1] + 1. / mu[x+1, y,   z-1] +
      1. / mu[x, y-1, z-1] + 1. / mu[x+1, y-1, z-1]
```

```
);

// find xx stress
xx[x, y, z; t+1] = xx[x, y, z; t] + dth * (
  a * (
    c1 * (u1[x+1, y, z] - u1[x,   y, z]) +
    c2 * (u1[x+2, y, z] - u1[x-1, y, z])
  ) +
  b * (
    c1 * (
      v1[x, y, z] - v1[x, y-1, z  ] +
      w1[x, y, z] - w1[x, y,   z-1]
    ) +
    c2 * (
      v1[x, y+1, z  ] - v1[x, y-2, z  ] +
      w1[x, y,   z+1] - w1[x, y,   z-2]
    )
  )
);

// find yy stress
yy[x, y, z; t+1] = yy[x, y, z; t] + dth * (
  a * (
    c1 * (v1[x, y,   z] - v1[x, y-1, z]) +
    c2 * (v1[x, y+1, z] - v1[x, y-2, z])
  ) +
  b * (
    c1 * (
      u1[x+1, y, z] - u1[x, y, z  ] +
      w1[x,   y, z] - w1[x, y, z-1]
    ) +
    c2 * (
      u1[x+2, y, z  ] - u1[x-1, y, z  ] +
      w1[x,   y, z+1] - w1[x,   y, z-2]
    )
  )
);

// find zz stress
zz[x, y, z; t+1] = zz[x, y, z; t] + dth * (
  a * (
    c1 * (w1[x, y, z  ] - w1[x, y, z-1]) +
    c2 * (w1[x, y, z+1] - w1[x, y, z-2])
  ) +
  b * (
    c1 * (
      u1[x+1, y, z] - u1[x, y,   z] +
```

```
          v1[x,   y, z] - v1[x, y-1, z]
        ) +
        c2 * (
          u1[x+2, y,   z] - u1[x-1, y,   z] +
          v1[x,   y+1, z] - v1[x,   y-2, z]
        )
      )
    );
  }
}
```

C.7.4 xyzq

```
stencil pmcl3d_xyzq
{
  domainsize = (nxb .. nxe, nyb .. nye, nzb .. nze);
  t_max = 1;

  operation (
    const float grid mu(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    const float grid lam(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    float grid r1(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    float grid r2(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    float grid r3(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    float grid xx(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    float grid yy(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    float grid zz(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    const float grid u1(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    const float grid v1(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    const float grid w1(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    const float grid qp(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    const float grid qs(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    const float grid tau(-1 .. nxt+2, -1 .. nyt+2, -1 .. nzt+2),
    float param dh, float param dt, float param dth,
    float param nz)
  {
    float c1 = 9./8.;
    float c2 = -1./24.;

    float d = 8. / (
      1. / mu[x, y,   z  ] + 1. / mu[x+1, y,   z  ] +
      1. / mu[x, y-1, z  ] + 1. / mu[x+1, y-1, z  ] +
      1. / mu[x, y,   z-1] + 1. / mu[x+1, y,   z-1] +
      1. / mu[x, y-1, z-1] + 1. / mu[x+1, y-1, z-1]
    );

    float a2 = 2 * d;
```

```
    float c = a2 + 8. / (
      1. / lam[x, y,   z  ] + 1. / lam[x+1, y,   z  ] +
      1. / lam[x, y-1, z  ] + 1. / lam[x+1, y-1, z  ] +
      1. / lam[x, y,   z-1] + 1. / lam[x+1, y,   z-1] +
      1. / lam[x, y-1, z-1] + 1. / lam[x+1, y-1, z-1]
    );

    float qpa = 0.125 * (
      qp[x, y,   z  ] + qp[x+1, y,   z  ] +
      qp[x, y-1, z  ] + qp[x+1, y-1, z  ] +
      qp[x, y,   z-1] + qp[x+1, y,   z-1] +
      qp[x, y-1, z-1] + qp[x+1, y-1, z-1]
    );

    float qsa = 0.125 * (
      qs[x, y,   z  ] + qs[x+1, y,   z  ] +
      qs[x, y-1, z  ] + qs[x+1, y-1, z  ] +
      qs[x, y,   z-1] + qs[x+1, y,   z-1] +
      qs[x, y-1, z-1] + qs[x+1, y-1, z-1]
    );

    // (we can't handle indirect grid accesses for the time being)
    // float tauu = tau[((coords1 * nxt + this.x) % 2) +
    //     2 * ((coords2 * nyt + this.y) % 2) +
    //     4 * ((nz + 1 - (coords3 * nzt + this.z)) % 2)];

    float vxx = c1 * (u1[x+1, y, z] - u1[x, y, z]) +
      c2 * (u1[x+2, y, z] - u1[x-1, y, z]);
    float vyy = c1 * (v1[x, y, z] - v1[x, y-1, z]) +
      c2 * (v1[x, y+1, z] - v1[x, y-2, z]);
    float vzz = c1 * (w1[x, y, z] - w1[x, y, z-1]) +
      c2 * (w1[x, y, z+1] - w1[x, y, z-2]);

    float a1 = -qpa * c * (vxx + vyy + vzz) / (2. * dh);

    //float x1 = tauu / dt + 0.5;
    float x1 = tau[x, y, z] / dt + 0.5;
    float x2 = x1 - 1; // (tauu/dt)-(1./2.)

    // normal stress xx, yy and zz
    xx[x, y, z; t+1] = xx[x, y, z; t] + dth * (c * vxx + (c - a2) *
      (vyy + vzz)) + dt * r1[x, y, z; t];
    yy[x, y, z; t+1] = yy[x, y, z; t] + dth * (c * vyy + (c - a2) *
      (vxx + vzz)) + dt * r2[x, y, z; t];
    zz[x, y, z; t+1] = zz[x, y, z; t] + dth * (c * vzz + (c - a2) *
      (vxx + vyy)) + dt * r3[x, y, z; t];

```

```
        float hdh = -d * qsa / dh;
        r1[x, y, z; t+1] = (x2 * r1[x, y, z; t] - hdh*(vyy + vzz) + a1)/x1;
        r2[x, y, z; t+1] = (x2 * r2[x, y, z; t] - hdh*(vxx + vzz) + a1)/x1;
        r3[x, y, z; t+1] = (x2 * r3[x, y, z; t] - hdh*(vxx + vyy) + a1)/x1;

        xx[x, y, z; t+1] = xx[x, y, z; t+1] + dt * r1[x, y, z; t+1];
        yy[x, y, z; t+1] = yy[x, y, z; t+1] + dt * r2[x, y, z; t+1];
        zz[x, y, z; t+1] = zz[x, y, z; t+1] + dt * r3[x, y, z; t+1];
    }
}
```

Index

Curriculum Vitae

Personal Data

Name	Matthias-Michael Christen
Date of birth	April 17, 1980, Zürich, Switzerland
Parents	Sophie and Walter Christen-Tchung
Nationality	Swiss
Citizenship	Affoltern BE, Switzerland
Family Status	unmarried

Education

1987 – 1999	Waldorf School in Adliswil ZH and Basel (Switzerland)
1999 – 2000	Gymnasium am Kirschgarten, Basel (Switzerland)
2000 – 2006	Diploma in Mathematics, minors in Computer Science and Musicology at the University of Basel (Switzerland)
2007 – 2011	Ph.D. candidate in Computer Science at the University of Basel (Switzerland)
Summer 2009 and Fall 2010	Internships at the Lawrence Berkeley National Laboratory, Berkeley CA (USA)
22 Sept. 2011	Ph.D. examination, University of Basel (Switzerland)

I enjoyed attending the lectures of the following professors and lecturers:

W. Arlt, H. Burkhart, N. A'Campo, D. Cohen, F.-J. Elmer, M. Grote, M. Guggisberg, M. Haas, L. Halbeisen, H.-C. Im Hof, A. Iozzi, J. Königsmann, H. Kraft, E. Lederer, Y. Lengwiler, D. Masser, D. Muller, O. Schenk, M. Schmidt, A. C. Shreffler, J. Stenzl, C. Tschudin, J. Willimann, C. Wattinger.