Performance engineering as a guiding principle for efficient implementations of algorithms in computational science

Habilitationsschrift

zur

Erlangung des akademischen Grades

doctor rerum naturalium habilitatus (Dr. rer. nat. habil.)

an der Mathematisch-Naturwissenschaftlichen Fakultät

der

Ernst-Moritz-Arndt-Universität Greifswald

vorgelegt von

Georg Hager

geboren am 21.08.1970

in Hof/Saale

Greifswald, im Oktober 2013

Dekan: .....

1. Gutachter: .....

2. Gutachter: .....

3. Gutachter: .....

Tag der Habilitation: .....

Zusammenfassung

Rechnergestützte Wissenschaften sind zusehends auf paralleles Rechnen angewiesen, um anspruchsvolle numerische Probleme lösen zu können. Die ständig steigende Leistungsfähigkeit paralleler Rechner ermöglicht es zusammen mit Fortschritten in der Algorithmik Modelle zu nutzen, die eine quantitative Beschreibung der Natur erlauben. Dennoch ist auch im Zeitalter der Petaflop-Systeme der Bedarf nach Rechenzyklen immer größer als das Angebot, und Wissenschaftler sind gezwungen, die knappen Ressourcen bestmöglich zu nutzen. Deshalb ist die Effizienz des parallelen Rechnens von entscheidender Bedeutung. Leider wird Effizienz leicht mit Skalierbarkeit verwechselt, was dann zu Problemen führt, wenn parallele Programme nur deswegen skalieren, weil die Ausführung des Codes in den Recheneinheiten so langsam ist.

Performance-Modellierung und *Performance-Engineering* auf der Ebene der Rechenknoten sind die Hauptthemen dieser Arbeit. Performance Engineering wird als Prozess verstanden, der ein tieferes Verständnis der Wechselwirkung von Hardware mit Software ermöglicht. Dies führt zu einem wohldefinierten Konzept von „bestmöglicher Performance“. Zu diesem Zweck verwendet Performance-Engineering *ressourcengetriebene Performancemodelle*, um die Laufzeit des Codes und den Nutzen von Optimierungen vorherzusagen. Ein Performancemodell basiert auf einem vereinfachten Maschinenmodell, das die wichtigsten Elemente einer Rechnerarchitektur umfasst.

Diese Arbeit beginnt mit einem Überblick über die Architektur moderner multicore-Prozessoren und Rechenknoten, soweit sie relevant für die ressourcengetriebene Performancemodellierung ist. Nach einer Einführung in das bekannte Roofline-Modell wird das „Execution-Cache-Memory-Modell“ (ECM-Modell) präsentiert, das als Verfeinerung des Roofline-Modells für multicore-CPUs gesehen werden kann. Das ECM-Modell ist insbesondere fähig, die Einzelkern-Performance und die Skalierung von Schleifenkonstrukten mit kontinuierlichem Speicherzugriff über die Rechenkerne eines Chips vorherzusagen.

Da die Leistungsaufnahme von Großrechnern und damit der Energieverbrauch der darauf genutzten Codes wegen der steigenden Kosten für die Infrastruktur immer mehr in den Fokus rückt, werden die Eigenschaften paralleler Programme im Hinblick auf diese Faktoren in naher Zukunft von zentraler Bedeutung sein. Die Frage, mit welchen Mitteln Energie gespart werden kann, ohne die Performance zu kompromittieren, kann mit Hilfe von *Energiemodellen* geklärt werden. Dazu wird ein phänomenologisches Energiemodell für multicore-Chips eingeführt, das den optimalen Arbeitspunkt in Bezug auf die Anzahl genutzter Kerne und die Taktfrequenz vorhersagen kann. Als wichtigste Einflussgröße geht dabei jedoch die Performance des Codes ein. Folglich bekommt das „race to idle“-Konzept im Rahmen des Energiemodells eine Doppelbedeutung: Energie kann sowohl durch effizienten Code als auch durch hohe Taktfrequenz gespart werden. Ersteres funktioniert immer, letzteres nur in einem bestimmten, durch statische und dynamische Leistungsaufnahme definierten Bereich.

Performance- und Energiemodelle können bei der Entwicklung effizienter Software behilflich sein, sie sollten jedoch in einen wohlstrukturierten Prozess eingebettet werden, der die Komplexität der Hardware-Software-Wechselwirkung handhabbar macht. *Mustergeleitetes Performance-Engineering* ist solch ein Prozess. Dazu wird eine Anzahl vorherrschender Performancemuster auf Knotenebene identifiziert und auf Signaturen abgebildet, die jeweils aus einem beobachtbaren Performanceverhalten und einer Kombination von Hardware-Metriken besteht. Diese Muster werden dann zur Konstruktion von Performancemodellen eingesetzt, die durch

Messungen bestätigt oder widerlegt werden. Da es sich um einen iterativen Prozess handelt, führt ein widerlegtes Modell zu neuen Einsichten, wenn ein neues Muster ausgewählt und/oder ein neues Modell konstruiert wird. Andererseits hat ein erfolgreiches Modell das Potenzial zur Vorhersage des möglichen Nutzens von Programmoptimierungen. Dadurch wird die Methode „Versuch und Irrtum“ durch ein wohlbegründetes Vorgehen ersetzt.

Abstract

Computational science relies heavily on parallel computing to solve challenging numerical problems. The ever-increasing performance of parallel computers together with algorithmic advances enables high-performance software to use models that provide quantitative descriptions of natural phenomena. However, the demand for compute cycles is always larger than the supply even in the petascale era, so scientists are hard-pressed to make best use of the scarce resources. This is why the *efficiency* of parallel computing is paramount. Unfortunately, efficiency is often confused with *scalability*, which is problematic since it is easy to make a parallel program scale by slowing down its code execution in the processors and compute nodes doing the actual work.

Performance modeling and performance engineering approaches on the node level are the main topics of this work. *Performance engineering* is understood as a process that helps in developing a thorough understanding of the interactions between software and hardware, leading to a well-defined concept of "best performance." To this end, performance engineering builds on *resource-driven performance models* to predict the runtime of code and the benefit of optimizations. Performance models are based on simplified machine models, which encompass the key features of a computer architecture.

This treatise starts with giving an overview on processor and node architecture, as far as it is relevant for resource-driven performance modeling. After an introduction to the well-known roofline model, the execution-cache-memory (ECM) performance model is presented as a refinement of the roofline model that is especially useful in predicting the single-core performance and multicore scalability of streaming loop kernels.

Since the power dissipation of computer systems, and hence the energy consumption of running programs, is gaining attention due to growing costs for the infrastructure of large installations, energy-awareness will be a cardinal quality of computer code in the near future. How to save energy with minimum loss of performance is the key question, which may be answered by power modeling techniques. For this purpose a phenomenological multicore power model is introduced. It can predict optimal operating points with respect to chip-level concurrency and processor clock speed for parallel code, but one of the main premises that go into the model is that code performance is the lowest-order energy-saving factor. Consequently, the *race to idle* concept has a double meaning in the model: Racing by code efficiency and racing by clock speed. While the former is always applicable, the latter only saves energy in a certain parameter range of static vs. dynamic power.

Using performance (and power) modeling is a way to learn more about efficient code execution, but such models should be embedded in a well-structured process that guides the way through the complexities of hardware-software interactions. *Pattern-driven structured performance engineering* provides such a process. A number of prevalent node-level performance patterns is defined, together with identifying signatures in performance behavior and hardware metrics. These patterns are then used to construct models, which can be validated or falsified using measurements. Since the process is iterative, a false model has the positive effect that new insights are gained as a new model is built or a new pattern is selected. A working model, on the other hand, has the potential to predict the possible gain of code optimizations, and substitutes trial-and-error by well-founded decisions.

Clarification about use of prior own work

This treatise contains new results as well as results that were previously published by the author and his co-authors. In cases where larger portions of previous work were used literally or almost literally, the section header has a reference to this prior own work. This pertains especially to the following sections and publications:

- Sect. 2.4.1 [1]
- Sect. 3.3.2 [2, 3]
- Sect. 3.4 [4, 1]
- Chapter 4 [1]
- Sect. 5.2 [5]
- Chapter 6 [6]
- Chapter 7 [7]

Contents

List of acronyms and abbreviations **5**

I Performance modeling and engineering **7**

1 Introduction **9**
1.1 Scientific computing and code optimization 9
1.2 Performance modeling 9
1.2.1 Light speed 10
1.2.2 Extrapolation 10
1.2.3 Machine model 11
1.3 Contributions 11
1.3.1 ECM Model 11
1.3.2 Multicore power model 11
1.3.3 Pattern-guided structured performance engineering on the node level . . 12
1.3.4 Applications 12
1.4 Related work in performance engineering 14
1.5 Organization of this thesis 15

2 Computer architecture **17**
2.1 Cores 17
2.1.1 Execution units and ports 17
2.1.2 Registers 19
2.1.3 SIMD execution 19
2.1.4 Instruction cache and decoders 20
2.1.5 SMT 20
2.1.6 Data cache 21
2.1.7 Clock frequency and turbo mode 21
2.2 Multicore chips 21
2.2.1 Multiple cores 21
2.2.2 Memory access 22
2.3 Node and memory architecture 22
2.4 Test bed and tools 23
2.4.1 Intel Xeon “Sandy Bridge” processor 23
2.4.2 Tools 24
2.4.3 SuperMUC 24

3 White-box performance modeling on the chip level **27**
3.1 Performance and speedup . . . 27
3.1.1 Useful performance metrics . . . 28
3.1.2 High-level scalability models . . . 28
3.2 The roofline model . . . 30
3.2.1 Building the model . . . 30
3.2.2 Model prerequisites and assumptions . . . 31
3.2.3 Model-guided code optimizations . . . 32
3.3 Examples for roofline modeling . . . 34
3.3.1 Pure streaming kernel . . . 34
3.3.2 Sparse matrix-vector multiplication . . . 36
3.3.3 Divide-accumulate kernel . . . 41
3.3.4 Conclusions and best practices for applying the roofline model . . . 42
3.4 The Execution-Cache-Memory (ECM) model: A refined performance model for streaming loop kernels on multicore . . . 43
3.4.1 The Execution-Cache-Memory (ECM) model: Single core . . . 43
3.4.2 The ECM model: Multicore scaling . . . 45
3.4.3 Validation via streaming benchmarks . . . 46
3.4.4 Conclusions and best practices for applying the ECM model . . . 50
3.5 Chapter summary . . . 51

4 Performance and power **53**
4.1 Power dissipation and performance on multicore . . . 53
4.1.1 Power and performance of benchmarks vs. active cores . . . 53
4.1.2 Power and performance vs. clock frequency for all benchmarks . . . 56
4.1.3 Conclusions from the benchmark data . . . 58
4.2 A qualitative power model . . . 58
4.2.1 Minimum energy with respect to the number of active cores . . . 60
4.2.2 Minimum energy with respect to code performance . . . 60
4.2.3 Minimum energy with respect to clock frequency . . . 61
4.2.4 Validation of the power model for the benchmarks . . . 62
4.3 Consequences for system design . . . 63
4.4 Chapter summary . . . 65

5 Structured performance engineering **67**
5.1 The performance engineering process . . . 67
5.1.1 Description of the process . . . 67
5.1.2 Case study: An OpenMP-parallel 3D Jacobi smoother . . . 70
5.2 Identification of performance patterns on the node level . . . 78
5.2.1 Hardware performance metrics . . . 78
5.2.2 likwid-perfctr . . . 79
5.2.3 Performance patterns and event signatures . . . 79
5.2.4 Pattern categorization . . . 85
5.3 Patterns and models: Performance engineering refined . . . 85

II Applications **89**

6 A medical image reconstruction algorithm **91**
6.1 Introduction . . . 91
6.1.1 Computed tomography . . . 91
6.2 Experimental testbed . . . 93
6.3 The algorithm . . . 93
6.3.1 Theory . . . 93
6.3.2 Code analysis . . . 94
6.3.3 Simple performance models . . . 96
6.3.4 Algorithmic optimizations . . . 97
6.4 Single core optimizations . . . 98
6.4.1 SIMD vectorization . . . 98
6.4.2 AVX implementation . . . 100
6.5 In-depth performance analysis . . . 101
6.5.1 ECM performance model . . . 101
6.5.2 ILP optimization and SMT . . . 103
6.6 OpenMP parallelization . . . 103
6.6.1 ccNUMA placement . . . 104
6.6.2 Blocking/unrolling . . . 104
6.7 Results . . . 105
6.7.1 Validation of analytical predictions . . . 105
6.7.2 Parallel results . . . 105
6.8 Conclusion . . . 106
6.8.1 Summary of results . . . 106
6.8.2 Reassessment in view of performance patterns . . . 107

7 A performance- and energy-optimized lattice-Boltzmann fluid solver **109**
7.1 Introduction . . . 109
7.1.1 Related work . . . 109
7.1.2 The lattice-Boltzmann method . . . 110
7.1.3 Implementation options and data traffic analysis for LBM . . . 111
7.1.4 Test bed and benchmark cases . . . 113
7.2 Chip-level performance and scaling . . . 113
7.3 ECM model for the ILBDC code . . . 115
7.3.1 In-core analysis . . . 115
7.3.2 Data transfers and saturation behavior on the chip . . . 115
7.3.3 Validation of the performance model . . . 118
7.4 Power model . . . 118
7.4.1 Energy to solution for the LBM solver on the chip . . . 119
7.5 Highly parallel LBM simulations . . . 121
7.5.1 MPI parallelization in ILBDC . . . 121
7.5.2 Performance and energy at strong scaling . . . 122
7.6 Conclusion . . . 126
7.6.1 Summary of results . . . 126
7.6.2 Reassessment in view of performance patterns . . . 127

8 Conclusion **129**
8.1 Summary 129
8.2 Outlook 131

Bibliography **133**

List of acronyms and abbreviations

AoS	Array of structures
AVX	Advanced vector extensions
BP	Backprojection
ccNUMA	Cache-coherent nonuniform memory access
CFD	Computational fluid dynamics
CISC	Complex instruction set computer
CL	Cache line
CPI	Cycles per instruction
CPU	Central processing unit
CRS	Compressed row storage
CT	Computed tomography
DCT	Dynamic concurrency throttling
DDR	Double data rate
DP	Double precision
DRAM	Dynamic random access memory
ECM	Execution-cache-memory
Flop	Floating-point operation
FLUP	Fluid lattice site update
FMA	Fused multiply-add
FP	Floating point
FPGA	Field-programmable gate array
GPGPU	General-purpose (computing on) graphics processing units
HPC	High performance computing
HPM	Hardware performance monitoring
HT	HyperTransport
IACA	Intel architecture code analyzer
IB	InfiniBand
ILP	Instruction-level parallelism
IMB	Intel MPI benchmarks
I/O	Input/output

L1D	Level 1 data cache
L1I	Level 1 instruction cache
L2	Level 2 cache
L3	Level 3 cache
LD	Locality domain
LIKWID	Like I knew what I'm doing
LUP	Lattice site update
MPI	Message passing interface
MVM	Matrix–vector multiplication
NT	non-temporal
OLC	Outer-level cache
OS	Operating system
PCI	Peripheral component interconnect
PDF	Particle distribution function
QDR	Quad data rate
QPI	QuickPath interconnect
RAM	Random access memory
RAPL	Running average power limit
RCM	Reverse Cuthill-McKee
RISC	Reduced instruction set computer
RHS	Right hand side
RFO	Read for ownership
SIMD	Single instruction multiple data
SMP	Symmetric multiprocessing
SMT	Simultaneous multithreading
SoA	Structure of arrays
SP	Single precision
spMVM	Sparse matrix-vector multiplication
SSE	Streaming SIMD extensions
TDP	Thermal design power
TLB	Translation lookaside buffer
TRT	Two relaxation-time

Part I

Performance modeling and engineering

Chapter 1

Introduction

1.1 Scientific computing and code optimization

Computing has become the third pillar of scientific research besides theory and experiment. It is today an indispensable tool, and deeply interwoven with most areas of science and engineering. In many cases the required computing power is quite small and can be handled by a user's own laptop; other applications need vast computational resources such as federal computing centers in order to gain even qualitative results. Then it is necessary to think about how these systems can be used most effectively, so that the money spent in their procurement and operation has the highest possible impact.

Unfortunately, the domain scientists who write the software for parallel computers do not have the required knowledge to write efficient code. Even if an appropriate algorithm has been chosen, implementations often lack the ability to make best use of the resources. As a first step to remedy this unsatisfactory situation, computing centers offer compact courses and lectures, trying to teach at least the basic aspects of computer architecture, code parallelization, and optimization. As a consequence, many domain scientists spend more time on their code in an effort to make it "faster," applying the strategies learned. While this is a commendable endeavor, there is always the question when to stop optimizing, i.e., when the performance of the application code is "good enough." If the best possible performance level is unknown, the invested time and resources may far outweigh the benefit. This typical pattern is related to a disregard for the "80/20 rule," also called the "Pareto principle" [8]: Eighty percent of the effects are due to twenty percent of the causes. Translated to high-performance software development this means "twenty percent of the effort spent in optimization lead to eighty percent of the possible benefit." But again, an unknown possible benefit makes applying this rule impossible.

1.2 Performance modeling

Performance modeling in a broad sense means establishing a mathematical model for software/hardware interaction in order to predict or explain the runtime characteristics of a program on a given hardware. More specifically, performance modeling can have two different but sometimes overlapping goals: light speed calculation or extrapolation.

1.2.1 Light speed

A realistic upper limit for the performance of a code on a particular hardware may be called its *light speed*. Light speed allows a well-defined answer to the question whether an implementation of an algorithm is "good enough." A model leading to an accurate light speed estimate requires thorough code analysis, knowledge of computer architecture, and experience on how software interacts with hardware. The notion of light speed depends very much on the machine model underlying the hardware model; if the machine model misses an important performance-limiting detail, one might arrive at the (false) conclusion that light speed is not attained by the code at hand, while it actually is. Which hardware features should be included to arrive at a good balance between simplicity and predictive power is a crucial question, to which this work tries to give useful answers.

We call this approach *white-box performance modeling*. In complex cases it may not be possible to establish a model at all. If a model can be built, one can gain a deeper understanding of the interactions between software and hardware. If the model works, this is an indication that it describes certain aspects of this interaction accurately. If the model does not work (e.g., if the predicted performance is much lower or higher than the measured performance), it must be refined, leading to more insights.

A working model can help with predicting the possible gain of code optimizations. Changing the program code may require adjustments in the model, or even building a completely new model when the underlying algorithm was changed.

1.2.2 Extrapolation

Another goal of performance modeling is to extrapolate performance behavior from one hardware (e.g., a small system, or a given architecture) to different hardware (e.g., a large system, or a different architecture), which may not even exist yet, and for which only the specifications might be known. The assessment of performance characteristics on the known hardware can take a variety of forms. One option is to build on the first goal above, and then change the model parameters to accommodate the change to the new hardware. Alternatively one may take the code on the known hardware "as is" and try to figure out which hardware characteristics have the most impact on its performance. We call this *black-box performance modeling*, because the focus is not on understanding underlying mechanisms but on producing an accurate mathematical description, with accuracy defined only in terms of predictive power and not in terms of precision in describing the true underlying mechanisms. Less insight into the hardware-software interaction is gained by this approach, but there is the big advantage that it can often be used in very complex scenarios, where the other strategies fail. The statistical nature of the models thus obtained sometimes lead to the discovery of effects that would otherwise go unnoticed. See Sect. 1.4 on related work below for an example.

Fortunately the "80/20 rule" also applies to the performance characteristics of many programs in computational science: Most of the runtime is spent in a small part of the code. Consequently, the performance profile of many applications, i.e., the distribution of time over functions or loops, tends to be simple, and light speed techniques are applicable.

1.2.3 Machine model

As mentioned above, the interaction of software with hardware can be modeled to various levels of sophistication. Little variation exists in evaluating the requirements of the program code; the worst that could happen is that the assembly code generated by a compiler must be analyzed to uncover problems with inefficient execution.

On the other hand, there is a considerable bandwidth of possible machine models. Parallel computers are complex machines, but they are in principle deterministic. It would thus be possible to use a "cycle-accurate" simulator of the hardware to emulate the code execution and be able to acquire every possible detail. In practice, this approach would be unrewarding. Firstly, cycle-accurate models of real, modern processors exist but are intellectual property that is not divulged by chip manufacturers. Beyond the chip level, the sheer complexity of system components and their interactions rule out cycle-accurate models. Secondly, even if a cycle-accurate model were available it would not be of much use, because it would require considerable expertise, even beyond the level of a professional HPC expert, to interpret the results.

As a compromise one can establish simplified machine models, which are simple enough to be understood completely but which also allow for sufficiently predictive performance modeling. Applied in this sense, performance modeling is similar to the modeling techniques used in the natural sciences: It is implicit that the model is "wrong," i.e., that it does not encompass all possible effects, and that there might be assumptions going into it that are in no way justified. However, it is useful enough to understand the key mechanisms and probably predict new effects that have not been encountered or looked at before. If the model fails, its assumptions and simplifications are challenged, and new insight is gained.

1.3 Contributions

This section summarizes the essential contributions described in this treatise.

1.3.1 ECM Model

The "Execution-Cache-Memory" white-box performance model is a refinement of the well-known roofline model for predicting the performance and scalability of streaming loop kernels on multicore processors. It is to date the only approach that uses a simple machine model to arrive at an accurate single-core performance prediction for a real processor. Compared to the roofline model it drops some crucial assumptions and needs less phenomenological input. The roofline model can be seen as the "saturation limit" of the ECM model.

The ECM model has so far been applied to simple microbenchmark kernels [4, 1], to stencil algorithms of varying complexity [9, 10], to lattice-Boltzmann flow solvers [1, 7], and to a volume reconstruction algorithm from medical image processing [6]. Some of these applications will be revisited here (see below).

1.3.2 Multicore power model

Energy consumption aspects of computing have been moving into the focus of research in recent years. The multicore power model is a phenomenological description of the energy consumption of load-balanced parallel code on a multicore processor, taking into account the clock speed, the

number of utilized cores, the single-thread performance, and the maximum (saturated) performance, and the static and dynamic power consumption (per core). When energy consumption is an important metric, it answers questions such as "Is it better to use more cores at lower frequency or fewer cores at higher frequency?", "What is the optimum clock speed for a code that scales/saturates across the cores?", "Which influence does single-thread performance have on energy consumption?", etc. The model can also be used for design space exploration, and allows to estimate the trade-offs between a system's size and its energy consumption over its lifetime.

Combining the power model with the ECM model is especially interesting for saturating (i.e., bandwidth-bound) codes, enabling energy and performance optimization at the same time for an optimal selection of the operation point (number of cores used, clock speed).

1.3.3 Pattern-guided structured performance engineering on the node level

Structured performance engineering can be seen as a part of software engineering. It is an iterative process in which algorithm and code analysis, performance modeling, and optimization are applied repeatedly to arrive at a well-defined concept of "best possible performance." Its goal is to replace "shot-in-the-dark" optimizations, for which the possible outcome is unknown, with code changes whose performance impact was *expected*. This kind of structured approach is vital for the computational scientist, for whom programming is just a means to an end. It still requires some expertise in modeling and computer architecture, however, but the process also provides guidelines to how this knowledge may be best conveyed in courses and lectures.

The simple but instructive example of a 3D Jacobi smoother is used to show the advantages of structured performance engineering. While all optimizations and models applied to this case are well known, the embedding in a performance engineering process is new, and can be extrapolated to more complex cases.

The performance engineering process is supported by *performance patterns*. A pattern is a combination of observed performance behavior and data obtained from hardware performance monitoring. Instead of blindly applying tools to find "problems," the developer uses tools for the specific purpose of validating or refuting a performance model. A collection of relevant patterns for node-level performance engineering is identified and categorized into *maximum resource usage*, *hazards*, and *work inefficiency*. With the help of patterns, "best performance" gets a well defined meaning as "computing at a bottleneck."

Although the process is generally applicable to all kinds of parallel computing, this work is mostly restricted to the chip and node level, where the actual computational "work" is done.

1.3.4 Applications

The ECM model, the power model, and the concepts developed in the structured performance engineering process are applied to several application cases.

Sparse matrix-vector multiplication

Many algorithms in computational science require sparse linear algebra: Sparse eigensolvers, numerical methods for time evolution of quantum systems, finite-difference and finite-element

in fluid and structural mechanics, etc. These usually require high-performance sparse matrix-vector multiplication (spMVM) as a dominating, or at least significantly time-consuming component. SpMVM is also an example where predictive modeling is problematic, but where the general idea of a performance model can still be used with success. The roofline model is used to assess the quality of parallel spMVM implementations, establish upper performance limits, and lead to a better understanding of how well resources are utilized. Taking into account how much compute resources go into spMVM, a statement about when an implementation is "good enough" can be of great value. Moreover, the performance of spMVM depends heavily on the matrix structure, i.e., the location of the non-zero entries. Performance modeling is able to predict the advantage from matrix reordering techniques.

A volume reconstruction algorithm in medical physics

Backprojection (BP) algorithms are used in 3D volume reconstruction from images delivered by computed tomography (CT) devices. The performance of an implementation is strictly limited from below due to the use of interventional CT imaging techniques in modern surgery. Performance engineering can be used to understand the key requirements of a BP algorithm implementation to the hardware. This is an especially interesting case since there is usually not a single performance-limiting aspect like peak arithmetic throughput or data transfers. Via through performance modeling one can identify shortcomings of current standard processor architectures and propose improvements that could make a difference in reconstruction performance or enable higher-resolution imaging at constant cost. Especially for the case of multicore processors, the model predicts that it is possible to meet the required performance goal in current clinical CT applications without resorting to special-purpose hardware like GPGPUs or FPGAs. It can also identify a definite cross-over point where one or the other architecture is more advantageous with respect to performance or price/performance.

BP is a complex example where the first attempt at performance modeling via the roofline model fails completely because of unjustified assumptions about code execution on the hardware and the applicable performance pattern. Changing the pattern, using the ECM model, and applying guided code optimizations one is able to arrive at a code that fulfills the clinical performance limit (which happens to be a very precise definition of "good enough"). In the end, it is not a single one but a combination of patterns that apply.

Performance and energy optimization of a lattice-Boltzmann algorithm

The lattice Boltzmann method (LBM) is today established as a successful alternative to traditional flow solvers. LBM is traditionally believed to be memory-bound on all modern computer architectures, but the details are complex to fathom. A sparse-lattice implementation of a two-relaxation-time (TRT) LBM algorithm is used to answer the question for "best possible performance" on multicore systems. The influence of SIMD vectorization, the processor clock frequency, and the propagation pattern on the performance and energy consumption of the solver is studied using a combination of the ECM and power models. This enables an understanding of the complex interplay between in-core execution and data transfers, and leads to definite predictions about the performance of an implementation and to the identification of an energy-performance optimization space.

Since this implementation is MPI-parallel it is also possible to study if and how the consistent picture obtained on the socket level can be generalized to the highly parallel, distributed-memory case with strong scaling. It turns out that non-negligible MPI communication introduces not only overhead but also a core-bound component into the code execution characteristics, which has decisive influence on the optimal performance/power operating point. Since the energy consumption characteristics of memory-bound and core-bound codes are conflicting in terms of the optimal clock speed, especially when the baseline power dissipation of the whole system is taken into account, the optimization space for performance and power becomes very sharply defined. High single-core performance and an optimal choice of the number of cores used per socket are the dominant factors. In other words, mediocre-quality code running at a low clock speed (because there is an implicit assumption about memory-boundedness) wastes compute cycles and energy at the same time.

1.4 Related work in performance engineering

This section describes related work relevant for the performance modeling and engineering approaches. All other relevant related work, especially for the application cases in Part II, will be covered in the respective sections.

White-box performance modeling has been used for a long time. In the times of single-core in-order scalar processor architectures, where each instruction had a well-defined duration, accurate runtime predictions on the chip level were possible without simplifications. Out-of-order superscalar designs rendered this option impossible. The roofline model reduces machine and code properties to a small number of parameters: computational intensity, memory bandwidth, and peak performance. Although the term has been coined by Williams et al. [11], the model has been in use since the 1980s [12], and was an integral part of performance optimization on vector and early parallel computers [13].

Beyond the node level, a lot of effort has been invested in performance modeling of massively parallel applications [14, 15, 16, 17]. The work of Petrini et al. [15] is especially interesting since it used performance models to identify the previously unknown problem of *operating system noise* as a main factor limiting the scalability of large-scale bulk-synchronous code. It is a supreme case for the notion that valuable insight is gained when a model fails.

In order to manage the complexity of modeling modern systems (also in the highly parallel, distributed case) and to lower the required expertise for users, a number of simulation-based systems have been devised, for example the Warwick Performance Prediction toolkit (WARPP) [18] and its predecessors. They use a combination of compiler-based instrumentation, trace collection, and simulation to arrive at runtime predictions even for highly complex, massively parallel applications. In contrast, the performance engineering process described in this work relies on patterns, thorough manual or tool-based sequential code analysis, and experience, to gain insight into software-hardware interaction.

One of several interesting automatic tools that can help in identifying performance bottlenecks also for inexperienced users is PerfExpert [19]. It provides node-level tuning advice based on application tracing with hardware performance metrics. There is an implicit use of patterns in the tool, and it exposes to the user the possible bottlenecks and their severity on a loop-by-loop (or basic block) basis. Nevertheless, such an analysis can only be a starting point and considerable experience is still required to take the necessary actions for improving code

performance. There are very few activities in the field of structured performance engineering that do not build on automatic frameworks. One recent approach was described in [20], where a useful workflow was developed in the specific context of optimizing OpenMP code for modern multicore systems and accelerators.

1.5 Organization of this thesis

This work is organized as follows. Chapter 2 gives a high-level overview on core, processor, and node architecture, as far as it is required for the modeling approaches described later. The Intel Xeon Sandy Bridge processor is the dominant architecture in current multicore-based systems, and is hence described in more detail. Most of the examples and case studies in later chapters were conducted on Sandy Bridge systems.

Chapter 3 gives the details of white-box performance modeling. After a discussion of useful performance metrics and high-level scalability laws, the roofline model is discussed, together with a detailed account of its prerequisites and its potential for guided code optimizations. Multiple examples are given, including sparse matrix-vector multiplication. Using the failure of the roofline model in certain situation as a starting point, the ECM model is then developed and validated using a streaming benchmark. Complex application scenarios are left for Part II.

Chapter 4 develops the multicore power model based on three simple benchmarks that are prototypical for large classes of applications. Using the model, guidelines for energy-efficient computation with respect to single-core performance, clock speed, and the number of active cores are derived and validated using the benchmarks. Finally, the power model is used to define a design space for energy-efficient systems.

In Chapter 5 the structured performance engineering process is formulated, first in a coarse form without the explicit use of patterns. It is applied to an in-depth performance analysis and optimization study of a three-dimensional Jacobi smoother. Defining and categorizing performance patterns then paves the way for a refined view on performance engineering.

Chapter 6 uses performance engineering to develop an optimized implementation of a back-projection algorithm for volume reconstruction in medical imaging. Starting from a simple roofline model, which wrongly predicts a memory-bound situation, simple algorithmic and code optimizations are applied before using the ECM model to arrive at a better description of the performance of the code.

In Chapter 7 an implementation of the lattice-Boltzmann algorithm with two-relaxation-time collision operator and a sparse lattice representation is analyzed in view of SIMD vectorization, propagation pattern, and clock speed. The ECM and multicore power models are combined to yield a prediction for a possible performance-energy optimization space, in which an optimal operating point can be found. In the multi-node parallel case at strong scaling the influence of MPI communication overhead on these chip-level results is studied and the optimization space re-evaluated.

Finally, Chapter 8 gives a summary and an outlook to possible future work.

Chapter 2

Computer architecture

This chapter gives a brief overview of computer architecture, as far as it is relevant for the performance modeling and engineering approaches described later. Since this work is mostly about node-level issues, the focus is on the chip and node architecture. More detail can be found in [21].

A vast literature, including long-standing standard works [22], exists about this topic, but is mostly concerned with details that are for the most part irrelevant for the computational scientist. It is one of the important prerequisites for a good understanding of performance issues that architectural details should be exposed only as far as they are relevant for the modeling approach at hand.

2.1 Cores

Figure 2.1 shows a simplified, high-level architectural view of a microprocessor core. It is somewhat similar to current Intel designs, but sufficiently general to cover the features of all modern microprocessors. The components will be described briefly in the following.

2.1.1 Execution units and ports

The execution units perform the actual work of the core in terms of carrying out the instructions in the machine code. Usually there are units for carrying out floating-point multiply and add operations (MULT/ADD), loading and storing data from and to the memory hierarchy (LOAD or STORE), for integer operations (ALU), for address generation (ADRS), for branching (JMP), for moving, masking, and shuffling data (MOV/MASK), and for special operations such as divide and square roots (DIV). Some of these functions share a unit; for instance, in the prototypical design shown in Fig. 2.1 the floating-point MULT unit is shared with the DIV unit.

These units are usually pipelined, which means that a complex operation such as a floating-point MULT is split into several small steps, which can be executed in a single cycle (this would not be possible for the full operation). Pipelining makes it possible to run the core at higher clock frequencies, at the price of execution *latencies*: The latency is the number of clock cycles it takes after an operation was started until the result is available. For instance, a double-precision floating-point multiply has a latency (or *pipeline depth*) of five cycles on current Intel processors. Nevertheless, the pipeline can deliver one result per cycle if enough

independent work is available and can be fed to it. In this case, the pipeline is *filled* and operates at its maximum throughput. Dependencies between operations cause pipeline *bubbles*, i.e., the maximum throughput cannot be met because one pipeline stage has to wait for another (possible in a different pipeline) to complete.

The cost for starting and stopping the pipeline counts as overhead, but is negligible if the number of independent operations is large compared to the pipeline depth. Not all operations are effectively pipelined. For example, divide and square root tend to be very expensive because their throughput is similar to their latency. On the other hand, multiple pipelines can potentially operate in parallel, leading to a maximum execution rate of more than one instruction per cycle. This is called *superscalarity* or *instruction-level parallelism* (ILP).

The execution units are fed via execution ports (which may be implemented as queues). In modern out-of-order designs the operations can be executed in any order that is compatible with the dependencies in the program flow, but the completion of the instructions is always in program order. Simpler architectures such as the Intel Xeon Phi coprocessor (in its current design) have in-order execution, which means that the order of instructions in the machine code is crucial, and software pipelining techniques mus be employed by the compiler or the

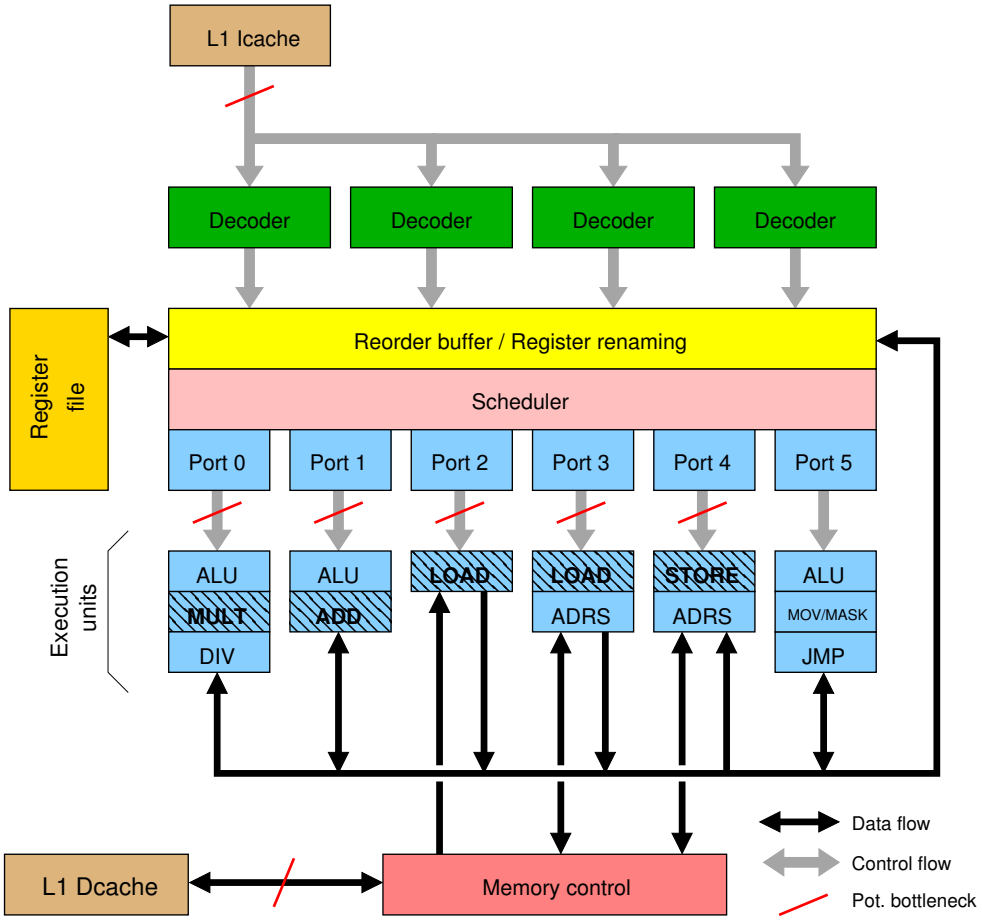


Figure 2.1: Simplified high-level architectural view of a modern microprocessor core. The most important execution units and potential bottlenecks are highlighted.

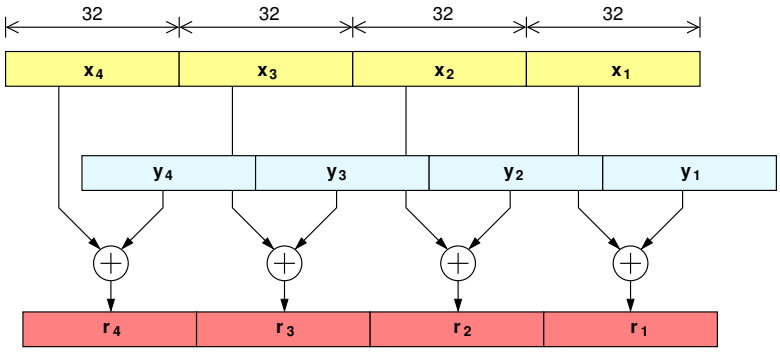


Figure 2.2: A SIMD-vectorized ADD instruction, operating on four single-precision operands simultaneously. The register width is 128 bits in this case. (Figure from [21])

programmer.

2.1.2 Registers

All instructions that are executed by the core work with operands stored in processor *registers*. The access to data in a register is latency-free, but the number of registers is limited (e.g., current x86 processors have 16 floating-point and 16 general-purpose [integer] registers). The register file is usually larger than what is visible to the programmer, since the hardware employs register renaming techniques to work around simple dependencies in the machine code.

It is one of the complex tasks in compilers to figure out which variables should be kept in registers, and when they can be written out to the memory hierarchy. Usually one can make valid assumptions, such as that an accumulation variable in a loop is kept in a register at least until the loop is finished. These assumptions go into the peak performance estimates required for performance modeling (see Chapter 3).

2.1.3 SIMD execution

Single instruction multiple data (SIMD) allows for simultaneous execution of operations on multiple operands by a single machine instruction. To this end, the architecture provides registers which are wider than they would have to be to store only a single operand. In modern SIMD-equipped processors the SIMD width is between 128 bits and 512 bits, allowing for two to eight "tracks" in double precision (four to 16 in single precision). Figure 2.2 shows the example of an ADD instruction operating on two 128-bit registers and performing four single-precision floating-point additions at the same time. The term *vectorization* is frequently used when SIMD is employed. It is decisive for performance that the SIMD principle cannot only be applied to arithmetic operations but also to data transfers, since the data throughput to and from the L1 cache may be a bottleneck. In such a case, SIMD instructions should be used for both arithmetic and LOAD/STORE. If this is not possible because the operands are not consecutive in memory, SIMD has limited benefit. See Chapter 6.4.1 for an example.

If SIMD instructions cannot be used, one must revert to *scalar* instructions, which typically use only the lowest part of a register. More advanced SIMD instruction sets allow for arbitrary masking of operations, so that any combination of SIMD tracks can be blocked out. SIMD is also no restricted to floating-point computation. For instance, the 128-bit SSE ("Streaming SIMD Extensions") instruction set on modern x86 designs also contains integer operations.

Ideally, a SIMD width of k elements increases the arithmetic peak performance of the core

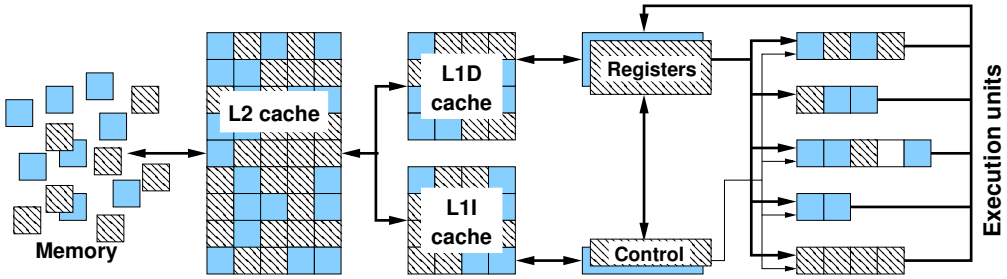


Figure 2.3: Simultaneous multi-threading allows the simultaneous execution of several (two in this case) threads on the same core. The threads share all resources except the register set. One thread can fill the bubbles left in the pipeline(s) by the other. (Figure from [21])

by a factor of k. Note that the SIMD concept is "orthogonal" to pipelining: There may be pipelining issues even if a code is perfectly SIMD-vectorized, since each SIMD track is a pipeline of its own in lockstep with the others, working on the same instruction(s) but on different data items.

2.1.4 Instruction cache and decoders

Instructions are read in program order from the Level 1 instruction cache, whose maximum transfer rate is limited (although this is a bottleneck that does not very often apply in scientific computing). All instructions must be decoded before they can be executed. A limited number of decoders is available for this task. Intel and AMD x86 designs have the special feature that the machine code read from the instruction cache is not the code that runs in the execution units. X86 machine instructions are translated by the hardware into so-called micro-ops (μops), which correspond to RISC-like instructions. RISC (Reduced Instruction Set Computing) is a design principle which allows only very simple instructions, so that they can be efficiently pipelined and executed at high clock frequencies. The x86 machine instruction set does not adhere to this principle, since it contains numerous complex instructions like the combination of a LOAD and an ADD. Splitting them to μops on the fly allows for more efficient execution in the core. Most restrictions on instruction throughput apply to the μops.

2.1.5 SMT

Simultaneous multi-threading (SMT) is used to increase the throughput of instructions on the core in certain cases. With SMT, a single core is able to execute multiple independent instruction streams at a time. To the applications and especially to the operating system it thus appears as multiple cores. However, almost all the resources are shared between the threads. The only fully duplicated resource is the register set, since each running program requires a full set of registers. It is certainly possible to run multiple processes or threads on a single core without SMT by time-sharing, but this is a feature of the operating system and not of the hardware.

The purpose of SMT is to make better use of the pipelines, which are often not fully utilized even with well-optimized software. Bubbles left in the pipeline stages by one thread or process can be filled by another (see Fig. 2.3). This can boost the throughput per core considerably in some cases (see Sect. 6 for an example). If the running threads are limited by a common

bottleneck outside the core pipelines, such as memory bandwidth or a shared queue, no benefit is expected.

For the application programmer, SMT is first and foremost a *topology* issue, because they must decide to use or to ignore the feature by proper binding of threads and processes to the resources.

2.1.6 Data cache

The results generated by executed instructions are either used from registers or eventually stored to the memory hierarchy. In general, the Level 1 data cache is the target for all LOAD and STORE operations. If a memory address is accessed whose contents are not already in the L1 cache (this event is called a *miss*), the corresponding cache line is read (the cache line size on x86 processors is 64 bytes). This pertains to STOREs as well: The cache line transfer initiated by a store miss is called a *write-allocate*.

2.1.7 Clock frequency and turbo mode

All operations in the execution core and the immediately connected caches run at the same clock speed (outer-level caches may run at a different frequency). This means that all performance numbers scale linearly with the clock frequency if no resources outside this area are used by the running code. Performance on this level is hence often discussed in terms of "amount of work per clock cycle," since this is a frequency-independent metric.

Modern processors often allow setting the clock frequency from user space. On Sandy Bridge and earlier designs this setting is global across all cores. Upcoming Intel processors will allow core-specific clock speeds.

A special feature found on all current x86 designs is "turbo mode" ("turbo core" for AMD). In turbo mode the chip can run faster than its nominal clock frequency, depending on the number of active cores and the die temperature. Intel processors even allow for a violation of the thermal design power (TDP) limit for a short amount of time. In view of Amdahl's Law, these measures are a way not only to increase the performance of sequential code but also to improve the scalability of parallel programs with non-negligible serial fraction.

2.2 Multicore chips

2.2.1 Multiple cores

To work around the power dissipation problems at high clock frequencies, processor manufacturers implement multiple cores on a die. This allows to make use of Moore's Law, i.e., the exponentially increasing transistor count per chip, within a constant power envelope.

Standard multicore processors feature a number of cache levels (usually two to three), most of which are private to each core. All caches except the L1 cache are traditionally unified, i.e., they can store instructions and data at the same time. The outer-level cache (OLC) is usually shared to allow for fast communication and synchronization between cores. When a cache line is brought from memory into the cache hierarchy there are essentially two options: In an *inclusive* cache hierarchy, any outer cache holds copies of all cache lines in the inner caches. An *exclusive* cache hierarchy always transfers cache lines to an inner level, from where they must be copied

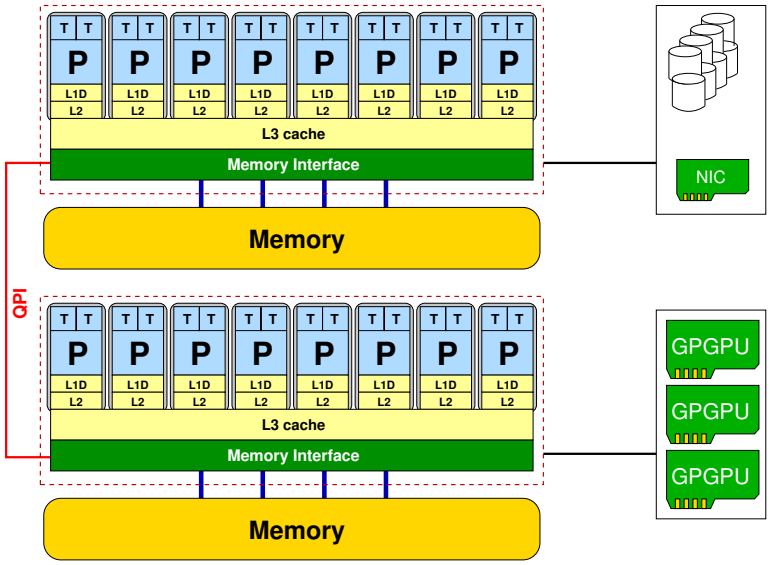


Figure 2.4: Simplified high-level structure, or "topology," of a shared-memory compute node based on two eight-core processors with Simultaneous Multi-Threading (SMT). I/O resources such as disks, network interfaces, or accelerator hardware, are connected via special buses, e.g., PCI Express.

down when replaced (evicted). A mixture of both concepts is also possible. Knowledge about the details of data transfers between the caches is required for accurate performance modeling (see Chapter 3 for examples).

Whenever multiple cores operate on different caches, a *cache coherence protocol* ensures that changes to different parts of the same cache line leave the caches in a consistent state. This can lead to performance problems if such changes happen in rapid succession, since cache lines have to be moved back and forth through the system ("false sharing").

2.2.2 Memory access

The memory interface is usually shared among the cores on a chip, so it is a typical candidate for a bottleneck. Memory *latency* is the time it takes to set up a cache line transfer, and is typically of the order of hundreds of core cycles. The overall time to transfer a cache line is absolutely dominated by latency. Prefetching mechanisms, either in hardware or in software, help with hiding the memory latency and actually reach the *bandwidth* limit of the memory interface if the data access pattern is appropriate. Best results are achieved with regular, unit-stride ("streaming") memory access. If the access pattern is strided or erratic, the memory bandwidth may not be exhausted and excess data transfers will occur due to the cache line concept (a full cache line is transferred but only part of it may be used before it gets evicted). See Sect. 3.3.2 for an example on the consequences of erratic memory access.

The latency and bandwidth considerations for main memory also apply to the higher levels of the cache hierarchy. Even though the latencies are shorter and the bandwidths are higher than for memory, non-unit strides or erratic accesses lead to large penalties, too.

2.3 Node and memory architecture

Two-socket servers have been at the price-performance "sweet spot" in commodity-based high performance computing for the last ten years. Figure 2.4 shows the structure of a typical compute node. Usually there is one chip per socket, with its own memory interface (some current

AMD-based server processors are a notable exception with two chips per socket). The chips are coupled via an interconnect network, which makes the whole setup a shared-memory system. In the commodity sector this interconnect is either *QuickPath* (QPI) or *HyperTransport* (HT). All the installed memory, no matter to which socket it is attached, can be accessed transparently by any core, and cache coherence is automatic. This principle is called *ccNUMA* (cache coherent non-uniform memory access).

A ccNUMA system is divided into *locality domains*. If a core accesses memory in a distant domain, this is more expensive (in terms of latency and bandwidth) than in the local domain where the thread is running. The penalty for non-local access is typically larger in systems with many domains, and is significant in any case. In order to make sure that any memory access is as local as possible, programs should make use of the first-touch principle, or "Golden Rule of ccNUMA:" The mapping of physical to logical memory addresses takes place not at the allocation, but at the *initialization* of a memory page. A page gets mapped into the locality domain of the core that writes to it first. Two crucial consequences arise from this: First, memory should be initialized by the same thread that uses it in a parallel computation, and second, threads should be bound to cores so that they cannot be migrated by the operating system to another ccNUMA domain, thereby losing their locality of memory access.

As Fig. 2.4 shows, even a single current multicore, multi-socket system has a rich "topology." Topology is the structure of a system in terms of the location of cores (and SMT threads) and the resources that they share. SMT threads always share a core, cores on a chip share some cache levels and the memory interface, and sockets share the coherent network and typical I/O resources such as the network interface, disks, accelerators, etc. Since shared resources are prone to become bottlenecks, topology is an essential component in performance assessment and modeling. Knowing about the sensitivity of a parallel program to the typical hardware bottlenecks leads the way to well-founded code optimization efforts. Chapter 5 describes a workflow which is based on this idea.

2.4 Test bed and tools

2.4.1 Intel Xeon "Sandy Bridge" processor [1]

Most of the performance data in this work was measured on compute nodes and systems based on the dual-socket eight-core Intel Sandy Bridge EP platform (Xeon E5-26XX). The Intel Sandy Bridge microarchitecture contains numerous enhancements in comparison to its predecessors, e.g., the "Westmere" and "Nehalem" chips. The following features are most important for the analysis in the following chapters [23]:

- Compared to SSE, the Advanced Vector Extensions (AVX) instruction set extension doubles the SIMD register width from 128 to 256 bits. At the same time, the load throughput of the L1 cache is doubled from 16 bytes to 32 bytes per cycle, so that a Sandy Bridge core can sustain one full-width AVX load and one half-width AVX store per cycle. With SSE or scalar execution, these limits are changed: In both cases the core can sustain either one load and one store, or two loads per cycle, to the effect that many loops do not show a $4\times$ speedup of core execution when going from scalar mode to AVX (see Sect. 3.3.1 for an example).

- The core can execute one ADD and one MULT instruction per cycle (pipelined). With double-precision AVX instructions, this leads to a peak performance of eight flops per cycle (sixteen at single precision). In general, the core has a maximum instruction throughput of six μops per cycle.
- Each core can execute two concurrent streams of instructions using simultaneous multithreading (SMT).
- The L2 cache sustains refills and evicts to and from L1 at 256 bits per cycle (half-duplex). A full 64-byte cache line refill or evict thus takes two cycles. This is the same as on earlier Intel designs.
- The L3 cache is segmented, with one segment per core. All segments are connected by a ring bus. Each segment has the same bandwidth capabilities as the L2 cache, i.e., it can sustain 256 bits per cycle (half-duplex) for refills and evicts from L2. This means that the L3 cache is usually not a bandwidth bottleneck, which is an improvement compared to previous Intel processors.
- All parts of the chip, including the L3 cache (which is part of the "Uncore"), run at the same clock frequency, which can be set to a fixed value in the range from 1.2–2.7 GHz. The speed of the DRAM chips is constant and independent of the core clock.
- One Xeon E5-26XX socket has four DDR3-1333 or DDR3-1600 memory channels for a theoretical peak bandwidth of 42.7 GB/s or 51.2 GB/s. In practice, between 36 GB/s and 42 GB/s can be achieved with the standard STREAM benchmark [24] at high clock frequencies.
- Sandy Bridge is the first Intel processor exposing a power measurement infrastructure to the user code. Power and energy measurements reported in this work are for the "package" only, i.e., they ignore the installed RAM. Preliminary results for the power dissipation of installed DIMMs are between 2 and 9 W per socket (16 GB RAM in 4 DIMMs of 4 GB each), depending on the workload (memory-bound vs. cache-bound).

Some of these low-level hardware properties will be revisited when discussing performance models and results.

2.4.2 Tools

Source code was compiled with the Intel compiler in version 12.1 or 13.1. The tools of the LIKWID tool suite [25, 26] were employed for binding threads of OpenMP programs to cores (`likwid-pin`), for hardware performance monitoring (`likwid-perfctr`), and for energy measurement (`likwid-powermeter`).

2.4.3 SuperMUC

The large-scale parallel runs of lattice-Boltzmann simulations used in Chapter 7 to demonstrate energy-efficient execution were conducted on the "SuperMUC" federal supercomputer at Leibniz Supercomputing Center (LRZ)[1] in Garching near Munich. It is a tier-0 PRACE[2] system and

[1] http://www.lrz.de/english/
[2] http://www.prace-ri.eu/

one of the main federal compute resources in Germany. It is built from a number of 512-node "islands," with a fully non-blocking fat tree FDR10 InfiniBand connectivity inside each island. A compute node comprises two Intel Sandy Bridge EP (Xeon E5-2680) eight-core processors with a base clock frequency of 2.7 GHz.

The actual clock speed of the processors in SuperMUC can be influenced by a so-called "energy tag," which is supplied upon job submission together with a parameter specifying how much performance degradation the user wants to tolerate for their job. A heuristic based on hardware performance counter measurements of the user's previous jobs with the same energy tag then sets the clock frequency for the job (turbo mode cannot be used). These measures try to establish a user-friendly semi-automatic mechanism for saving energy.

Chapter 3

White-box performance modeling on the chip level

As described in Chapter 1, performance modeling can be a powerful tool for software engineering in computational science. Taking a modeling approach to the interaction of software with hardware is, while not new, a concept that is not yet in wide use. Since performance for problem solving is generated in the execution units of processor cores, and since all relevant computational resources are replicated when using multiple chips, modeling and optimization activities must start at the chip level. After revisiting high-level scalability models, this chapter introduces the well-known *roofline model* and the new *ECM performance model*. These models are shown to provide valuable insights into the performance properties of modern processor chips and the code that runs on them. In Chapter 5 the models will be put into the larger context of node-level performance engineering.

3.1 Performance and speedup

This section addresses performance and scalability of serial and parallel programs from an abstract point of view. In computing, *performance* is usually defined as *work* divided by *time*, where "work" is a problem (or a well-defined part thereof), and "time" is the wall-clock time required to solve it:

$$P = \frac{W}{T} \tag{3.1}$$

An accurate definition of "work" is crucial for a sensible assessment of performance. For instance, if solving the problem involves necessary overhead that takes time but is not in itself part of the result, this overhead does not constitute "work." Communication or synchronization in parallel computing are typical examples.

Parallel computing is often concerned with the question of how much more performance can be achieved if the work is done with "accelerated" resources, such as multiple cores, chips, or nodes, or with special hardware like GPGPUs. *Speedup* can thus be defined as

$$S = \frac{P_\mathrm{a}}{P_0} = \frac{W_\mathrm{a}}{W_0}\frac{T_0}{T_\mathrm{a}}\,, \tag{3.2}$$

where P_a denotes "accelerated" performance and P_0 is a given baseline level. This definition does not specify whether the same amount of work is done in the baseline and in the accelerated

case (W_0 vs. W_a). The baseline performance is frequently chosen to be equal to one, so that speedup and accelerated performance are identical. Another popular choice is $W_0 = W_a$, i.e., the same problem is solved in both cases.

3.1.1 Useful performance metrics

Most simulation tasks are centered around algorithms that require floating-point computations. A natural unit to choose for "work" is thus the floating-point operation, or *flop*. The peak "speed" of processors or whole systems is also usually given in flop/s, since it is the most generic and widely applicable measure for performance. It also allows for a rough first estimate of how "effectively" the compute resources of a system are utilized: A large deviation of actual from peak flop/s performance *might* indicate a problem with code execution that should be addressed. However, there are various objections to using flops for assessing program performance:

- The flop/s metric can be easily manipulated. It is straightforward to rewrite implementations so that the flop count is strongly increased, without improving the time to solution of the actual problem.
- Different implementations of one algorithm, or even machine codes generated by different compilers from the same source, can exhibit strongly deviating flop counts.
- There are algorithms which do not exclusively rely on floating-point computations.

Hence, one should be careful whenever the flops metric is used. Alternatives exist in many cases, such as in iterative solvers where one "iteration," or "update," may serve as a convenient and implementation-independent unit of work.

3.1.2 High-level scalability models

It is clear that the concepts of *performance* and *speedup* must be clearly separated. Especially looking at speedup figures alone may lead to false conclusions about the "quality" of parallel or accelerated execution. In the end, all that counts is how much work per unit of time can be done; if P_0 in (3.2) is small, the achievable speedup S may be significant even if P_a is mediocre. If and how it can be determined whether P_0 or P_a are "good" will be the topic of Chapter 3.

Nevertheless, the speedup metric can still be useful, since it allows a quantitative judgment about how efficiently resources are put to use when not all of a program's execution can be accelerated. In many cases the speedup can be written as

$$S = \frac{W_a}{W_0} \frac{T_s + T_p}{T_s + T_p^{acc} + c} , \tag{3.3}$$

where $T_0 = T_s + T_p$ is the non-accelerated runtime of the program, and T_p is a part which can be perfectly accelerated so that this part takes a time of T_p^{acc} in the accelerated case. Note that T_p^{acc} may also include any change in runtime caused by $W_0 \neq W_a$, e.g., if the accelerated execution is performed on a bigger problem. The parameter c quantifies any overhead that is caused by the process of acceleration, such as communication or synchronization. Frequently the non-accelerated runtime T_0 is normalized to one, so that $T_s = s$ and $T_p = p$ become "non-accelerated" and "accelerated" fractions, respectively, and $s + p = 1$. At the same time one can

set $W_0 = s + p = 1$. If we finally interpret a non-accelerated fraction as a part of the overhead, (3.3) becomes

$$S = \frac{W_\mathrm{a}}{p_\mathrm{acc} + \delta} \,, \tag{3.4}$$

with $\delta = c + s$. The quantity p_acc is "accelerated, normalized runtime" and describes the "perfect" part of execution, while δ contains all factors impeding good scalability.

Some important special cases are worth noting. If $W_0 = W_\mathrm{a} = 1$ and $\delta = s$ we have $p_\mathrm{acc} = p/\alpha$, where α is an acceleration factor. This leads directly to *Amdahl's Law* [27],

$$S = \frac{1}{s + \frac{1-s}{\alpha}} \,. \tag{3.5}$$

It quantifies the *law of diminishing returns*: The more effort is put into improving one part of the problem (in this case the accelerated fraction $p = 1 - s$), the less effect it has on the overall time to solution. In the limit $\alpha \to \infty$ we get $S \to s^{-1}$. If $\alpha = N$, with N being the number of "workers" used for solving the accelerated part, we speak of *strong scaling*, and "acceleration" becomes "parallelization." For finite $c > 0$ the effective speedup is diminished:

$$S = \frac{1}{s + \frac{1-s}{\alpha} + c} \,, \tag{3.6}$$

and if $c'(\alpha) > 0$ this means that the speedup does not even increase when the acceleration factor goes up. A typical example is OpenMP parallelization overhead, which is linear or logarithmic in the number of threads used for parallelizing a loop. If the amount of work in an OpenMP-parallel loop is too small, performance will go down when the number of threads is increased.

If $W_\mathrm{a} = s + \alpha p$, i.e., the accelerated problem size is increased by a factor which is equivalent to the achievable acceleration on p, (3.4) becomes

$$S = \frac{s + (1-s)\alpha}{c} \,, \tag{3.7}$$

which is *Gustafson's Law*. For $\alpha = N$ we speak of *weak scaling*. The impact of the overhead c on scalability is much weaker in this case. For large α it is sufficient to have $c'(\alpha) < 1$ for getting a speedup that grows without bounds.

Although these high-level models are useful for deriving general guidelines and scaling properties, they are completely detached from any concrete hardware, and can on first sight not account for many of the effects seen on real systems. However, it is possible to modify and refine the high-level scalability laws to accommodate many different performance-limiting factors. For instance, the dependence of the overhead c on the acceleration factor α (or the number of workers N) can be modeled after some often-encountered patterns, such as halo exchange, or be set to mimic the communication characteristics of special networks. A coverage of some interesting cases can be found in [21].

It turns out that Amdahl's and Gustafson's Laws must be substituted by more specific models when trying to describe and understand the performance behavior of modern chips. However, there is one effect that can be described well by a slight modification of Amdahl's Law with overhead (3.6): Boosting scalability by code slowdown. Going back to (3.3) and normalizing such that $T_\mathrm{s} + T_\mathrm{p} = \mu$, (3.6) becomes

$$S = \frac{\mu}{\mu\left(s + \frac{1-s}{\alpha}\right) + c} = \frac{1}{s + \frac{1-s}{\alpha} + c\mu^{-1}} \,. \tag{3.8}$$

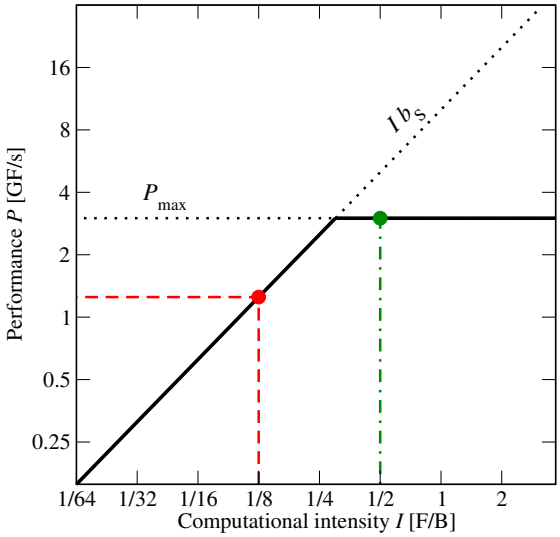


Figure 3.1: Roofline model for a processor with a memory bandwidth of $b_S = 10\,\text{GB/s}$ running a code with an applicable peak performance of $P_{\max} = 3\,\text{GF/s}$. The dashed (dotted-dashed) line represents a computational intensity that leads to memory-bound (core-bound) performance.

This change models a performance increase ($\mu < 1$) or decrease ($\mu > 1$) of the pure execution time (non-accelerated plus accelerated parts). It is clear from this formulation that scalability is improved for $\mu > 1$ if $c \neq 0$: Whenever there is non-negligible overhead, slowing down code execution boosts scalability. This is why white-box performance modeling based on hardware parameters and code inspection is so important. It answers the question which bottleneck is relevant and whether it has been reached, and scalability (or speedup) becomes subordinate.

3.2 The roofline model

The roofline model [12, 13, 11] is a well-established approach to predicting upper performance limits for code execution on a processor. While it is possible to model arbitrary code, the roofline model works best when applied to loop kernels with streaming data access patterns. Due to its generality it can be used with multicore processors, GPGPUs, and other hardware for which its basic assumptions are valid.

3.2.1 Building the model

The central premise of the model is that the performance of a loop is either limited by data transfers or by code execution, whichever takes longer. A detailed account of the assumptions and prerequisites will be given in Sect. 3.2.2 below.

The following steps are required to build the model for a specific loop:

1. By algorithm and code inspection, determine $P_{\max}$, the applicable upper performance limit for the loop code, assuming that all required data comes from the cache that is closest to the core(s) (i.e., the L1 cache). Considerable knowledge about the hardware architecture may be required to arrive at a realistic $P_{\max}$ value.

2. By algorithm and code inspection, determine the *computational intensity*[1] I of the loop code. This is the ratio of "work" performed and data volume required to do the "work."

[1] The reciprocal of the computational intensity is called *code balance*: $B_C = I^{-1}$.

Only the bottleneck data path is considered for the data volume (see next point).

3. Determine the applicable peak bandwidth b_S of the data path that constitutes the bottleneck for transferring the necessary data to the core(s) and back. This step may require measurements using microbenchmarks, either because of undocumented hardware features or because some data path cannot be operated at 100% of its theoretical bandwidth. Note that the L1 cache is not a bottleneck in this sense, since it is included in the modeling of $P_{\max}$.

Once these quantities are known, the expected performance of the loop code is

$$P = \min\left(P_{\max}, I \cdot b_S\right) \ . \tag{3.9}$$

Figure 3.1 gives a graphical representation of the roofline model, for a hypothetical processor with a maximum main memory bandwidth of $b_S = 10\,\text{GB/s}$ and for an applicable peak performance of 3 GF/s. The minimum function in (3.9) is expressed by the roofline shape (solid line), while the inaccessible performance regions are shown as dotted lines. At a given computational intensity, the expected performance can be read off the diagram as shown by the dashed ($I = 0.125\,\text{B/F}$) and dotted-dashed ($I = 0.5\,\text{B/F}$) lines.

In the first case the limiting factor is the main memory bandwidth, since the roofline is hit in the sloped part. The expected performance of $P = 1.25\,\text{GF/s}$ is far below $P_{\max}$, i.e., the computational units run idle most of the time. The second case shows a core-bound situation, where the expected performance is determined by the code execution in the core(s).

Several aspects are worth noting here. First, the roofline model is *resource-driven* in the sense that the maximum available resources (bandwidth or execution) are the limiting factors for code performance. It is not specified how exactly these resources are put together on the chip; for instance, the number of cores, the width and number of memory channels, the details of core execution, etc., are not part of the model (3.9), although they can certainly be used to determine the parameters b_S and $P_{\max}$. Second, we have implicitly assumed that $P_{\max}$ is independent of I, at least for the two cases shown in Fig. 3.1. This will not be the case in general, since different algorithms (or even different implementations of the same algorithm) have usually a different composition in terms of low-level loop code (number of instructions, fraction of LOAD/STORE vs. arithmetic operations, SIMD vs. scalar, etc.). Consequently, the roofline model must not be seen as a machine that produces a correct number (P) when fed with an input (I); it is rather a *method* and must be revised whenever the code under consideration changes substantially, even if I stays constant.

The roofline model can be helpful in performing guided code optimizations [11]. Optimizations such as unrolling and blocking [21] can influence the computational intensity of a loop. On the other hand, modifications of the low-level machine code (software prefetching, SIMD vectorization, etc.) can move the positions of both parts of the roofline without changing the computational intensity. Both will lead to an immediate prediction of the expected change in performance. Hence, the model helps with judging whether an optimization would be worth the effort. See Sect. 5.1.2 for an example.

3.2.2 Model prerequisites and assumptions

The roofline model is based on clear concepts of "work" and "data traffic to do the work." One possible kind of "work" is "number of floating-point operations," but this is not always desir-

able, as was shown in Sect. 3.1.1. Any other well-defined and countable quantity will also do, including problem-specific metrics: loop iterations, solver iterations, lattice site updates, image pixels, function evaluations, etc. The "data traffic" across the bottleneck data path includes all data, not only the data that is seen by LOAD and STORE instructions in the code. See Sect. 3.3 for examples.

A number of critical assumptions go into the roofline model:

- *Bottleneck assumption.* The slowest data path, i.e., the bottleneck is modeled only; all others are assumed to be infinitely fast. "Slow" is not defined here in terms of bandwidth but of the time it takes to transfer the required data. Hence, a high-bandwidth data path can still be the bottleneck if the data volume is large. For instance, if ten times the data volume must be delivered by the L3 cache than by the main memory, the L3 cache will be the bottleneck, despite having a five times larger bandwidth.

- *Overlap assumption.* Data transfer and core execution overlap perfectly. If this assumption did not hold, the roofline in Fig. 3.1 would change into a smooth "archline," and there would be no clear inflection point at $I \cdot b_\mathrm{S} = P_\mathrm{max}$.

- *Saturation assumption.* It is possible to fully utilize the bandwidth of the bottleneck ("saturation") if the model predicts a bandwidth limitation. The saturated bandwidth is either a documented number or must be determined via microbenchmarking.[2]

- *Streaming assumption.* There are no latency effects, i.e., all data accesses use perfect streaming mode. This assumes that hardware- or software-based prefetching mechanisms work perfectly, and that the large latency for accessing a cache line can be completely hidden.

Any of these assumptions may not hold in some situations, but they are reasonably loose to support many code patterns in scientific computing. The ECM model, which will be introduced in Sect. 3.4, can handle some of the cases in which the roofline model fails to deliver useful results.

3.2.3 Model-guided code optimizations

Building a performance model opens several possibilities for performance optimizations. Instead of blindly applying code changes and hoping for improvements, *guided* decisions can be made, using the model as a predictive tool for the expected gain. This is a crucial part of the performance engineering process, which will be introduced in Chapter 5.

Figure 3.2 shows examples of typical code optimizations and their consequences in terms of the roofline model. As a prerequisite, we assume that the model is always "correct" in the sense that it reflects the minimum requirements of the implementation utilizing the maximum capabilities of the hardware. For instance, strided array access must already be taken into account by a correct assessment of the data traffic. The labels (numbers) in the graph correspond to the items in the following list:

[2] In the non-saturated case, measured *effective* bandwidths can serve as a substitute for saturated bandwidth, but the ECM model (see below) clearly shows that this approach delivers inaccurate results.

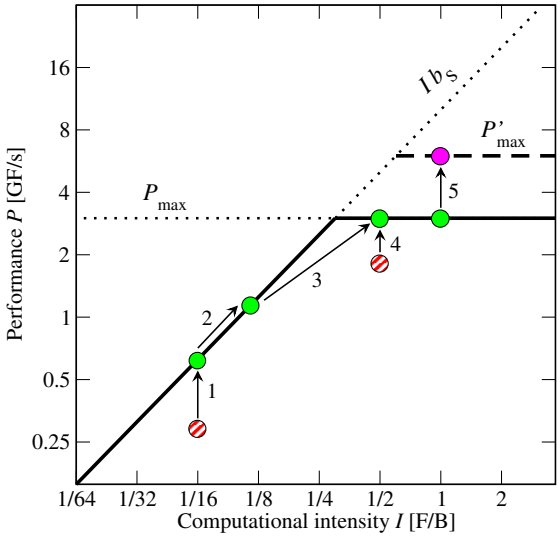


Figure 3.2: Typical optimization approaches in the roofline model. P'_{max} is a new applicable performance limit, which emerges from making use of architectural features that were not accessible before. Deviations from the model (hatched points) are here caused by code deficiencies and not by flaws in the model, i.e, it is assumed that the model is always "correct." See text for details.

1. *Reaching bandwidth saturation.* If the model predicts a limitation by memory bandwidth but is too optimistic with regard to a measurement, this can point to deficiencies in the code, such as missing software prefetching instructions, which prevent saturation because the streaming assumption cannot be met. Hardware performance monitoring (HPM) can then reveal whether this conjecture is true (see Sect. 5.2 for more details on HPM-assisted performance engineering).

2. *Improving computational intensity at bandwidth saturation.* If the loop is bandwidth-bound and exhausts the memory bandwidth, increasing the computational intensity by typical optimizations such as stride reduction, unrolling, and blocking [21] will lead to a proportional gain in performance.

3. *Improving computational intensity and going core bound.* If an improvement of the computational intensity does not cause a proportional gain in performance, chances are that the inflection point at $I \cdot b_{\text{S}} = P_{\text{max}}$ was crossed, and that the loop has become core bound. A further increase of I will then not lead to any speedup.

4. *Improving in-core efficiency.* A deviation from the core-bound applicable maximum performance P_{max} usually points to suboptimal low-level loop code. This can be verified by careful inspection of the assembly code or the compiler's diagnostic messages. A typical example is the lack of SIMD vectorization, which may be caused by the compiler missing important information, such as the non-existence of array aliasing [21].

5. *Improved use of architectural features.* If the code or the algorithm can be changed so that new, performance-critical architectural features become accessible, the model must usually be adapted for a new $P'_{\text{max}} > P_{\text{max}}$. For instance, stalls caused by pipeline hazards may be removed by reformulating the algorithm to become purely data-parallel.

Using the model as a guide for expected performance gain it becomes possible to judge whether some (possibly complex) code changes would be worth the effort. The following section highlights some instructive examples, and Chapter 5 embeds optimization approaches in a structured performance engineering process.

Listing 3.1: Pseudo-code for the vector triad throughput benchmark, including performance measurement. The actual benchmark loop is highlighted.

```
1    double precision, dimension(:),allocatable :: A,B,C,D
2  ! Intel-specific: 512-byte alignment of allocatables
3  !DEC$ ATTRIBUTES ALIGN: 512 :: A,B,C,D
4
5    call get_walltime(S)
6
7  !$OMP PARALLEL PRIVATE(A,B,C,D,i,j)
8  ...
9        do j=1,R
10 ! Intel-specific: Assume aligned moves
11 !DEC$ vector aligned
12 !DEC$ vector temporal
13           do i=1,N
14              A(i) = B(i) + C(i) * D(i)
15           enddo
16           ! prevent loop interchange
17           if(A(N/2).lt.0) call dummy(A,B,C,D)
18       enddo
19 !$OMP END PARALLEL
20
21   call get_walltime(E)
22
23   WT = E-S
```

3.3 Examples for roofline modeling

The following examples serve to demonstrate the roofline model in a simple situation (purely streaming kernel) and a more complex setting, where the fourth of the above assumptions does not hold (sparse matrix-vector multiplication). See [11] for a comprehensive coverage of application cases.

3.3.1 Pure streaming kernel

A standard example for a streaming kernel that is limited by data transfers on any architecture in any memory hierarchy level is the *Schönauer vector triad* [13] as shown in Listing 3.1. Note that there is no real work sharing in the benchmark loop (lines 13–15), since the purpose of the code is to fathom the bottlenecks of the architecture. The code is equipped with Intel compiler directives to point out some crucial choices: All array accesses are aligned to suitable address boundaries (lines 3 and 11) to allow for aligned MOV instructions, which are faster on some architectures. Furthermore, the generation of nontemporal store instructions ("streaming stores") is prevented (line 12).[3] For benchmarking purposes this kernel is executed many times with the same data set, so that the data transfer capabilities of each memory level can be accurately measured [21].

First we conduct a large-N (in-memory) roofline analysis for an eight-core Intel Sandy

[3] Intel compiler options `-O3 -openmp -xAVX -opt-streaming-stores never -nolib-inline -fno-inline` were used for these tests.

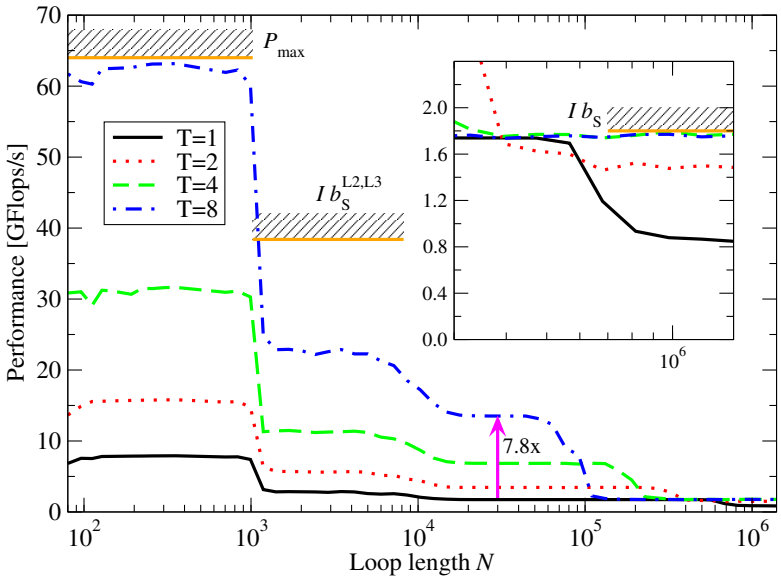


Figure 3.3: Throughput performance vs. loop length per core of the AVX-vectorized Schönauer vector triad on 1, 2, 4, and 8 cores of an Intel Sandy Bridge processor at 3.0 GHz. Inset: enlarged region for $N > 3 \cdot 10^5$ (memory-bound). The core-bound, memory-bound, and L2/L3-bound roofline limits are highlighted.

Bridge chip running at a clock frequency of 3.0 GHz. One core can sustain one full-width AVX load and one half-width AVX store per cycle (see Sect. 2.4.1). Hence, the execution bottleneck on the core is the load port throughput, and four loop iterations can be done in three cycles (the two half-wide stores needed for four consecutive elements of `A(:)` can be overlapped with the three loads for four consecutive elements of `B(:)`, `C(:)`, and `D(:)`). The arithmetic instructions, i.e., one ADD and one MULT instruction, take only a single cycle, and there is sufficient superscalarity in the core so that they can be overlapped with the loads and stores. Thus, the maximum performance for code execution if the data is in the L1 cache is 8 flops in 3 cy, i.e., $P_{\max} = 8$ GF/s per core or 64 GF/s on eight cores (at 3 GHz).

The loop code causes the same data traffic per flop in all memory hierarchy levels beyond the L1 cache, so the bottleneck is the main memory interface. The STORE on the elements of `A(:)` causes a write miss on every cache line, triggering a write-allocate transfer. Thus the actual data volume per iteration is not 32 bytes but 40 bytes. At a computational intensity of 2 flops/40 bytes (or 0.05 B/F) and a maximum memory bandwidth of $b_S = 36$ GB/s, the memory-bound performance limit is $I \cdot b_S = 1.8$ GF/s. This is far below the $P_{\max}$ limit on the cores, so we expect a memory-bound performance of $P \lesssim 1.8$ GF/s. Figure 3.3 shows the performance characteristic of the vector triad in "throughput mode," i.e., every core runs an independent loop with length N and there is no work sharing. The roofline prediction is very accurate when using four cores or more, but is much too optimistic at one or two cores (see inset). Even for one core the model still predicts the memory-bound limit (1.8 GF/s < 8 GF/s). Obviously one or more of the assumptions above do not hold when using too few cores.

For small data sets ($N \leq 1024$) all arrays fit in the L1 cache and the $P_{\max}$ prediction applies. It can be seen from Fig. 3.3 that the compiler was able to generate the "perfect" machine code for this loop, since the maximum possible in-core performance is achieved (see Listing 3.2). Every core has its own private L1 cache, so there is no bottleneck and the scalability from one to eight cores is also optimal.

When the data is in the core-private L2 cache, whose bandwidth limit is 32 bytes/cy per core ($b_S^{L2} = 768$ GB/s), we get $I \cdot b_S^{L2} = 38.4$ GF/s for the bandwidth prediction, which is far above the

Listing 3.2: "Perfect" AVX-vectorized assembly code for the bulk section of the Schönauer vector triad (remainder loop omitted). The add and multiply instructions are highlighted. Not that this is x86 assembly code, which does not reflect the actual RISC-like μops which get executed on the hardware. The compiler has unrolled the original loop 16 times (each AVX instruction applies to four double-precision operands).

```
1  label:
2    vmovupd   (%rdx,%r8,8), %ymm1
3    vmovupd   32(%rdx,%r8,8), %ymm4
4    vmovupd   64(%rdx,%r8,8), %ymm7
5    vmovupd   96(%rdx,%r8,8), %ymm10
6    vmulpd    (%rcx,%r8,8), %ymm1, %ymm2
7    vmulpd    32(%rcx,%r8,8), %ymm4, %ymm5
8    vmulpd    64(%rcx,%r8,8), %ymm7, %ymm8
9    vmulpd    96(%rcx,%r8,8), %ymm10, %ymm11
10   vaddpd    (%r13,%r8,8), %ymm2, %ymm3
11   vaddpd    32(%r13,%r8,8), %ymm5, %ymm6
12   vaddpd    64(%r13,%r8,8), %ymm8, %ymm9
13   vaddpd    96(%r13,%r8,8), %ymm11, %ymm12
14   vmovupd   %ymm3, (%rdi,%r8,8)
15   vmovupd   %ymm6, 32(%rdi,%r8,8)
16   vmovupd   %ymm9, 64(%rdi,%r8,8)
17   vmovupd   %ymm12, 96(%rdi,%r8,8)
18   addq      $16, %r8
19   cmpq      %r9, %r8
20   jb        label
```

measurement. The same prediction applies for the L3 cache, which is shared but segmented, so that its bandwidth scales across all cores. Again, some underlying assumptions of the roofline model do not hold here. See Sect. 3.4 for a detailed account of these effects.

3.3.2 Sparse matrix-vector multiplication [2, 3]

Given the pivotal role that sparse matrix-vector multiplication (spMVM) plays for many algorithms in computational science, high-performance implementations of this kernel are of utmost importance, and have been the subject of intense research over the last decade [28, 29, 30, 31, 32, 33, 2, 34, 35, 3]. For large data sets, the spMVM is strongly memory-bound. Many different storage schemes exist to make data access to the matrix and LHS and RHS vectors as efficient as possible. Most of these schemes are highly specific to certain hardware architectures, although there is a recent development of a universal sparse matrix format [3]. Here we highlight only those aspects of the spMVM operation that are relevant in the context of the roofline model. Figure 3.4 shows a sketch of an spMVM operation, without any specific choice of matrix data format. Usually the access to the LHS and matrix data can be organized to be compatible with the cache line structure, but the RHS accesses may incur large overhead due to low spatial and/or temporal locality. The streaming assumption for the roofline model may thus not be valid. This section shows how one can deal with this problem and still employ the roofline model to gain insight, although its predictions are "wrong."

The most popular storage scheme, and the one that is suited for a wide range of matrices on standard cache-based microprocessors, is the "Compressed Row Storage" (CRS) format (see

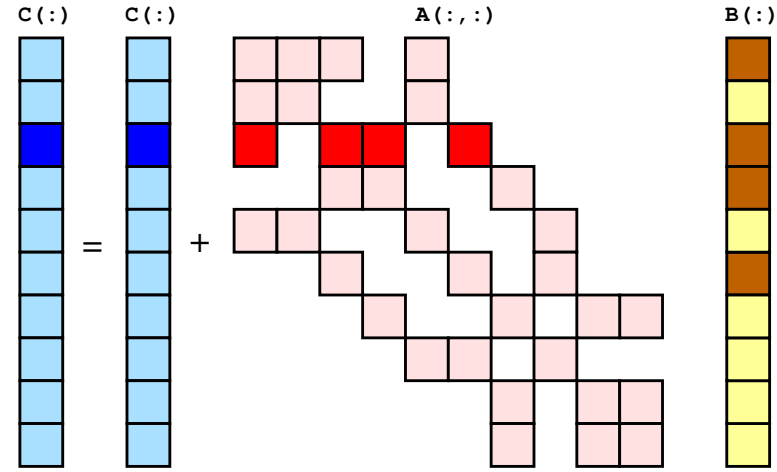


Figure 3.4: Sparse matrix-vector multiply. Dark elements visualize entries involved in updating a single LHS element. Unless the sparse matrix rows have no gaps between the first and last nonzero elements, some indirect addressing of the RHS vector is inevitable. (Figure from [21])

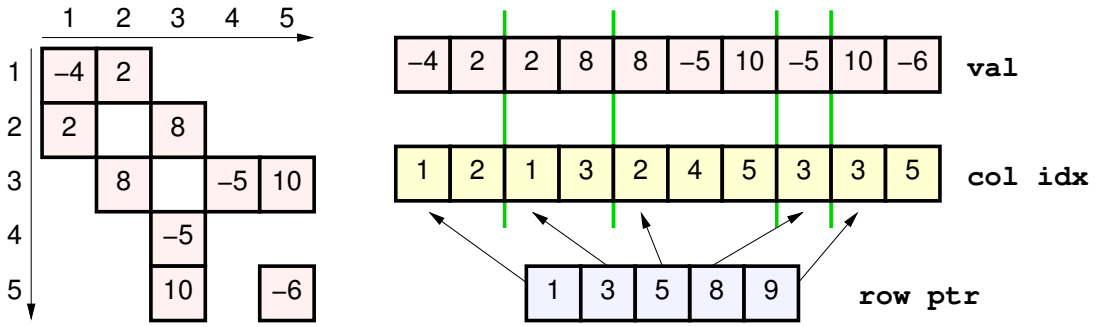


Figure 3.5: CRS sparse matrix storage format. (Figure from [21])

Fig. 3.5). The nonzero entries of the matrix are stored consecutively, row by row, in an array `val(:)`. The original column indices of those entries are stored in another consecutive (integer) array `col_idx(:)`, and the starting offsets of all rows are put in the array `row_ptr(:)`. Using this format, the spMVM kernel takes the form shown in Listing 3.3. It is characterized mainly by data streaming (arrays `val[(:)` and `col_idx(:))` with partially indirect access (RHS vector `B(:)`). Under the assumptions given in Sect. 3.2.2, it is possible to establish roofline-type performance models [11, 21]. For matrix formats that require some amount of zero-padding to make the data layout compatible with the requirements of the hardware, such as in SELL-C-σ or ELLPACK, the required data traffic can be adjusted [3].

Listing 3.3: OpenMP-parallel CRS spMVM kernel.

```
1 !$OMP parallel do
2 do i = 1,N_r
3   do j = row_ptr(i), row_ptr(i+1) - 1
4     C(i) = C(i) + val(j) * B(col_idx(j))
5   enddo
6 enddo
7 !$OMP end parallel do
```

The computational intensity can be read off from Listing 3.3 for *square matrices*: [2, 34]

$$I_{\mathrm{CRS}}^{\mathrm{DP}} = \left(\frac{2\,\mathrm{flops}}{v_{\mathrm{mat}} + v_{\mathrm{RHS}} + v_{\mathrm{LHS}}} \right) , \tag{3.10}$$

where v_{mat} accounts for reading the matrix entries and column indices, v_{RHS} is the traffic incurred by reading the RHS vector (including excess traffic due to insufficient spatial and/or temporal locality), and v_{LHS} is the data volume for updating one LHS element. Assuming double precision matrix and vector data and four-byte integer indices we have $v_{\mathrm{mat}} = (8+4)\,\mathrm{bytes}$ and $v_{\mathrm{LHS}} = 16\,\mathrm{bytes}/N_{\mathrm{nzr}}$, where N_{nzr} is the average number of nonzeros per row. The RHS vector must be read at least once, but the actual data volume may be much larger. This discrepancy is quantified by the parameter α in $v_{\mathrm{RHS}} = 8\alpha\,\mathrm{bytes}$. Hence, we get:

$$I_{\mathrm{CRS}}^{\mathrm{DP}} = \left(\frac{2}{8+4+8\alpha+16/N_{\mathrm{nzr}}} \right) \frac{\mathrm{flops}}{\mathrm{byte}} . \tag{3.11}$$

The following considerations are simpler to express in terms of the code balance, since individual effects can be easily attributed to distinct additive terms. The code balance is thus

$$B_{\mathrm{CRS}}^{\mathrm{DP}} = \left(6+4\alpha+\frac{8}{N_{\mathrm{nzr}}} \right) \frac{\mathrm{bytes}}{\mathrm{flop}} . \tag{3.12}$$

The value of α is governed by a subtle interplay between the matrix structure and the memory hierarchy on the compute device: If there is no cache, i.e., if each load to the RHS vector goes to memory, we have $\alpha = 1$ and the RHS causes the same traffic as the matrix entries. A cache may reduce the balance by some amount, to get $\alpha < 1$. In the ideal situation when $\alpha = 1/N_{\mathrm{nzr}}$, each RHS element has to be loaded only once from main memory per spMVM[4]. The worst possible scenario occurs when the cache is organized in cache lines of length L_{C} elements, and each access to the RHS causes a cache miss. In this case we have $\alpha = L_{\mathrm{C}}$, with $L_{\mathrm{C}} = 8$ or 16 on current processors. The locality of the RHS vector access and, consequently, the value of α can be improved by applying matrix bandwidth reduction algorithms, such as "Reverse Cuthill McKee" (RCM) [36]. Note also that, depending on the algorithm and the problem size, the RHS vector may reside in cache for multiple subsequent spMVM kernel invocations, although the matrix must still be fetched from memory. In this special case we have $\alpha = 0$.

The CRS-based roofline model (3.12) must be modified for data formats that require some zero fill-in. For instance, the SELL-C-σ format cuts the matrix into horizontal stripes, whose height (number of rows) is a small multiple of the applicable SIMD width (register width on standard processors, warp size on GPGPUs). These "chunks" are padded with zeros to become rectangular. This eliminates the need for conditionals in the inner loop and thus enables SIMD vectorization and prevents warp divergence [34, 3].

The severity of the fill-in overhead can be quantified by an additional parameter $\beta \leq 1$, which in case of SELL-C-σ we call "chunk occupancy," but which can certainly be defined without reference to any specific storage format. The reciprocal of β quantifies the format-inherent average data traffic per non-zero matrix element. Note the excess traffic for $\beta < 1$ only arises for the matrix value and column index but not for the RHS element. This is because all padded column indices should be set to zero; thus, the same (the first) RHS element is accessed

[4] This corresponds to the $\kappa = 0$ case in [2]

Listing 3.4: Read-only microbenchmark for bandwidth assessment.

```
#pragma omp parallel for reduction(+:sum)
for(i = 0; i < N; ++i) {
    sum += a[i];
}
```

for all padded elements and the corresponding relatively high access frequency will ensure that this element stays in cache. The corrected code balance is then

$$\begin{aligned} B^{\mathrm{DP}}(\alpha,\beta,N_{\mathrm{nzr}}) &= \left(\frac{1}{\beta}\left(\frac{8+4}{2}\right)+\frac{8\alpha+16/N_{\mathrm{nzr}}}{2}\right)\frac{\text{bytes}}{\text{flop}} \\ &= \left(\frac{6}{\beta}+4\alpha+\frac{8}{N_{\mathrm{nzr}}}\right)\frac{\text{bytes}}{\text{flop}}\,. \end{aligned} \tag{3.13}$$

The roofline model can now be used to predict the maximum achievable spMVM performance (we skip the P_{max} derivation since it is evident that spMVM is memory-bound):

$$P(\alpha,\beta,N_{\mathrm{nzr}},b_{\mathrm{S}}) = \frac{b_{\mathrm{S}}}{B^{\mathrm{DP}}(\alpha,\beta,N_{\mathrm{nzr}})}\,. \tag{3.14}$$

As usual, b_{S} is the achievable memory bandwidth as determined by a suitable microbenchmark. Since the spMVM kernel is dominated by read operations unless N_{nzr} is very small, such a microbenchmark should reflect this behavior (see Listing 3.4). Using $\beta = 1$ in (3.14) we obtain the analogous expression for CRS or any other format without zero-padding overhead.

As a special case we focus on the $\alpha = 1/N_{\mathrm{nzr}}$ scenario, which has been described above. In many realistic scenarios, processors with large last-level caches can often hold the RHS vector in the cache, even if it is updated frequently. Then the performance model reads:

$$P(1/N_{\mathrm{nzr}},\beta,N_{\mathrm{nzr}},b_{\mathrm{S}}) = \frac{b_{\mathrm{S}}}{(6/\beta+12/N_{\mathrm{nzr}})\,\frac{\text{bytes}}{\text{flop}}}\,. \tag{3.15}$$

For matrices with a sufficiently large number of non-zeros per row ($N_{\mathrm{nzr}} \gg 12$) one finally arrives at the best attainable performance for spMVM operations:

$$\bar{P} = \frac{b_{\mathrm{S}}\beta}{6\,\frac{\text{bytes}}{\text{flop}}}\,. \tag{3.16}$$

Note that these estimates are based on the optimistic assumptions of the roofline model. Nevertheless, (3.16) provides an upper bound for spMVM performance on all compute devices if the matrix data comes from main memory.

In the most general case, the code balance depends on α and β, the latter of which can be determined from the sparse matrix format. On the other hand, α can only be predicted in very simple cases. Moreover, the value of b_{S} determined by the microbenchmark (Listing 3.4) could be too optimistic because the streaming assumption for the roofline model may not be satisfied due to erratic access patterns. Hence it seems that the roofline model cannot be used for spMVM kernels with less than optimal spatial and temporal locality. While this is true for performance

prediction, it is still valuable to think in terms of bandwidth limitations in oder to find out more about how well resources are used: The value of α can be determined by *measuring* the memory bandwidth (or data volume) of the spMVM kernel using a tool such as likwid-perfctr [25, 26] and setting the code balance equal to the ratio between the measured transferred data volume V_{meas} and the number of executed "useful" flops, $2 \times N_{\text{nz}}$. Note that this is only possible if the code is limited by memory bandwidth. We then obtain

$$B^{\text{DP}} = \left(\frac{6}{\beta} + 4\alpha + \frac{8}{N_{\text{nzr}}} \right) \frac{\text{bytes}}{\text{flop}} = \frac{V_{\text{meas}}}{N_{\text{nz}} \cdot 2\,\text{flops}} , \tag{3.17}$$

which can be solved for α:

$$\alpha = \frac{1}{4} \left(\frac{V_{\text{meas}}}{N_{\text{nz}} \cdot 2\,\text{bytes}} - \frac{6}{\beta} - \frac{8}{N_{\text{nzr}}} \right) . \tag{3.18}$$

Once α is known, (3.13) allows to determine what fraction of the memory bandwidth is used by the RHS accesses.

As an example we pick the "kkt_power" matrix from the University of Florida sparse matrix collection.[5] It originates from a non-linear optimization (Karush-Kuhn-Tucker) for finding the optimal power flow. The matrix has $N_{\text{nz}} = 14.6 \cdot 10^6$ nonzeros and $N_{\text{r}} = 2.06 \cdot 10^6$ rows, which leads to $N_{\text{nzr}} = 7.1$ nonzeros per row on average. An OpenMP-parallel CRS-based spMVM with this matrix on an Intel Sandy Bridge chip yields an observed performance of $P = 4.1\,\text{GF/s}$ and an overall memory traffic volume of about $V_{\text{meas}} \approx 258\,\text{MB}$. Inserting V_{meas} into (3.18) and setting $\beta = 1$ (no padding) we get $\alpha = 0.43$. From the number of matrix rows one could expect that the RHS vector should fit into the 20 MiB last-level cache of the processor, leading to $\alpha_{\text{min}} \lesssim 1/N_{\text{nzr}} = 0.14$. However, since the two million elements for the RHS would already take 80% of the cache capacity, competition with other data (notably the matrix) causes capacity misses and frequent evictions. A value of $\alpha = \alpha_{\text{min}}$ would incur the minimum data volume for loading the RHS (once), so the product $\alpha N_{\text{nzr}} \approx 3.1$ quantifies the actual data traffic generated by it. Every RHS element is thus loaded three times from memory. Using (3.13) we can finally calculate the relative overhead for this:

$$\frac{B^{\text{DP}}(\alpha)}{B^{\text{DP}}(\alpha_{\text{min}})} \approx 1.15 . \tag{3.19}$$

If the extra RHS traffic accounts for 15% overhead, this is also the optimization potential for matrix reordering techniques such as RCM, assuming that the achievable memory bandwidth stays the same. Note that the sparsity pattern of the matrix influences the access pattern. The overhead may become very large if the nonzeros are very scattered. See [2] for a case study involving matrix reordering to improve performance.

The efficiency of the memory access can be evaluated by comparing the achieved bandwidth when running spMVM to the maximum bandwidth obtained with a microbenchmark:

$$\varepsilon_{\text{mem}} = \frac{PB^{\text{DP}}(\alpha)}{b_{\text{S}}} \tag{3.20}$$

The numerator is $PB^{\text{DP}}(\alpha) = 36.3\,\text{GB/s}$ in this case. Although the maximum read-only bandwidth of the Intel Sandy Bridge chip used for these tests is $b_{\text{S}} = 43\,\text{GB/s}$, the applicable baseline

[5] http://www.cise.ufl.edu/research/sparse/matrices

Listing 3.5: Double-precision divide-accumulate kernel.

```
1 double precision :: sum, c
2 double precision, dimension(N) :: a
3 ! loop called many times with different c
4 sum = 0.d0
5 !$OMP parallel do reduction(+:sum)
6 do i = 1,N
7   sum = sum + c / a(i)
8 enddo
9 !$OMP end parallel do
```

is probably lower due to the low number of nonzeros per row. Nevertheless, the erratic RHS access causes some inefficiency, which may also be lowered by reducing α.

In summary, applying the roofline model to the sparse matrix-vector multiplication kernel seems to be impossible at first sight. The uncertainties in assessing the actual data traffic caused by accesses to the right-hand side vector can lead to an overly optimistic bandwidth-bound performance prediction. Turning the model around, however, and measuring the performance and the actual data traffic, allows to fix the free parameter α and estimate optimization opportunities. Hence, the roofline model is still very useful, although it does not actually "work."

3.3.3 Divide-accumulate kernel

A simple but instructive example for the prediction of the effect of optimizations is the divide-accumulate kernel in Listing 3.5. It is also a preview to the use of patterns in performance modeling, which will be discussed in Chapter 5. We use a 3.0 GHz six-core Intel Xeon "Westmere" processor as a test platform.

The applicable peak performance of this kernel can be easily computed by taking into account that the double-precision divide instruction on this processor has a throughput of 22 cy, since it is essentially non-pipelined. This means that a divide can be completed only in every 22nd cycle [23]. All other execution units that are needed in this kernel (LOAD and ADD) cannot be bottlenecks even if pipelining did not work, since their latency can be easily hidden behind the 22-cycle divide. There is a vectorized double-precision divide in the SSE4.2 instruction set, which brings down the effective throughput to 11 cycles per loop iteration (2 flops). At 3 GHz and six cores we thus have

$$P_{\max} = 6 \cdot \frac{3 \cdot 10^9 \mathrm{cy/s}}{11\,\mathrm{cy}/2\,\mathrm{flops}} = 3.27\,\mathrm{GF/s}\ . \tag{3.21}$$

For large N the bandwidth limitation is given by the code balance of $B = 4\,\mathrm{B/F}$ and the (measured) memory bandwidth of $b_S = 21\,\mathrm{GB/s}$, so the roofline model is

$$P = \min\left(3.27\,\mathrm{GF/s}, \frac{21\,\mathrm{GB/s}}{4\,\mathrm{B/F}}\right) = 3.27\,\mathrm{GF/s}\ . \tag{3.22}$$

Hence, this kernel is clearly core-bound on the Westmere chip (all other things being equal, it would be memory-bound starting at ten cores). As a consequence, the performance of the loop does not depend on the location of the data; even if the loop were short and all elements of

Listing 3.6: Optimized version of the divide-accumulate kernel with pre-computed reciprocals in `ra(:)`.

```
1  double precision :: sum, c
2  double precision, dimension(N) :: a, ra
3  ! ra(:) is pre-computed once
4  !$OMP parallel do
5  do i = 1,N
6    ra(i) = 1.d0 / a(i)
7  enddo
8  !$OMP end parallel do
9  ...
10 ! loop called many times with different c
11 sum = 0.d0
12 !$OMP parallel do reduction(+:sum)
13 do i = 1,N
14     sum = sum + c * ra(i)
15 enddo
16 !$OMP end parallel do
```

`a(:)` came from the L1 cache, above performance limit would still apply. For very short loops the overhead from the OpenMP parallelization would become a problem, of course. We can calculate the array loop length where this will occur: At $11/6$ cy per loop iteration, and a typical (measured) OpenMP overhead (barrier latency plus thread team start) of about 3000 cy on the full chip (which can be measured using, e.g., the EPCC OpenMP microbenchmarks [37]), the penalty from OpenMP will have less than 10% impact at $N \gtrsim 16000$. This data set would still fit in the L1 cache of 32 KiB per core. Since the OpenMP overhead is highly compiler-dependent, this estimate can change when another compiler is used.

Often, additional knowledge about the processing of data outside the current loop of interest is useful for optimizing code. For instance, if the parameter `c` changes between different invocations of the loop kernel but the elements of `a(:)` stay the same, it is more efficient to pre-calculate the reciprocals and store them in a separate array (see Listing 3.6). With proper unrolling in place to circumvent the stalls in the ADD pipeline, the applicable peak performance of this new loop is the overall arithmetic peak of the processor, since one Westmere core can sustain the LOAD, the ADD, and the MULT instructions in the same cycle. Hence, $P_{\max} = 6 \cdot 2 \cdot 2 \cdot 3\,\text{GF/s} = 72\,\text{GF/s}$, and the loop is strongly memory-bound for large N with an expected performance of $P = 21/4\,\text{GF/s} = 5.2\,\text{GF/s}$. If N is small and the data is in the L1 cache, it would take only 2048 cy to process the full L1 cache size. Assuming again an OpenMP penalty of 3000 cy this means that the region of working set sizes where OpenMP overhead plays a significant role extends far into the L2 cache.

3.3.4 Conclusions and best practices for applying the roofline model

The roofline model is simple enough to enable a straightforward performance prediction in simple cases, but often the problem is to determine a realistic $P_{\max}$ limit. In a first step one can estimate $P_{\max}$ by assuming the hardware peak execution rate for arithmetic operations, i.e., the pure flops. In terms of the architectural model of the single core described in Chapter 2 this would mean that the ADD and MULT ports are the relevant bottleneck on the core level. Taking

LOADs and STOREs into account, as shown in the vector triad example above, will already lead to a considerable refinement, but there is still the implicit assumption that all instructions in the loop body are independent and can be executed at the highest possible rate allowed by the pipelines. If the critical code execution path contains dependencies (leading to, e.g., pipeline bubbles), the prediction of $P_{\max}$ becomes more involved and may require the use of tools except for very simple situations. See Chapter 6 for an example from medical imaging.

In all but the most trivial cases the construction and validation of a performance model is an *iterative process*, which may require several cycles of refinement until a model is "good." See Chapter 5 for a general view on structured performance engineering.

3.4 The Execution-Cache-Memory (ECM) model: A refined performance model for streaming loop kernels on multicore [4, 1]

For large data sets, typical memory-bound kernels in computational science show a peculiar scaling behavior across the cores of a multicore chip: Up to some critical number of cores t_s scalability is good, but for $t > t_s$ performance saturates and is capped by some maximum level. Beyond the saturation point, the roofline model can often be used to predict the performance, or at least its qualitative behavior with respect to problem parameters, but it does not encompass effects that occur between the cache levels. For the same reason it cannot correctly explain the observed performance levels for streaming kernels if the bottleneck is within the cache hierarchy (see Fig. 3.3 above). The "Execution-Cache-Memory" (ECM) model adds basic knowledge about the cache bandwidths and organization on the multicore chip to arrive at a more accurate description on the single-core level. Although the model can be used to predict the serial and parallel performance of codes on multicore processors, its main purpose is to develop a deeper understanding of the interaction of code with the hardware. This happens when the model *fails to coincide* with the measurement (see Sect. 3.4.1 below).

The following sections give a brief account of this model and show how it connects to the roofline model. It is then applied to parallel streaming kernels, and refined to account for some unknown (or undisclosed) properties of the cache hierarchy. In Part II the model is applied to several important algorithms in computational science: stencil smoothers, a lattice-Boltzmann flow solver, and a backprojection algorithm from medical imaging.

3.4.1 The Execution-Cache-Memory (ECM) model: Single core

The main premise of the ECM model is that the runtime of a loop is composed of two contributions: (i) The "core time," which is the time it takes to execute all instructions, with all operands of loads or stores coming from or going to the L1 data cache. (ii) The "data delays," which is the time it takes to transfer the required cache lines into and out of the L1 cache. The model further assumes, just like the roofline model, that hardware or software prefetching mechanisms are in place, hiding all cache transfer latencies. Here we additionally assume that the cache hierarchy is strictly inclusive, i.e., that the lines in each cache level are also contained in the levels below it. The model can accommodate exclusive caches as well; see [4] for examples.

Since all data transfers between cache levels occur in packets of one cache line, the model always considers one cache line's worth of work. For instance, if a double precision array must be read with unit stride for processing, the basic unit of work in the model is eight iterations at

a cache line size of 64 bytes. The execution time for one unit of work is then composed of the in-core part T_{core} and the data delays T_{data}, with potential overlap between them.

T_{data} reflects the time it takes to transfer data to the L1 cache through the memory hierarchy and back. This value will be larger if the required cache line(s) are "far away." Note that, since we have assumed perfect prefetching, this is not a simple latency effect: It comes about because of limited bandwidth and several possibly non-overlapping contributions. This assumption does not work, e.g., on GPGPUs, where latency is hidden by massive threading; the ECM model in its current form is not appropriate for such architectures.

On a Sandy Bridge core, the transfer of a 64-byte cache line from L3 through L2 to L1 takes a maximum of four and a minimum of two cycles (32-byte wide buses between the cache levels), depending on whether the transfers can overlap or not. Furthermore, the L1 cache of Intel processors is "single-ported" in the sense that, in any clock cycle, it can either reload/evict cache lines from/to L2 or communicate with the registers, but not both at the same time.

The core time T_{core} is more complex to estimate. In the simplest case, execution is dominated by a clear bottleneck, such as load/store throughput or pipeline stalls. Some knowledge about the core microarchitecture, like the kind and number of execution ports or the maximum instruction throughput, is helpful for getting a first estimate. For example, in a code that is completely dominated by independent ADD instructions, the core time is, to first order, determined by the ADD port throughput (one ADD instruction per cycle on modern Intel CPUs). In a complex loop body, however, it is often hard to find the critical execution path that determines the number of cycles. The Intel Architecture Code Analyzer (IACA) [38] is a tool that can derive more accurate predictions by taking dependencies into account. See Sect. 6 for a detailed analysis of a complex loop body with IACA.

Putting together a prediction for the overall execution time requires making best- and worst-case assumptions about possible overlaps of the different contributions described above. If the measured performance data is far off those predictions, the model misses an important architectural or execution detail, and must be refined. A simple example is the write-allocate transfer on a store miss: A naive model for the execution of a store-dominated streaming kernel (like, e.g., array initialization `A(:)=0`) with data in the L2 cache will predict a bandwidth level that is much higher than the measurement. Only when taking into account that every cache line must be transferred to L1 first will the prediction be correct.

Although an accurate determination of T_{core} (or, equivalently, P_{max}) is also required for the roofline model, there are two crucial differences between the roofline model and the ECM model:

- The roofline model only considers a single bottleneck, i.e., *one* data path or the in-core code execution. The overlap and bottleneck assumptions ensure that whatever it is that takes the longest time will determine the performance of the loop. Since the actual amount of overlap depends on factors that are outside of the model (and which are mostly unknown anyway), these assumptions is lifted in the ECM model. This allows for a range of predicted performance values depending on how much overlap actually occurs.

- The roofline model relies on the saturation assumption, which states that 100% of the bandwidth of the slowest data path can be utilized. The ECM model, on the other hand, starts with a single-core analysis and thus *predicts* non-saturation for all data paths involved, within the limits given by different assumptions for the overlap. This is the case

where the roofline model often fails (see Sect. 3.3.1). See below for how multicore scaling behavior is incorporated into the ECM model.

As shown in the vector triad example in Sect. 3.3.1, a single core cannot saturate the memory interface, although a roofline analysis of peak performance vs. memory bandwidth suggests otherwise: The single-threaded triad benchmark only achieves about 840 MF/s, which corresponds to a bandwidth of less than 17 GB/s. The ECM model attributes this discrepancy to non-overlapping contributions from core execution and data transfers. While loads and stores to the four arrays are accessing the L1 cache, no refills or evicts between L1 and L2 can occur. The same may be true for the lower cache levels and even memory, so that memory bandwidth is not the sole performance-limiting factor anymore. Core execution and transfers between higher cache levels are not completely hidden and the maximum memory bandwidth cannot be met. See Sect. 3.4.3 below for a detailed account of how to apply the model to streaming kernels.

However, when multiple cores access main memory (or a lower cache level with a bandwidth bottleneck, like the L3 cache of the Intel Westmere processor), the associated core times and data delays can overlap among the cores, and a point will be reached where the bottleneck becomes relevant. Thus, it is possible to predict when performance saturation sets in with increasing number of cores.

3.4.2 The ECM model: Multicore scaling

The single-core ECM model predicts lower and upper limits for the bandwidth pressure on all memory hierarchy levels. When multiple cores are executing a loop, shared data paths become potential bottlenecks, since the combined "pressure" from all cores may exceed their capacity. When the bandwidth of one data path is exhausted, performance starts to saturate [39]. This principle is visualized in Fig. 3.6 using a timeline graph: T_{chip} encompasses T_{core} and all contributions from T_{data} that emerge from scalable data paths, such as inner cache levels. The remaining time T_{mem} is the time spent with transferring data over bottlenecks, whose bandwidth does not scale with the number of cores. Here we assume no overlap between these contributions. Once the number of cores is greater than

$$t_{\mathrm{s}} = \frac{T_{\mathrm{chip}} + T_{\mathrm{mem}}}{T_{\mathrm{mem}}} \ , \tag{3.23}$$

saturation sets in and the performance is completely determined by T_{mem}, which happens at three cores in this example. We call this the "saturation point." At this point, the bandwidth-based prediction from the roofline model works well. The performance at t cores is thus:

$$P(t) = \min\left(tP_0, P_{\mathrm{roof}}\right) \ , \tag{3.24}$$

where P_0 is the single-core performance (or ECM prediction) and P_{roof} is given by the bandwidth limitation in the roofline model. On the Intel Sandy Bridge processor the only shared bandwidth resource is the main memory interface.

Just as in the roofline model, the maximum main memory bandwidth is an input parameter. In principle it is possible to use the known hardware properties of the memory interface (clock speed, bus width, number of memory channels), but this is over-optimistic in practice. For current Intel and AMD processors, the memory bandwidth achievable with standard streaming benchmarks like the McCalpin STREAM [40, 24] is between 65 and 90% of the theoretical

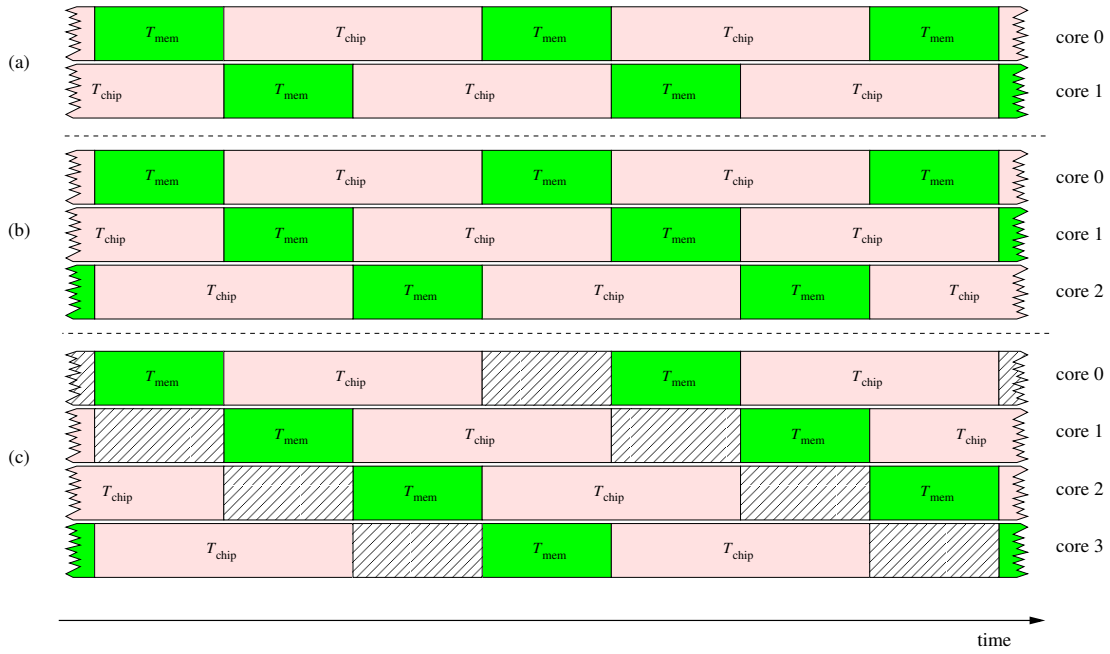


Figure 3.6: ECM model multicore scaling and saturation on a chip with a memory bandwidth bottleneck. T_{chip} is the time for running the loop with data from scalable resources, while T_{mem} is the memory transfer time. In this example we assume $T_{\text{chip}}/T_{\text{mem}} = 2$. (a) With two cores, memory access is not a bottleneck and scalability is perfect. (b) Three cores are needed to saturate the memory interface. (c) Beyond three cores, performance does not increase since the bottleneck is exhausted. This results in idle phases (hatched boxes).

maximum. Architectural peculiarities, however, may impede the optimal use of the memory interface with certain types of code. One example are data streaming loops with a very large number of concurrent load/store streams, which appear, e.g., in implementations of the lattice-Boltzmann algorithm (see Sect. 7). The full memory bandwidth as seen with the STREAM benchmarks cannot be achieved under such conditions. The reasons for this failure are as yet unknown and are subject to further investigation.

3.4.3 Validation via streaming benchmarks

We validate the ECM model by using the Schönauer vector triad [13] as a throughput benchmark (see Listing 3.1) on the Sandy Bridge architecture.

Single-core analysis

All loop iterations are independent. The in-core analysis is analogous to the roofline model, with the exception that we now consider a unit of work of one cache line's length, i.e., comprising eight scalar iterations (sixteen flops). Six full-width AVX loads and two full-width AVX stores are required to execute the unit of work. From the microarchitectural properties we know that this takes six cycles. In Fig. 3.7 the LOADs and STOREs are represented by the arrows between the L1D cache and the registers. The floating-point instructions do not constitute a bottleneck,

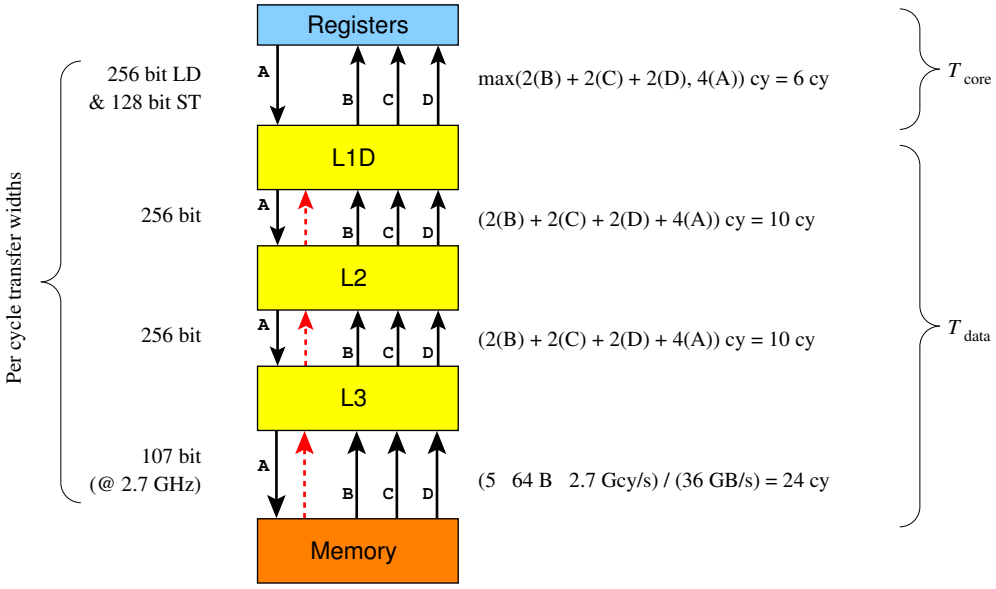


Figure 3.7: Single-core ECM model for the Schönauer triad benchmark (`A(:)=B(:)+C(:)*D(:)`) on an Intel Sandy Bridge processor at 2.7 GHz. The indicated cycle counts refer to eight loop iterations, i.e., a full cache line length per stream. The transfer width per cycle for refills from memory to L3 is derived from the measured STREAM bandwidth limit of 36 GB/s. Dashed arrows indicate write-allocate transfers.

because only two ADDs and two MULTs are needed. Overall, twelve μops must be executed per unit of work, not counting the loop counter and branch "mechanics," which is justified because its impact can be minimized by inner loop unrolling, and because the corresponding execution ports have free resources anyway. Hence, the code has a (useful) instruction throughput of two μops per cycle, which is far below the core's capabilities. The in-core performance is limited by load/store throughput, and we have $T_{\text{core}} = 6\,\text{cy}$. If the data set fits into the L1 cache, $T_{\text{data}} = 0$, and T_{core} is all that is needed to predict an upper performance limit.

If the working set is larger than the L1 cache, calculating T_{data} demands an accurate determination of the real data traffic through the memory hierarchy. In addition to LOADs and STOREs that can be found in the code, every write miss on array `A(:)` triggers a cache line write-allocate transfer to the L1 cache. This is indicated by the dashed arrows in Fig. 3.7. Since the buses between cache layers can transfer half a cache line per cycle, ten cycles each are needed for the data transfers between L2 and L1, and between L3 and L2, respectively. The achievable memory bandwidth of 36 GB/s leads to a per-cycle effective transfer width of 107 bits, which adds another 24 cy. In the worst case, these contributions must be added up to the memory hierarchy level where the data resides. For instance, the most conservative limit for data in memory would be $T_{\text{data}}^{\text{max}} = 44\,\text{cy}$.

Figure 3.8 shows how the different parts can be put together to arrive at an estimate for the execution time. In the worst case, the contributions to T_{data} can neither overlap with each other nor with T_{core}, leading to $T = 50\,\text{cy}$ for data in memory, 26 cy for L3, and 16 cy for L2 (see Fig. 3.8a). On the other hand, assuming full overlap beyond the L2 cache (see Fig. 3.8c), the minimum possible execution times are $T = 24\,\text{cy}$, 16 cy, and 16 cy, respectively. The only well-

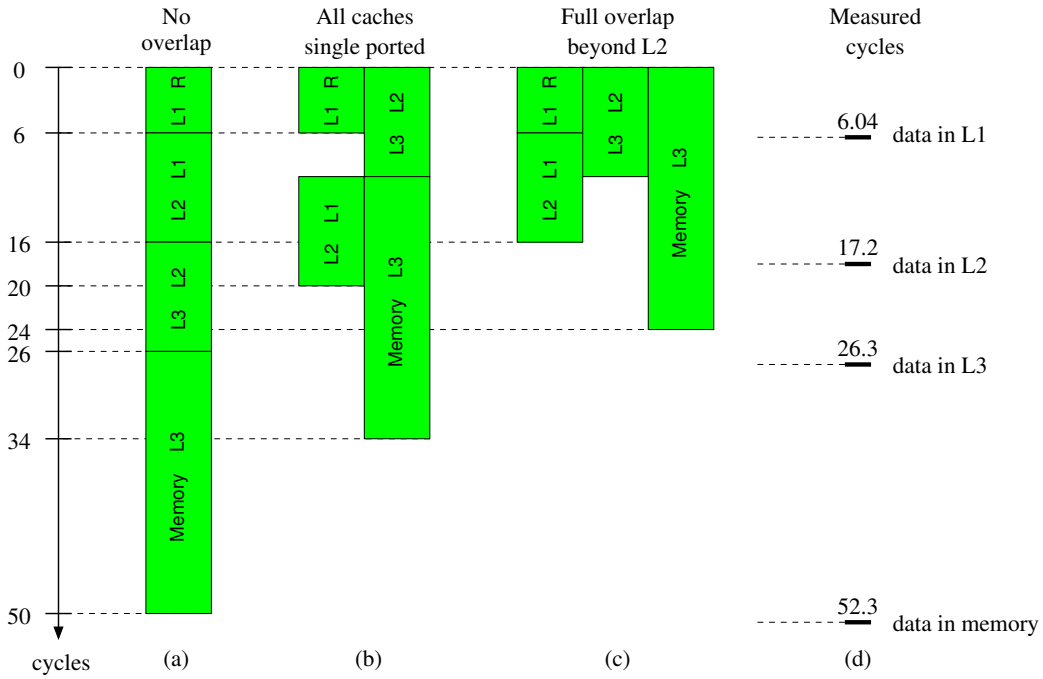


Figure 3.8: (a)–(c): Single-core timeline visualizations of the ECM model with cycle estimates for eight iterations (length of one cache line) of the Schönauer triad benchmark on Sandy Bridge, with different overlap assumptions: (a) no overlap between all contributions from data transfers, (b) overlap under the condition that all caches are single-ported, i.e., can only talk to one immediately neighboring cache level at a time, (c) full overlap of all cache line transfers beyond L2. For comparison the rightmost column (d) shows measurements in cycles per eight iterations at the base clock frequency of 2.7 GHz, with the working set residing in different memory hierarchy levels.

known fact in terms of overlap is that the L1 cache is single-ported, which is why no overlap is assumed even in the latter case. Note that this no-overlap condition is only valid for cycles in which the L1 cache is actually busy serving either the L2 cache or the registers; if the core executes instructions other than LOADs and STOREs, partial or full overlap of T_{core} and T_{data} is possible (see later for an example).

Assuming the non-overlap condition for all cache levels, we arrive at the situation depicted in Fig. 3.8b: Contributions can only overlap if they involve a mutually exclusive set of caches. We then get a prediction of $T = 34\,\text{cy}$ for in-memory data, 20 cy for L3, and again 16 cy for L2 (the latter cannot be shown in the figure).

Figure 3.8d shows measured execution times for comparison. We must conclude that there is no overlap taking place between any contributions to T_{core} and T_{data}. Note that this analysis is valid for a single type of processor, and that other microarchitectures may show different behavior.

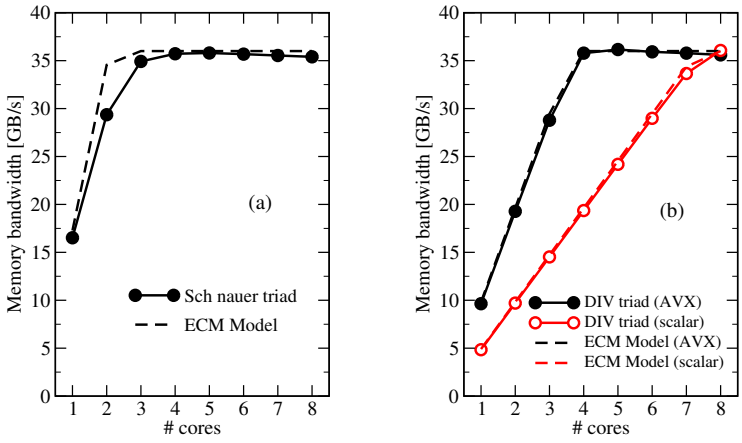


Figure 3.9: Multicore scaling of (a) the memory-bound Schönauer triad benchmark and (b) the modified triad with a divide (`A(:)=B(:)+C(:)/D(:)`), in comparison with the corresponding ECM models (dashed lines) on a 2.7 GHz Sandy Bridge chip. The model for (a) assumes no overlap, while the model in (b) assumes full overlap of T_{core} with T_{data}.

Multicore scaling

All resources in the Sandy Bridge processor chip, except for the memory interface, scale with the number of cores. Hence we predict good scalability of the benchmark loop up to eight cores if the data resides in the L3 cache. Indeed we see a speedup of 7.8 from one to eight cores (see the arrow in Fig. 3.3). In the memory-bound regime we expect scalability up to the bandwidth limit of 107 bits/cy, which is a factor of 2.09 larger than the single-core bandwidth prediction of $320\,\text{bytes}/50\,\text{cy} = 51.2\,\text{bits/cy}$. The performance of the Schönauer triad loop should thus saturate at three cores, with a small speedup from two to three. 3.9 shows a comparison of the model with measurements on a Sandy Bridge chip at 2.7 GHz. The model tracks the overall scaling behavior well, especially the number of cores where saturation sets in. The perfect scaling assumption is slightly optimistic, however, near the saturation point. Note that one can expect the same general characteristics for all loop kernels that are strongly load/store-bound in the L1 cache if the data traffic volume between all cache levels is roughly constant. At $t = 2$, the model over-predicts the performance by about 15%. This deviation is yet to be investigated.

The performance of the Schönauer vector triad is completely bound by data transfer in all memory hierarchy levels including L1. The ECM model should also be able to predict the performance and scaling behavior of loops that are limited by other resources. One example is a modified vector triad that uses a divide operation instead of the multiplication between arrays `C(:)` and `D(:)`. The throughput of the double-precision full-width AVX divide on the Sandy Bridge microarchitecture is 44 cycles if no shortcuts can be taken by the hardware [23], while the throughput of a scalar divide is 22 cycles. All required loads and stores in the L1 cache can certainly be overlapped with the large-latency divides, leading to an in-core execution time of 88 cy and 172 cy, respectively, for one unit of work with the AVX and scalar variants. In this case, the single-portedness of the L1 cache is not applicable, since the in-core code is not load/store-bound. Even if no overlap takes place in the rest of the hierarchy, the $10 + 10 + 24 = 44$ additional cycles for data transfers (see Fig. 3.7) can be hidden behind the in-core time. The results in Fig. 3.9b show a very good agreement of the ECM model with the

measurements. Interestingly, the prediction is now very accurate also near the saturation point.

3.4.4 Conclusions and best practices for applying the ECM model

Out of the four underlying assumptions in the roofline model (see Sect. 3.2.2), the ECM model requires only the streaming assumption, since latency effects are not modeled. The model is primarily suited for providing a prediction of the single-core performance of a loop. As was shown in Sect. 3.4.3, the scalability prediction (3.24) works best if the in-core execution is not dominated by LOADs and STOREs, i.e., if there is considerable "time slack" in the memory hierarchy (see Fig. 3.9b). If the in-core execution is strongly limited by the LOAD and STORE ports, the ECM model prediction, while still accurate for a single core, is too optimistic in the vicinity of the saturation point, i.e., the linear scaling assumption in (3.24) works well only if there is some bandwidth headroom left (see Fig. 3.9a). The reason for the overhead or contention effect that causes this deviation is unknown and is not part of the ECM model. Note that non-temporal stores, due to their strong coupling to the memory interface, can as of now not be accurately incorporated into the model.

The question arises as to why it is so interesting to accurately predict the single-core performance, especially if enough cores are available to eventually reach the saturation point. This was the case in Fig. 3.9b, where even the scalar code was able to saturate the memory interface, if only just. It turns out that knowing about how much time goes into core execution vs. data transfer opens the possibility for assessing the potential of code optimizations much more accurately than with the roofline model:

- If $T_{\mathrm{core}} \gg T_{\mathrm{data}}$, avoiding slow data paths in the memory hierarchy (which is commonly the most promising approach to code optimizations) will not result in a big performance improvement. In this case bandwidth saturation can often not be achieved with the available number of cores. Hence, as a rule of thumb, chip-scalable loops should be optimized for more efficient code execution in the core, e.g., by SIMD vectorization, elimination of pipeline stalls, simultaneous multi-threading (SMT), and avoiding costly operations such as floating-point divides, square roots, or more complex functions. Thinking in terms of the roofline model, such optimizations would typically lead to increasing P_{max}. Since performance is dominated by in-core effects, it will be roughly proportional to the clock frequency of the processor unless saturation sets in. Another consequence is that typical temporal blocking techniques for stencil algorithms, which are based on increasing the computational intensity, will not improve performance if the stencil update code is very complex or non-vectorizable.

- If $T_{\mathrm{core}} \ll T_{\mathrm{data}}$, data transfers are the limiting factor for chip performance. Moving less data across slow data paths is then the most expedient option for speeding up the loop: Non-temporal stores, outer loop unrolling, loop blocking, temporal blocking, are typical strategies to follow. In terms of the roofline model, these optimizations will result in an increased computational intensity. Surprisingly, even if a loop is strongly bound by data transfers, a large fraction of the overall serial execution time $T_{\mathrm{core}} + T_{\mathrm{data}}$ is spent in parts of the hardware whose performance is proportional to the clock speed. This was shown prominently with the vector triad benchmark (see Fig. 3.8a), in which more than half of the cycles came from on-chip data paths on the hardware under consideration. Hence, the clock frequency has a big impact on the single-core performance even if the data comes

from main memory; to lowest order one can often assume that they are proportional unless there is saturation, just as in the core-dominated case.

A further advantage of an accurate modeling of on-chip scaling behavior is that knowing the scaling properties of a code leads to accurate guidelines for choosing an optimal operating point for minimal energy to solution with controlled (or predictable) loss in performance. The energy consumption properties of processors depend crucially on how many cores are in use, on the processor's clock speed, and on the type of code being executed. These issues will be covered in detail in Chapter 4.

Finally it must be stressed that the ECM model is not only useful for simple benchmark kernels. It has also been used successfully to describe the performance and scaling properties of stencil smoothers [9, 10] and medical image reconstruction kernels [6] (see also Chapter 6). It is the first approach that can successfully model the single-thread performance and on-chip scalability of data streaming applications on multicore processors.

3.5 Chapter summary

In this chapter, two approaches to white-box performance modeling on the chip level were presented: The well-known roofline model and the recently introduced Execution-Cache-Memory (ECM) model. It was shown how erratic data access can be incorporated into the roofline model, not for performance prediction but for learning more about data transfer overhead and optimization opportunities. The ECM model can be seen as a refinement of the roofline model, taking into account not only core execution and a single bottleneck, but also the time for data transfers through the memory hierarchy and multicore scaling. It was demonstrated that the ECM model can predict the single-core performance and intra-chip scaling of streaming loop kernels, providing guidelines for optimization approaches.

To the author's knowledge, the ECM model is the first model that can successfully describe single-threaded performance and multicore scaling for modern processors.

Chapter 4

Performance and power [1]

Power dissipation and energy consumption are aspects of scientific computing that have been moving into the focus of research in recent years. Energy and cooling costs for running large-scale clusters are comparable to the pure hardware costs, and bringing the dissipated heat out of a server case is a challenge of its own. Scientific users seem to have little influence on these matters, but they may be forced to find energy-optimal operating points for their software in the very near future, when allocations on parallel systems are granted not in terms of CPU cycles but also in terms of consumed energy.

As it turns out, users who know about the performance properties of their implementations can save a significant amount of energy, without compromising time to solution, by taking care that the serial program code is as fast as possible (most probably by applying a suitable white-box performance model) and choosing some tunable parameters: clock speed and number of cores used. In this chapter we develop a simple but meaningful power model for multicore processors that captures the influence of these essential factors on the energy consumption of running code. Combining this power model with the ECM model then enables combined, predictive energy and performance modeling on the chip level. The Intel "Sandy Bridge EP" server processor is used for all benchmarks and energy measurements; it is the first Intel chip for which accurate energy consumption information is accessible from user code.

4.1 Power dissipation and performance on multicore

In this section the power dissipation properties of the Sandy Bridge processor are investigated by studying several benchmark codes. Then a simple power model is constructed and the most interesting features for the "energy to solution" metric with respect to clock frequency, number of cores utilized, and serial code performance are derived from it. While the model is too coarse to provide quantitative predictions, the qualitative insights are extremely useful.

4.1.1 Power and performance of benchmarks vs. active cores

In order to study energy consumption for various different code patterns, a couple of test codes were chosen. Each of those shows a somewhat typical performance behavior for a certain class of applications. Performance, CPI (cycles per instruction), and power dissipation were measured on a Sandy Bridge EP (Xeon E5-2680) chip, with respect to the number of cores used. The

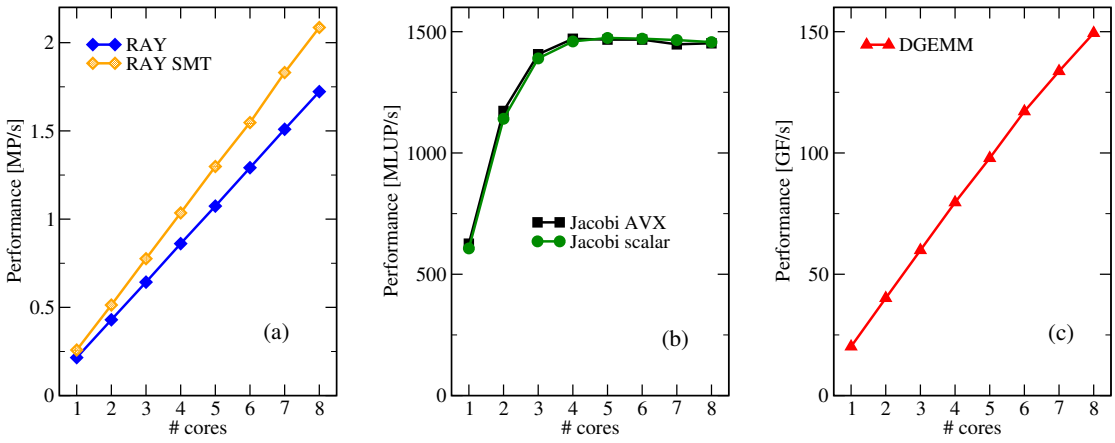


Figure 4.1: Performance of the benchmark codes on a Sandy Bridge chip with respect to the number of active cores at the base frequency of 2.7 GHz.

"turbo mode" feature was deliberately ignored, and the chip was given a sufficient warm-up time before the actual measurements were taken. Without the warm-up, variations of up to 10% in power dissipation could be observed across multiple runs with the same code. The energy measurements were done using the `likwid-perfctr` tool from the LIKWID tool suite.

In the following, the benchmarks are briefly described together with performance and power data with respect to the number of cores used (see Figs. 4.1 and 4.2).

RAY

is a small, MPI-parallel, master-worker style ray-tracing program, which computes an image of 15000^2 gray-scale pixels of a scene containing several reflective spheres. Performance is reported in million pixels per second (MP/s).

Scalability across the cores of a multicore chip is perfect (see Fig 4.1a), since all data comes

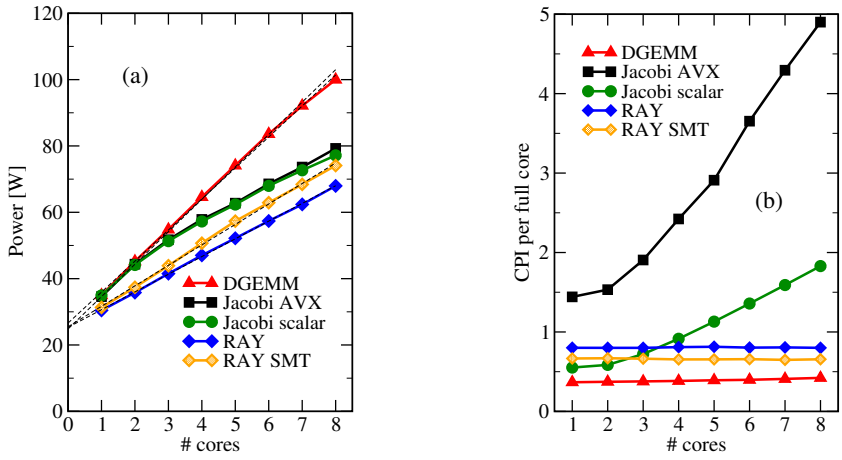


Figure 4.2: (a) Power dissipation and (b) cycles per instruction of the benchmark codes with respect to the number of cores used, at the base frequency of 2.7 GHz.

from the L1 cache, load imbalance is prevented by dynamic work distribution, there is no synchronization, and only infrequent communication of computed tiles with the master process, which is pinned to another socket and thus not taken into account in the analysis. The code is purely scalar and shows a mediocre utilization of the core resources with a CPI value of about 0.8 (see Fig 4.2b). It benefits to some extent from the use of simultaneous multi-threading (SMT), which reduces the CPI to 0.65 per (full) core for a speedup of roughly 15%. At the same time, power dissipation grows by about 8% and is roughly linear in the number of cores used for both cases (see Fig. 4.2a).

Jacobi

is an OpenMP-parallel 2D Jacobi smoother (four-point stencil) used with an out-of-cache data set (4000^2 lattice sites at double precision). Being bandwidth-bound with an effective code balance of 6 B/F [21], it shows the typical saturation behavior described by the ECM model for streaming codes. Performance is reported in million lattice site updates per second (MLUP/s), where one update comprises four flops. Hence, we expect a saturation performance of 6 GF/s or 1500 MLUP/s on a full Sandy Bridge chip, which is fully in line with the measurement (see Fig. 4.1b).

This benchmark was built in two variants, an AVX-vectorized version and a scalar version, to see the influence of data-parallel instructions on power dissipation. Both versions have very similar scaling characteristics, with the scalar code being slightly slower below the saturation point, as expected. The performance saturation is also reflected in the CPI rate (Fig. 4.2b), which shows a linear slope after saturation. Surprisingly, although there is a factor of 2.5-3 in terms of CPI between the scalar and AVX versions, the power dissipation hardly changes (Fig. 4.2a). Beyond the saturation point, the slope of the power dissipation decreases slightly, indicating that a large CPI value is correlated with lower power (cores waiting for data). However, the relation is by no means inversely proportional, just as for the RAY benchmark.

Note that the particular choice of problem size causes the layer condition (see Sect. 5.1.2) to fail in the L1 cache, leading to increased L2 cache traffic compared to a perfect blocking strategy. Together with the increased load/store throughput at scalar execution (see Sect. 2.4.1) this means that both code variants show very similar serial performance. Moreover, the compiler employs half-wide (i.e., SSE) LOAD instructions in the AVX case in order to decrease the probability of split loads across cache line boundaries, which incur penalty cycles [23]. This is the reason for the CPI value not being a factor of four lower for AVX in the saturated case. These deficiencies could be fixed by proper data alignment and probably the use of SIMD intrinsics.

DGEMM

performs a number of multiplications between two dense double precision matrices of size 5600^2, using the thread-parallel Intel MKL library that comes with the Intel compiler (version 10.3 update 9). Performance is reported in GF/s.

The code scales almost perfectly with a speedup of 7.5 on eight cores, and reaches about 86% of the arithmetic peak performance on the full Sandy Bridge chip at a CPI of about 0.4 (i.e, 2.5 instructions per cycle). The power dissipation is almost linear in the number of cores used (Fig. 4.2a).

DGEMM achieves the highest power dissipation of all codes considered here. Note that

at the base frequency of 2.7 GHz, the thermal design power (TDP) of the chip of 130 W is not nearly reached, not even with the DGEMM code. With turbo mode enabled (3.1 GHz at eight cores) one can observe a maximum sustained power of 122 W. The Sandy Bridge chip can exceed the TDP limit for short time periods [41], but this was not investigated here.

Surprisingly, the power dissipation of DGEMM is identical to the Jacobi code (scalar and AVX versions) as long as the latter is not bandwidth-bound, whereas the RAY benchmark draws about 15% less power at low core counts. This can be attributed to the mediocre utilization of the execution units in RAY, where some long-latency floating-point divides incur pipeline stalls, and the strong utilization of the full cache hierarchy by the Jacobi smoother.

4.1.2 Power and performance vs. clock frequency for all benchmarks

Figure 4.3a shows the power dissipation of all benchmarks with respect to the clock frequency ($f = 1.2 \ldots 2.7\,\mathrm{GHz}$) when all cores are used (all virtual cores in case of the SMT variant of RAY). The Sandy Bridge chip only allows for a "global" frequency setting, i.e., all cores run at the same clock speed. The solid lines are least-squares fits to a second-degree polynomial,

$$W(f) = W_0 + w_1 f + w_2 f^2 \,, \tag{4.1}$$

for which the coefficient of the linear term is very small compared to the constant and the quadratic term. The quality of the fit suggests that the dependence of dynamic power dissipation on frequency is predominantly quadratic with $7\,\mathrm{W/GHz^2} < w_2 < 10\,\mathrm{W/GHz^2}$, depending on the code characteristics. Note that one would naively expect a cubic dependence in f if the core voltage were adjusted to always reflect the lowest possible setting at a given frequency. Since we cannot look into the precise algorithm that the hardware uses to set the core voltage, we use the observed quadratic function as phenomenological input to the power model below, without questioning its exact origins. The conclusions we draw from the model would not change qualitatively if $W(f)$ were a cubic polynomial, or any other monotonically increasing function with a positive second derivative.

It is plausible that the "baseline power" $W_0 \approx 25\,\mathrm{W}$ is largely independent of the type of code, since part of it can be associated with the chip leakage power. It may vary depending on the actual state that idle cores assume, i.e., when not executing code. Modern processor cores usually have several power-saving states, which differ greatly in their power dissipation per core and also in the time it takes to return to normal operation. One should also note that an extrapolation to $f = 0$ is problematic here, so that the estimate for W_0 is very rough. The extrapolation to zero cores in Fig. 4.2 yields roughly the same value for W_0, which is reassuring. However, the actual power dissipation with all cores in the halt state would be much lower due to advanced power gating techniques. Hence, giving W_0 a concrete physical meaning is debatable and we regard it as a pure model parameter that accommodates all power contributions that do not vary with the number of active cores and the clock frequency. When system components beyond the chip are included in the model, their power dissipation is mostly constant and can thus be included in W_0 (see below).

Figure 4.3b shows the single-core performance of all benchmarks with respect to clock frequency, normalized to the level at $f = 2.7\,\mathrm{GHz}$. As expected, the codes with near-perfect scaling behavior across cores show a strict proportionality of performance and clock speed, since all required resources run with the core frequency. In case of the Jacobi benchmark the linear extrapolation to $f = 0$ has a non-zero y-intercept, because resources are involved that

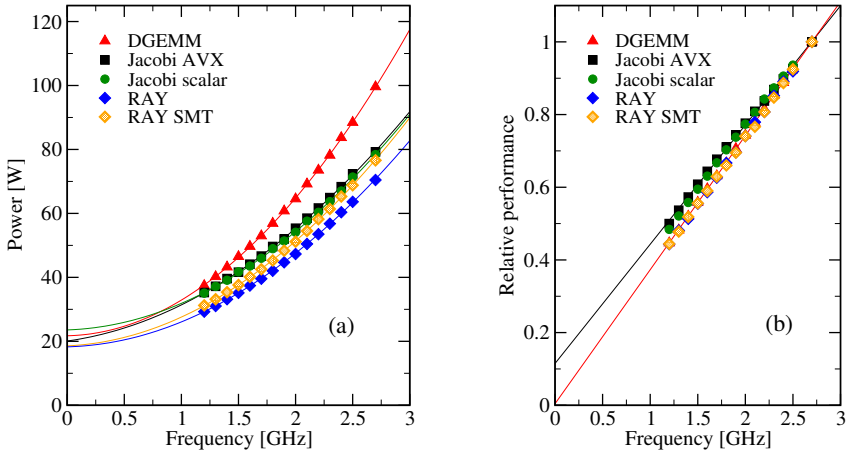


Figure 4.3: (a) Power dissipation of a Sandy Bridge chip with respect to clock speed for the benchmark codes. All eight physical cores were used in all cases, and all 16 virtual cores for the "RAY SMT" benchmark. The solid lines are least-squares fits to a second-degree polynomial in f. (b) Relative performance versus clock speed with respect to the 2.7 GHz level of single-core execution for the benchmark codes. Two processes on one physical core were used in case of RAY SMT. The solid lines are linear fits to the Jacobi AVX and DGEMM data, respectively.

are clocked independently of the CPU cores. The ECM model predicts this behavior if one can assume that the maximum bandwidth of the memory interface is constant with varying frequency.

Figure 4.4 shows the saturated memory bandwidth of a Sandy Bridge chip with respect to clock speed. If we assume that the core frequency should not influence the memory interface, there is no explanation for the drop in bandwidth below about 1.7 GHz: The ECM model predicts constant bandwidth for a streaming kernel like, e.g., the Schönauer triad (one may speculate whether a slow Uncore clock speed could cause a lack of outstanding requests to the memory queue, reducing achievable bandwidth). For the purpose of developing a multicore power model, we neglect these effects and assume a strictly linear behavior (with zero y-intercept) of

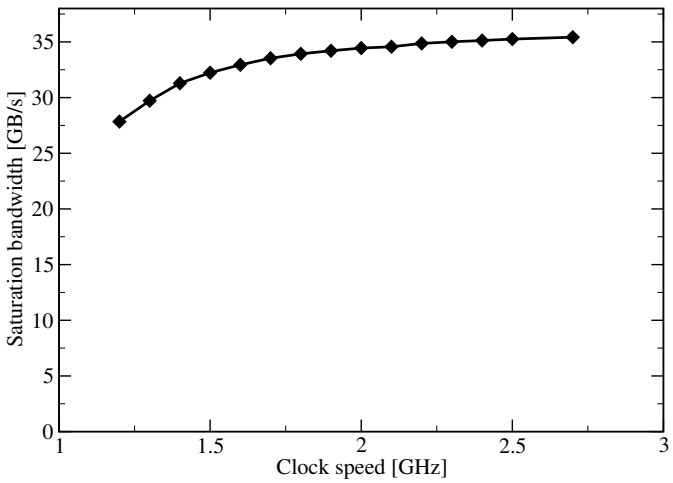


Figure 4.4: Maximum memory bandwidth (saturated) versus clock frequency of a Sandy Bridge chip. See [42] for a detailed account of the influence of clock speed on bendwidth.

performance vs. clock speed in the non-saturated case.

4.1.3 Conclusions from the benchmark data

In order to arrive at a qualitative model that connects the power and performance features of the multicore chip, some generalizing conclusions must be drawn from the data that was discussed in the previous sections.

From Fig. 4.3a, we conclude that the dynamic power dissipation is a quadratic polynomial in the clock frequency and parametrized by w_2 in (4.1). w_2 depends on the type of code executed, and there is some (inverse) correlation with the CPI value (see Fig. 4.2), but a simple mathematical relation cannot be derived. The linear part w_1 is generally small compared to w_2.

A linear extrapolation of power dissipation vs. the number of active cores to zero cores (dashed lines in Fig. 4.2a) shows that the baseline power of the chip is $W_0 \approx 25\,\mathrm{W}$, independent of the type of running code. In case of the bandwidth-limited Jacobi benchmark only the one- and two-core data points were considered in the extrapolation. The result for W_0 is also in line with the quadratic extrapolations to zero clock frequency in Fig. 4.3a. Note that W_0, as a phenomenological model parameter, is different from the documented "idle power" of the chip, which is considerably lower due to power gating mechanisms.

From the same data we infer a linear dependence of power dissipation on the number of active cores t in the non-saturated regime,

$$W(f,t) = W_0 + (W_1 f + W_2 f^2)t \ , \tag{4.2}$$

so that $w_{1,2} = t \cdot W_{1,2}$. Although the power per core rises more slowly in the saturation regime, we regard this as a second-order effect and neglect it in the following: The fact that a core is active has much more influence on power dissipation than the characteristics of the running code.

As Fig. 4.2a indicates, using both hardware threads (virtual cores) on a physical Sandy Bridge core increases power dissipation due to the improved utilization of the pipelines. The corresponding performance increase depends on the code, of course, so it may be more power-efficient to ignore the SMT threads. In case of the RAY code, however, the increase in power is over-compensated by a larger boost in performance, as shown in Fig. 4.1a. See Sect. 4.2.4 for further discussion.

One of the conclusions from the ECM model was that, in the non-saturated case, performance is proportional to the core's clock speed. Fig. 4.3b suggests that this true for the scalable benchmarks, and approximately true also for saturating codes like Jacobi.

4.2 A qualitative power model

Using the measurements and conclusions from the previous section a simple power model can be derived, which describes the overall power properties of a multicore chip with respect to number of cores used, the scaling properties of the code under consideration, and the clock frequency. As a starting point we choose the "energy to solution" metric, which quantifies the energy required to solve a certain compute problem and is thus restricted to strong scaling scenarios. This is not a severe limitation, since weak scaling is usually applied in the massively (distributed-memory) parallel case, where the relevant scaling unit is a node or a ccNUMA domain (which is usually

a chip). The optimal choice of resources and execution parameters on the chip level, where the pertinent bottlenecks are different, are usually done at a constant problem size.

The following basic assumptions go into the model:

1. The whole N_c-core chip dissipates some baseline power W_0 when powered on, which is independent of the number of active cores and of the clock speed.

2. An active core consumes a dynamic power of $W_1 f + W_2 f^2$. We will also consider deviations from some baseline clock frequency f_0 such that $f = (1+\Delta\nu) f_0$, with $\Delta\nu = \Delta f / f_0$.

3. At the baseline clock frequency, the serial code under consideration runs at some performance P_0. As long as there is no bottleneck, the performance is linear in the number of cores used, t, and the normalized clock speed, $1+\Delta\nu$. The latter dependence will not be exactly linear if some part of the hardware (e.g., the outer-level cache) runs at its own clock speed. In presence of a bottleneck (like, e.g., memory bandwidth), the overall performance with respect to t is capped by some maximum value P_{roof}:

$$P(t) = \min\left((1+\Delta\nu) t P_0, P_{\mathrm{roof}}\right) = \min\left(t P_0 f / f_0, P_{\mathrm{roof}}\right) \ . \tag{4.3}$$

   This is just an extension of (3.24) for varying clock speed. Note that we have not included an explicit frequency dependence of the saturated performance. If the applicable bottleneck is within the cache hierarchy, the model can be easily extended to accommodate this case.

Since time to solution is inverse performance, the energy to solution becomes

$$E = \frac{W_0 + (W_1 f + W_2 f^2) t}{\min\left(t P_0 f / f_0, P_{\mathrm{roof}}\right)} \ . \tag{4.4}$$

A direct consequence of this model is that any increase in performance (P_0 or P_{roof}) leads to proportional savings in energy to solution. Performance is thus the first-order tuning parameter for minimum energy. This general rule will be revisited several times in this work.

The model parameters W_0, W_1, and W_2 must be determined by measurements, as shown above for the example benchmarks. Since W_1 and W_2 depend on the actual loop code, this measurement must be repeated for every loop if the application is complex. For qualitative results it is sufficient to assume approximate values that reflect general loop properties known from code analysis and performance modeling (memory-boundedness, SIMD vectorization, pipeline utilization). Choi et al. [43] have derived a "roofline model of energy," which relies on microbenchmarking and a refined power measurement infrastructure to determine the energy consumption of elementary operations such as flops and data transfers, and then allows to parametrize the power dissipation of a chip over a wide range of the computational intensity. Their model is more targeted toward design space exploration and comparisons of different architectures, and they do not explore the core and frequency dependence of energy consumption. In principle, however, the parameters W_0, W_1, and W_2 could be determined by their method as well.

Note that it is out of the scope of the model to study different strategies for dynamic voltage and frequency scaling (DVFS); the exact algorithm a processor uses to set the core voltage at a certain frequency is taken as-is, and is hidden in the model parameters W_0, W_1, and W_2.

4.2.1 Minimum energy with respect to the number of active cores

Due to the assumed saturation of performance with t, we have to distinguish two cases:

Case 1: $tP_0 f/f_0 < P_{\text{roof}}$ Performance is linear in the number of cores, so that (4.4) becomes

$$E = \frac{W_0 + (W_1 f + W_2 f^2)t}{tP_0 f/f_0} , \tag{4.5}$$

and E is a decreasing function of t:

$$\frac{\partial E}{\partial t} = -\frac{W_0}{t^2 P_0 f/f_0} < 0 . \tag{4.6}$$

Hence, the more cores are used, the shorter the execution time and the smaller the energy to solution.

Case 2: $tP_0 f/f_0 > P_{\text{roof}}$ Performance is constant in the number of cores, hence

$$E = \frac{1}{P_{\text{roof}}} \left(W_0 + (W_1 f + W_2 f^2)t\right) \tag{4.7}$$

$$\Rightarrow \quad \frac{\partial E}{\partial t} = \frac{1}{P_{\text{roof}}} \left(W_1 f + W_2 f^2\right) > 0 . \tag{4.8}$$

In this case, energy to solution grows with t, with a slope that is proportional to the dynamic power, while the time to solution stays constant; using more cores is thus a waste of energy. Leaving cores idle to save energy is known as "dynamic concurrency throttling" (DCT) [44].

For codes that show performance saturation at some t_{s}, it follows that energy (and time) to solution is minimal just at this point:

$$t_{\text{s}} = \frac{P_{\text{roof}}}{P_0 f/f_0} . \tag{4.9}$$

If the code scales to the available number of cores, case 1 applies and one should use them all.

4.2.2 Minimum energy with respect to code performance

Since the serial code performance P_0 only appears in the denominator of (4.4), increasing P_0 leads to decreasing energy to solution unless $P = P_{\text{roof}}$. A typical example for this scenario is the SIMD vectorization of a bandwidth-bound code: Using data-parallel instructions (such as SSE or AVX) will generally reduce the in-core execution time (T_{core}), so that P_0 grows and the saturation point P_{roof} is reached at smaller t (see (4.9)). Consequently, the potential for saving energy is twofold: When operating below the saturation point, optimized code requires less energy to solution. At the saturation point, one can get away with fewer active cores to solve the problem at maximum performance.

The energy to solution is also inversely proportional to the saturated performance P_{roof} (if saturation applies), thus P_{roof} has the same energy-saving potential as P_0. However, since P_{roof} is typically determined by a bottleneck in the memory hierarchy, code optimizations to increase P_{roof} are typically targeted toward higher computational intensity (see also Sect. 3.2.3).

4.2.3 Minimum energy with respect to clock frequency

We again have to distinguish two cases:

Case 1: $tP_0 f/f_0 < P_{\text{roof}}$ Energy to solution is the same as in (4.5) and $f = (1+\Delta\nu)f_0$, so that

$$\frac{\partial E}{\partial \Delta\nu} = \frac{f_0^2}{tP_0}\left(W_2 t - \frac{W_0}{f^2}\right) . \tag{4.10}$$

The derivative is positive for large f; setting it to zero and solving for f thus yields the frequency for minimal energy to solution:

$$f_{\text{opt}} = \sqrt{\frac{W_0}{W_2 t}} . \tag{4.11}$$

A large baseline power W_0 forces a large clock frequency to "get it over with" ("clock race to idle"). Depending on W_0 and W_2, f_{opt} may be larger than the highest possible clock speed of the chip, so that there is no energy minimum. This may be the case if one includes the rest of the system in the analysis (i.e., memory, disks, etc.). On the other hand, a large dynamic power W_2 allows for smaller f_{opt}, since the loss in performance is over-compensated by the reduction in power dissipation. The fact that f_{opt} does not depend on W_1 just reflects our assumption that the serial performance is linear in f.

Since t appears in the denominator in (4.11), it is tempting to conclude that a clock frequency reduction can be compensated by using more cores, but the influence on E has to be checked by inserting f_{opt} from (4.11) into (4.5):

$$E(f_{\text{opt}}) = \frac{f_0}{P_0}\left(2\sqrt{\frac{W_0 W_2}{t}} + W_1\right) \tag{4.12}$$

This confirms the conjecture that more cores at lower frequency save energy below the saturation point. At the same time, performance at f_{opt} is

$$P(f_{\text{opt}}) = \frac{f_{\text{opt}}}{f_0} tP_0 = \frac{P_0}{f_0}\sqrt{\frac{W_0 t}{W_2}} , \tag{4.13}$$

hence it grows with the number of cores: trading cores for clock slowdown does not compromise time to solution.

However, if t is fixed, (4.13) also tells us that, if $f_{\text{opt}} < f_0$, performance will be smaller than at the base frequency f_0, although the energy to solution is also smaller. This may be problematic if t cannot be made larger to compensate for the loss in performance. In this case the energy to solution metric is insufficient and one has to choose a more appropriate cost function, such as energy multiplied by runtime:

$$C = \frac{E}{P} = \frac{W_0 + (W_1 f + W_2 f^2)t}{(tP_0 f/f_0)^2} . \tag{4.14}$$

Differentiating C with respect to $\Delta\nu$ gives

$$\frac{\partial C}{\partial \Delta\nu} = -\frac{2W_0 + W_1 ft}{(f/f_0)^3 P_0^2} < 0 , \tag{4.15}$$

because $W_0, W_1 > 0$. Hence, a higher clock speed is always better if C is chosen as the relevant cost function. Note that a large baseline power W_0 emphasizes this effect, e.g., when the whole system is taken into account (see also above in the discussion of f_{opt}).

The question remains how to deal with the code slowdown, since a machine running at lower clock speed will deliver less "science per day," and this is what the user typically cares about. One option is to invest the money saved on the power bill in a larger system. See Sect. 4.3 for an analysis of this point of view.

Case 2: $tP_0 f/f_0 > P_{\mathrm{roof}}$ Beyond the saturation point, energy to solution is the same as in (4.7), so it grows with the frequency: The clock should be as slow as possible. Together with the findings from case 1 this means that minimal energy to solution is achieved when using all available cores, at a clock frequency which is so low (if possible) that the saturation point is right at $t = N_{\mathrm{c}}$.

These results reflect the popular "clock race to idle" rule, which basically states that a processor should run at maximum frequency to "get it over with" and go to sleep as early as possible to eventually save energy. Using the energy to solution behavior as derived above, we now know how this strategy depends on the number of cores used and the ratio of baseline and dynamic power. "Clock race to idle" makes sense only in the sub-saturation regime, and when $f < f_{\mathrm{opt}}$. Beyond f_{opt} (if such frequencies are allowed), the quadratic dependence of power on clock speed will waste energy. Beyond the saturation point, i.e., if $t > t_{\mathrm{s}}$, lower frequency is always better.

4.2.4 Validation of the power model for the benchmarks

The multicore power model has been derived from the benchmarks' power dissipation using considerable simplifications. Hence, it is now important to check whether the conclusions drawn above are still valid for the benchmark codes when looking at the measured energy to solution data with respect to the number of active cores, the clock speed, and the single-core performance.

Figure 4.5 shows energy to solution measurements for the scalable codes (Fig. 4.5a) and the Jacobi AVX benchmark (Fig. 4.5b) versus clock frequency and number of cores, respectively. Comparing the frequency for minimum energy to solution between DGEMM and RAY at eight cores (solid symbols in Fig. 4.5a), we can identify the behavior predicted by (4.11): A large dynamic power factor W_2 leads to lower f_{opt}. The SMT version of RAY consumes more power than the standard version, but, as anticipated above, the larger performance leads to lower energy to solution: Better resource utilization on the core, i.e., optimized code, saves energy; this provides another possible attitude towards the "race to idle" rule. Given the huge amount of optimization potential that is still hidden in many production codes on highly parallel systems, this view must be regarded as even more relevant than optimizing clock speed for a few percent of energy savings.

Eq. (4.11) predicts a larger optimal frequency f_{opt} at fewer cores, which is clearly visible when comparing the four- and eight-core energy data for DGEMM in Fig. 4.5a (solid vs. open triangles). At the same time, fewer cores also lead to larger minimum energy to solution at f_{opt}, which was shown in (4.12).

The Jacobi benchmark shows all the expected features of a code whose performance saturates at a certain number of cores t_{s}: As predicted by the ECM model, the saturation point is shifted to a larger number of cores as the clock frequency goes down; it was derived in Sect. 4.2.1 that this is the point at which energy to solution is minimal. Lowering the frequency,

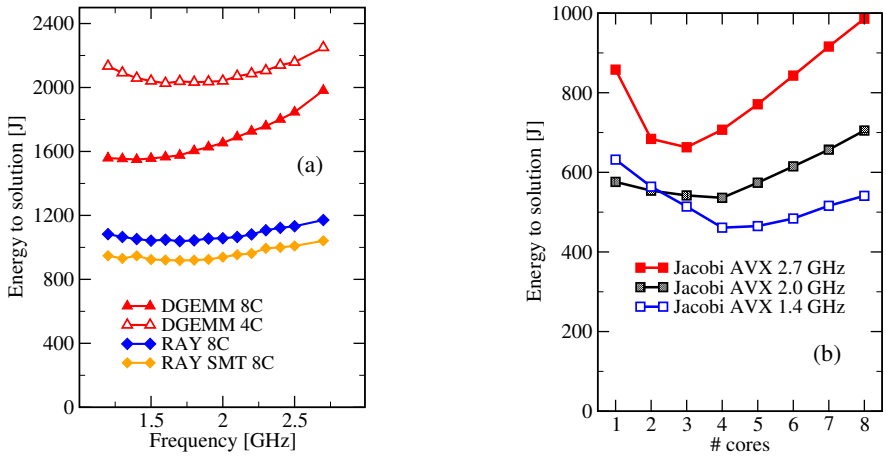


Figure 4.5: Energy to solution for (a) the scalable benchmarks DGEMM (eight and four cores) and RAY (eight cores) versus clock frequency on a Sandy Bridge socket and (b) the Jacobi AVX benchmark versus number of cores at different core frequencies.

t_s gets larger, but energy to solution decreases (see (4.12)). When $t > t_s$, more cores and higher clock speed both are a waste of energy. At $t < t_s$ the Jacobi code is largely frequency-bound and there is an optimal frequency $f_{opt} \approx 2\,\text{GHz}$ with minimal energy to solution. Here we substantiate the prediction from Sect. 4.2.3 that "clock race to idle" is largely counterproductive if we look at the chip's power dissipation only. See also Sect. 7 for a discussion of "race to idle" in the context of a lattice-Boltzmann CFD solver.

In conclusion, although considerable simplifications have been made in constructing the model (4.4), it is able to describe the qualitative behavior of the benchmark applications with respect to energy to solution.

Applying the model in practice to achieve minimum power consumption for a real application may be complex if the code is composed of many parts that take only a small amount of time. Every loop must be analyzed and modeled for performance and power, and clock speed adjustments and DCT must be applied on a loop-by-loop basis via suitable libraries [45] or automatic frameworks [44].

4.3 Consequences for system design

Interesting conclusions for system design can be drawn from the energy to solution model (4.4) when typical requirements in computing center environments are taken into account. While the model predicts that it is possible to save energy by reducing the clock speed to the point where a bandwidth-bound code scales across all cores of the socket, the situation is more complex with scalable code. As shown in Sect. 4.2.3, adjusting the clock speed to get minimum energy to solution may compromise the time to solution. Apart from choosing more appropriate metrics such as the energy-delay product, one can also assume the point of view that a certain system is running at the optimal clock frequency f_{opt}, which is given by (4.11), and then adjust the size of the system to compensate for the performance loss (or gain, if $f_{opt} > f_0$). If one assumes that the price for a system is roughly constant for constant peak performance, the only difference between a system running at f_0 and a system running at f_{opt} is its energy consumption over its

lifetime. The question remains as to if and how much energy, and thus money, can be saved by setting the optimal frequency.

The optimal frequency f_{opt} depends on the baseline power W_0, the dynamic power W_2, and on the number of cores t (see (4.11)). The ratio of power dissipation between the optimized and the base clock frequencies is

$$\frac{W(f_{\text{opt}},t)}{W(f_0,t)} = \frac{W_0 + W_2 f_{\text{opt}}^2 t}{W_0 + W_2 f_0^2 t} = \frac{2W_0}{W_0 + W_2 f_0^2 t} \tag{4.16}$$

if we neglect the usually small linear part in the power law (4.2). Comparing systems of the same size, this ratio is certainly smaller than one. However, if we adjust the size of the "optimized" system by a factor that reflects the chip performance ratio (and assume perfect scaling for applications), we get

$$R = \frac{W(f_{\text{opt}},t)}{W(f_0,t)} \frac{f_0}{f_{\text{opt}}} = \frac{2f_0\sqrt{W_0 W_2 t}}{W_0 + W_2 f_0^2 t} \; . \tag{4.17}$$

The dimensionless ratio R quantifies the energy saving potential of setting an optimal clock speed and adjusting the size of the machine to compensate for the change in performance. It is straightforward to show that $R = 1$ for $W_0 = W_2 f_0^2 t$, and $R < 1$ otherwise.

Using the R metric we can explore the design space of possible parallel machines with different values for W_0, W_2, and t. In the limit of very small $W_0 \ll W_2 f_0^2 t$, which should be regarded as a very desirable goal, we get

$$R_{\text{cool}} \approx \frac{2}{f_0}\sqrt{\frac{W_0}{W_2 t}} = \frac{2f_{\text{opt}}}{f_0} \; . \tag{4.18}$$

A large number t of cores per chip thus favors large, "cool" systems, since f_{opt} is inversely proportional to $\sqrt{t}$. On the other hand, for "hot" machines where $W_0 \gg W_2 f_0^2 t$ we have

$$R_{\text{hot}} \approx \frac{2f_0\sqrt{W_2 t}}{\sqrt{W_0}} = \frac{2f_0}{f_{\text{opt}}} \; . \tag{4.19}$$

Hence, energy can also be saved with a very high clock frequency if W_0 is large ("clock race to idle").

A value of $R = 1$ marks the inflection point where a machine is "optimal," i.e., where it is not possible to save energy by trading clock speed for machine size. If we plot $R(W_0)$, it is the combination $W_2 f_0^2 t$ which determines the shape of the curve, and especially the position of the inflection point $R = 1$ (see Fig. 4.6). At a given value of $W_2 f_0^2 t$, the region left of $R = 1$ is where a clock slowdown (and a correspondingly larger machine) can save energy compared to the baseline clock speed, because the baseline power is small. The region to the right of $R = 1$ is where "clock race to idle" applies. Here the chip is so hot that it is beneficial to run at very high clock speed to "get it over with." The size of the machine can be smaller in this case.[1] Whether these clock speeds are technically accessible is not predicted by the model, of course.

Figure 4.6 shows three different scenarios by choosing a different number of cores t and a different base clock speed f_0. Note that any change in W_2 can be mapped to a proportional change in t, so W_2 was fixed to $1.5\,\text{W}/\text{GHz}^2$, which is roughly the measured value for the Intel

[1] The two regions can be associated with the limits mentioned Seymour Cray's famous quote "If you were plowing a field, which would you rather use: two strong oxen or 1024 chickens?"

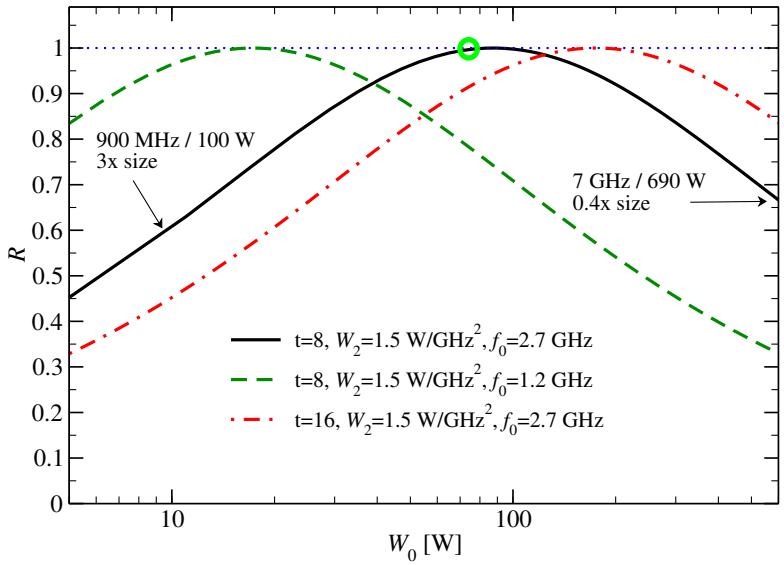


Figure 4.6: Energy saving potential R vs. baseline power W_0 for running a system (with a given number of cores t, a given default clock speed f_0, and a given dynamic power parameter W_2) at the optimal clock speed f_{opt} with a scalable code and adjusting the system size for constant aggregate performance. The circle marks the federal supercomputer system at LRZ Garching (SuperMUC). The baseline power is understood to include the chip's share of the whole system.

Sandy Bridge EP processors in the federal "SuperMUC" system at LRZ Garching. For the case $f_0 = 2.7\,\text{GHz}$ and $t = 8$, the $R = 1$ point is at $W_0 \approx 90\,\text{W}$. All other parameters being equal, a small baseline power of $W_0 = 10\,\text{W}$ leads to a system that can be made 3 times larger, runs at $f = 900\,\text{MHz}$, and dissipates about 20 W per chip (including the dynamic power). On the other hand, if $W_0 = 600\,\text{W}$ we get processors running at $f = 7\,\text{GHz}$ and 1200 W, but the system can be built at 40% of its original size. The circle in Fig. 4.6 marks the position of SuperMUC, whose base frequency is 2.7 GHz at eight cores and an overall baseline power of $W_0 \approx 73\,\text{W}$ per chip. Since this point is very near the maximum where $R = 1$, it is hardly possible to save energy on this system by reducing the clock speed in favor of more hardware. In this sense, SuperMUC is an "ideal machine" for scalable codes, i.e., for programs whose performance scales across the cores of the chip.

These considerations stretch the power model very far, and it is not expected that the numbers derived above have any useful accuracy. However, the model is still good for qualitative design space exploration.

4.4 Chapter summary

This chapter has described a phenomenological power model for multicore processors. By taking the measured static (baseline) power and dynamic power per core as input parameters, the model can predict optimal operating points in terms of clock speed, the number of cores used, and the code performance. Since the latter is the only quantity that goes linearly (or rather inversely) into the model, it is one of the basic premises of the model that "fast" code saves energy. This may be called the "code race to idle" principle.

The model distinguishes *scalable* from *saturating* parallel code. It predicts that saturating code should be run with a number of cores that is just able to reach saturation, at the lowest possible clock speed. For scalable code, the model predicts an optimal clock frequency for minimal energy to solution, which depends on the ratio of static vs. dynamic power dissipation. This frequency may so large that it is not accessible by the hardware. In the latter case, the "clock race to idle" principle applies.

For scalable code the model can be used for design space exploration, leading to a clear concept of "hot" vs. "cool" systems when the baseline power is large or small.

Chapter 5

Structured performance engineering

Now that the concepts of performance, scalability, and white-box performance modeling (Chapter 3), together with its implications on power dissipation (Chapter 4) have been introduced, they can be embedded in a larger setting, which we call *structured performance engineering*. Structured performance engineering can be regarded as a part of software engineering. It is a process in which algorithm and code analysis, performance modeling, and optimization are applied repeatedly to arrive at a well-defined concept of "best possible performance." Since all these components require considerable experience to be applied, it is an accepted fact that structured performance engineering will never be implemented as a turnkey software tool. As described in Chapter 3, it is the *failure* of a performance model that leads to new insights and challenges a previous understanding of the interaction between software and hardware. Especially this point is not compatible with automated tooling that can be applied by anyone.

This chapter summarizes the layout of the performance engineering process, repeating some of the key concepts described earlier. Then, some typical performance patterns are given together with their hardware metric signatures. Patterns support the performance engineering process by formalizing some of its frequently recurring aspects.

5.1 The performance engineering process

The ultimate purpose of running simulation tasks on high performance computers is to solve numerical problems. The *performance* of an algorithm, or rather an implementation, is significant in several respects: Either a given problem should be solved in the least possible amount of time or a larger problem should be solved in an "acceptable" time; in both cases, the used resources must be utilized as efficiently as possible so that overall throughput and return on investment are maximized for all users of a system. This goal was formulated as the "bottleneck computing" paradigm in Chapter 3. Structured performance engineering is a process that helps to reach this ideal situation.

5.1.1 Description of the process

Figure 5.1 summarizes a possible approach to structured performance engineering. The individual components are described in detail in the following.

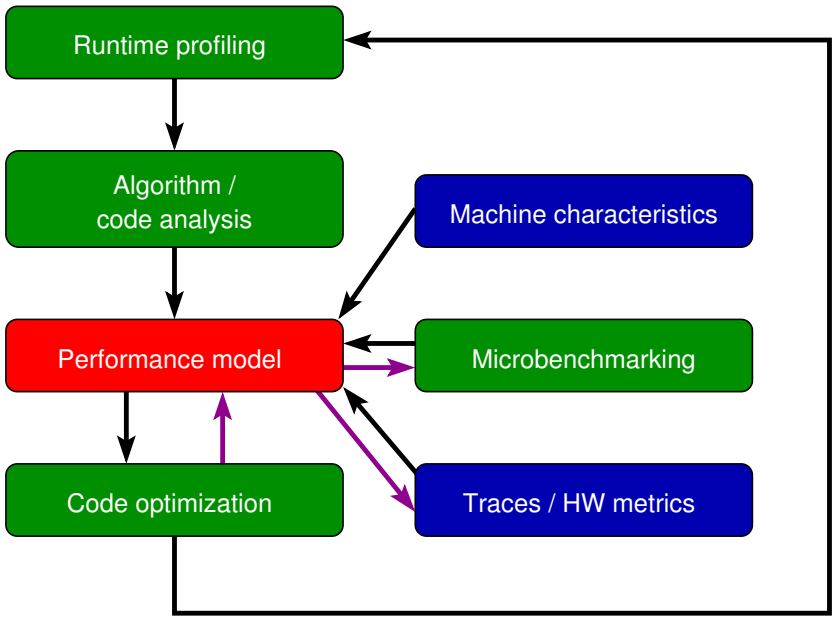


Figure 5.1: The performance engineering process.

- *Runtime profiling.* All performance work with an application starts with runtime profiling. Here the dominant parts of the program, i.e., the hot spots which take most of the runtime, are identified. Such hot spots are usually loops or loop nests. Code analysis then starts with the dominant hot spot and works down from there, given that the goal of "optimal performance" has been reached. Runtime profiling is thus the start and the end of all performance engineering efforts.

- *Algorithm/code analysis.* Analyzing a hot spot requires looking at the algorithm, i.e., the minimum steps to achieve the required task, as well as its specific implementation in the code. In the ideal case, the minimum requirements of the algorithm towards the hardware are already met by the implementation. This may mean, e.g., that the implementation performs the same number of floating-point operations as the algorithm or, more generally, the same amount of "work." Note that there is some uncertainty to this analysis, since the implementation is usually done in a high-level computer language; they way the compiler translates this language to machine code may change above requirements considerably. If the performance model validation (see below) reveals a substantial deviation from expected results, it may be necessary to check the generated assembly code.

  The algorithm and code analysis is one of the crucial inputs for the performance model, and it is probably the step that is least suited for automation via tools.

- *Machine characteristics.* Some parameters of the machine under consideration are usually well documented, such as peak performance, superscalarity, SIMD width, pipeline depths, cache bandwidths and latencies, etc. They can be used as inputs to the performance model, usually making some assumptions about code execution. For instance, a typical assumption could be that hardware prefetching mechanisms work perfectly such that latency effects can be ignored (this is one of the crucial conditions for the ECM model

to work; see Sect. 3.4 for details).

- *Microbenchmarking.* Machine characteristics that cannot be obtained from documentation but are required input for performance modeling may be fixed by accurate microbenchmarking. A prominent example is the maximum main memory bandwidth of a processor chip, whose theoretical limit is often much higher than the achievable value. The precise reasons for this deviation are usually not divulged by the manufacturers.

- *Performance model.* A loop-level performance model may be built with input from algorithm/code analysis, machine characteristics, and microbenchmarking, as shown in Chapter 3. The model should be able to accurately describe the performance of the loop for a given data set, thereby identifying the relevant bottlenecks. This approach works best if the minimum requirements of the algorithm with respect to work and resources are also the minimum requirements of the implementation, and if these requirements can be satisfied by the maximum capabilities of the machine on which the code is executed.

  If the model prediction deviates from performance measurements, the assumptions used for building the model are challenged. Whatever the reasons may be, this is an opportunity to refine and advance the model towards better accuracy, which leads to new insights. See Part II for examples. Of course, in some situations it is very hard to come up with an accurate predictive model. One prominent example is sparse matrix-vector multiplication, for which it was shown in Sect. 3.3.2 that, although predictive modeling is often ruled out due to the erratic memory access, a model-based approach can still be used to learn about the exact overhead caused by this hazard.

  There is an intimate two-way connection between modeling and microbenchmarking. Sometimes it is not clear from the start which microbenchmark is most suitable for exploring a certain aspect of an architecture. The model validation may thus lead to the conclusion that some (measured) parameter must be determined in a different way. For example, the achievable memory bandwidth can vary significantly across different ratios of load versus store streams; if the microbenchmark does not reflect this ratio as given in the modeled application code, the bottleneck is not accurately assessed. See Sect. 5.1.2 below for a similar case (non-temporal vs. standard STORE instructions).

- *Traces and hardware metrics.* Regardless of whether the model yields accurate predictions or not, software tools can help to acquire more information about the interaction of the software code with the hardware. Even if the prediction was accurate, it may still be the case that several opposing effects cancel out. Hence, software tools, and especially tools for measuring hardware metrics, are extremely helpful in validating or disproving the model. As an example, the number of cache lines transferred between adjacent cache levels in a certain phase of program execution can be counted. The ECM performance model predicts this number, so that this aspect of the program execution is easily verified. On the other hand, it is the model itself which selects the "interesting" metrics out of the usually hundreds of available options on modern processors. See Sect. 5.2.3 for a systematic overview of hardware metrics and corresponding performance patterns.

- *Code optimization.* With a working performance model it is often possible to predict the consequences of code optimizations. For instance, the ECM model gives an accurate

account of how dominant the core execution time of a loop is in comparison to the contributions from data transfers through the cache hierarchy. If any of those contributions can be reduced by some amount by means of a code optimization, it can be estimated whether the gain is worth the effort. After applying the change, the model must be re-validated.

Once a certain hot spot has been handled in the way described, the cycle starts anew with the next most important loop.

5.1.2 Case study: An OpenMP-parallel 3D Jacobi smoother

The three-dimensional Jacobi solver for the finite-difference discretization of the Laplace equation with Dirichlet boundary conditions on a regular lattice is a well-understood algorithm, which is, while not in wide use for scientific computing, a very useful example for teaching the basics of performance modeling and optimization. In fact, many of the lessons learned with the Jacobi smoother can be generalized to more complex scenarios with stencil-like update schemes, such as the lattice-Boltzmann algorithm (LBM) [46, 47]. We revisit it here, taking the point of view imposed by the performance engineering process.

Listing 5.1 shows a simple implementation. The convergence criterion is unimportant for the modeling and was omitted. A two-grid implementation was chosen where the current time step `phi(i,j,k,t1)` is updated with values from the previous time step `phi(i,j,k,t0)` in the first loop nest, and the updated grid is copied back in the second loop nest. These two loop nests are the only loops in the code (apart from initialization code).

We use a dual-socket server with Intel Sandy Bridge processors as described in Sect. 2.4.1, running at a clock speed of 2.7 GHz. In order to expose the in-socket bottlenecks we perform full-socket runs (eight threads on eight cores) at a constant problem size of 500^3 grid points, for a working set of 2 GB.

Runtime profiling

Profiling reveals that about 63% of the runtime goes into the first loop nest, whereas the remaining 37% are taken up by the second loop nest. The performance model will thus be applied to both loop nests. For brevity we consider both of them together.

Algorithm/code analysis

We choose a "lattice site update" (LUP) as the relevant work unit.

In the first loop, performing one AVX-vectorized update (4 LUPs) from the L1 cache requires six LOADs, five ADDs, one MULT, and one STORE, without any relevant dependencies. For the data traffic analysis we assume that all LOADs ans STOREs go to main memory (since the data set does not fit in any cache) and must be sustained throughout the cache hierarchy. In addition, each incurs a write-allocate transfer of the cache line, so we arrive at a code balance of $B_1 = 64\,\text{bytes/LUP}$, or a computational intensity of $I_1 = 1/64\,\text{LUPs/byte}$.

The second loop has a requirement of one AVX LOAD and one AVX STORE per AVX-vectorized update (4 LUPs), and a code balance of $B_2 = 24\,\text{bytes/LUP}$, or $I_2 = 1/24\,\text{LUPs/byte}$.

Listing 5.1: Naive OpenMP-parallel implementation of the 3D Jacobi algorithm (adapted from [21]).

```
1  double precision :: oos
2  double precision, dimension(:,:,:,:) :: phi
3  integer :: t0,t1
4  t0 = 0 ; t1 = 1 ; oos = 1.d0/6.d0
5
6  allocate(phi(0:imax+1,0:jmax+1,0:kmax+1,0:1))
7  ! initialization code omitted
8  ...
9  ! loop over sweeps
10 do s=1,ITER
11   ! sweep over grid
12 !$OMP parallel
13 !$OMP do schedule(static)
14   do k = 1,kmax
15     do j = 1,jmax
16       do i = 1,imax
17         ! stencil update
18         phi(i,j,k,t1) = oos * ( &
19           phi(i-1,j,k,t0)+phi(i+1,j,k,t0)+phi(i,j-1,k,t0)+ &
20           phi(i,j+1,k,t0)+phi(i,j,k-1,t0)+phi(i,j,k+1,t0) )
21       enddo
22     enddo
23   enddo
24 !$OMP end  do
25   ! copy back
26 !$OMP do schedule(static)
27   do k = 1,kmax
28     do j = 1,jmax
29       do i = 1,imax
30         phi(i,j,k,t0) = phi(i,j,k,t1)
31       enddo
32     enddo
33   enddo
34 !$OMP end do
35 !$OMP end parallel
36 enddo
```

Machine characteristics

As described in Sect. 2.4.1, the Sandy Bridge core can execute one AVX LOAD, one half AVX STORE, one AVX MULT, and one AVX ADD per cycle. These are the relevant execution units for the Jacobi kernel. The L3 cache size per core is 2.5 MB. One socket has a theoretical memory bandwidth of 51.2GB/s (with DDR3-1600 memory modules).

Microbenchmarking

It is known that the theoretical memory bandwidth cannot be met even under best conditions, so this parameter must be measured with a microbenchmark. The effective STREAM copy bandwidth is $b_S = 40$ GB/s (including the write-allocate transfers) with eight threads .

Building and failure of a naive roofline model

For the first loop, the applicable peak performance can be estimated by assuming a simple throughput limitation on the LOAD port, and we have a performance of 4 LUPs (one AVX update) in six cycles, leading to $P_{\max,1} = 14.4$ GLUP/s on the full socket (eight cores). The bandwidth limitation is at $b_S/B_1 = 625$ MLUP/s. The second loop is limited by the STORE port in L1 cache, because only one half AVX STORE can be sustained per cycle. Hence, we have 4 LUPs in two cycles, or $P_{\max,2} = 43.2$ GLUP/s on the socket. The bandwidth limitation is at $b_S/B_2 = 1670$ MLUP/s.

It follows from this analysis that both loops are strongly memory-bound, and that the first loop should take a fraction of

$$\frac{B_1}{B_1 + B_2} \approx 73\% \tag{5.1}$$

of the overall runtime. This prediction does not coincide with the profiling result of 63%. Although this deviation seems minor, one should not stop here but check the validity of the roofline model.

Both loops together have a code balance of $B_{1+2} = (64 + 24)$bytes/LUP, since both are necessary to perform a complete lattice update. Instead of the expected overall memory-bound performance expectation of $b_S/B_{1+2} = 455$ MLUP/s, however, the measurement is at about 612 MLUP/s, raising more suspicions about the correctness of the model.

Using hardware metrics for validation

In order to explore the deviation from the model one can employ tools that can measure the *actual* memory traffic caused by a code, even on a loop-by-loop basis, by counting hardware events on the chip (see Sect. 5.2.2 below for an example). Using such a tool one discovers that the second loop indeed causes a memory traffic of 24 bytes/LUP, but the first loop needs only 40 bytes/LUP instead of 64. These numbers are perfectly in line with the measured runtime ratio of 63%. Hence, the performance model for the first loop needs to be corrected.

A direct measurement of the memory bandwidth achieved by the running code reveals that it is close to the STREAM maximum, so the deviation from the model is not caused by insufficient utilization of resources.

Update of the roofline model for the first loop

Investigating the first loop in Listing 5.1 more closely, it becomes clear that not all the loads to the source are memory accesses. For instance, `phi(i-1,j,k,t0)` has usually been loaded two `i`-iterations before as `phi(i+1,j,k,t0)`, so we can assume that it is still in cache. This would lead to a code balance of 56 bytes/LUP. If there is enough space in the cache to accommodate at least two successive rows of the lattice (i.e., `phi(:,j:j+1,k,t0)`), the loads to `phi(i+1,j,k,t0)` and `phi(i,j-1,k,t0)` also come from cache, saving another 16 bytes for a balance of 40 bytes/LUP. Finally, if the cache can even hold two successive layers of the lattice (i.e., `phi(:,:,k:k+1,t0)`), only `phi(i,j,k+1,t0)` must be fetched from memory, and the balance goes down to 24 bytes/LUP. Hence, these "layer conditions" determine the actual balance. In terms of the lattice and cache sizes, the two-layer condition for minimum balance (24 bytes/LUP) at t cores is

$$(j_{\max}+2)(i_{\max}+2)\cdot 8\text{bytes}\cdot 2\cdot t \leq C_{\text{eff}}\ , \tag{5.2}$$

where C_{eff} is an effective cache size. As a rule of thumb one can set it to half of the overall cache size, but this depends on the code and how much other data is streamed through the cache hierarchy. The two-row condition for 40 bytes/LUP is

$$(i_{\max}+2)\cdot 8\text{bytes}\cdot 2\cdot t \leq C_{\text{eff}}\ . \tag{5.3}$$

According to the layer conditions it is the extension of the lattice in the i and j directions (but not in k direction) which determines the balance. At the given problem size of 500^3 grid points and a cache size of 20 MB, the condition (5.2) is not fulfilled, since two layers require approximately 32 MB of cache. Hence, we fall back to the row condition (5.3) and expect a balance of 40 bytes/LUP, which is exactly the value measured by hardware counters.

We have now established an agreement between the model and the measurement. Since the memory bandwidth is exhausted, both loops run at the maximum possible performance as given by their code balance. Since both loop nests are required to perform a complete lattice update, one can also determine an overall code balance of $(40+24)$ bytes/LUP $= 64$ bytes/LUP for an expected performance of 625 MLUP/s. The measurement of 612 MLUP/s is in good agreement.

Optimization 1: Common sense

The fact that the performance model describes the actual performance well does not mean that there is no optimization potential. About 37% of the runtime goes into the second loop, which does nothing but copy the updated grid points (`phi(:,:,:,t1)`) back to the source array (`phi(:,:,:,t0)`). There is no real "work" involved, so one may think about this operation as overhead. A simple way to avoid it is to substitute the second loop by a simple construct that exchanges the values of `t0` and `t1` (see line 15 in Listing 5.2). This code has now an overall code balance of 40 bytes/LUP and we can expect a speedup of 60% (since $64/40 = 1.6$) for a performance of 1000 MLUP/s. The measurement of 980 MLUP/s is in good agreement.

Optimization 2: Spatial blocking

At a grid size of 500^3, the layer condition (5.2) cannot be met if the loop nest is executed as given in the code. However, it is completely unimportant in which order the updates are performed

Listing 5.2: Improved implementation of the 3D Jacobi algorithm (adapted from [21]) with the copy loop substituted by a simple swap of time variables (line 15). The relevant LOADs and STOREs for failing the layer condition (5.2) and meeting the row condition (5.3) are also highlighted.

```
1  ! loop over sweeps
2  do s=1,ITER
3    ! sweep over grid
4  !$OMP parallel do schedule(static)
5    do k = 1,kmax
6      do j = 1,jmax
7        do i = 1,imax
8          phi(i,j,k,t1) = oos * ( &
9            phi(i-1,j,k,t0)+phi(i+1,j,k,t0)+phi(i,j-1,k,t0)+ &
10           phi(i,j+1,k,t0)+phi(i,j,k-1,t0)+phi(i,j,k+1,t0) )
11       enddo
12     enddo
13   enddo
14 !$OMP end parallel do
15 i=t0 ; t0=t1; t1=i  ! swap arrays
16 enddo
```

within a time step. In order to establish the layer condition it is thus sufficient to update parts of the grid at a time for which (5.2) is fulfilled. This can be achieved by *spatial blocking* of one or both of the inner loop levels. Since the layer condition does not depend on $k_{\max}$, blocking in this dimension will not help. It turns out that blocking the inner loop leads to performance breakdowns if the loop length is significantly shorter than a page (512 elements) due to TLB and prefetching issues, so we block the middle loop only (see Listing 5.3). Note that inner loop blocking may still be necessary if the inner loop length is very large.

The choice of the block size `bj` may be guided by solving the layer condition for $j_{\max}$: At an (estimated) effective cache size of 10 MB we get $\texttt{bj} \leq 156$. To be on the safe side we choose $\texttt{bj} = 70$ (due to the large number of streams hitting the L3 cache, the effective size is actually just half of the above estimate). This brings the model down to a 24 bytes/LUP balance and to an expected performance of 1670 MLUP/s, of which about 1550 MLUP/s can be measured.

Optimization 3: Non-temporal stores

An important conclusion from the performance model is that exactly one third of the memory bandwidth (8 out of 24 bytes/LUP) is taken up by the write-allocate transfers. These do not even appear as true LOAD instructions in the code but are a simple consequence of the fact that the core can communicate directly only with the L1 cache.[1]

Intel and AMD x86 processors feature special instructions to circumvent the write-allocate when writing to main memory, the *non-temporal stores*. A non-temporal ("NT") store instruction is a normal store, but it writes directly to memory instead of the L1 cache.[2] There are some

[1] In certain situations it is possible for Intel Nehalem and later processors to automatically circumvent the write-allocate between the L2 and the L1 cache [48]. In case of a miss in L2 or L3, the write-allocate will occur.

[2] There is in fact a small number of write-combine buffers caching subsequent non-temporal stores, but for prac-

Listing 5.3: Further improvement of the 3D Jacobi algorithm (adapted from [21]) with spatial blocking in the j direction. The additional outer loop over the blocks, the blocked j loop, and the relevant LOADs and STOREs for meeting the layer condition (5.2) are highlighted.

```
1  ! loop over sweeps
2  do s=1,ITER
3    ! loop nest over blocks
4    do js=1,jmax,bj
5      ! sweep one block
6    !$OMP parallel do schedule(static)
7      do k = 1,kmax
8        do j = js,min(jmax,js+bj-1)
9          do i = 1,imax
10           phi(i,j,k,t1) = oos * ( &
11             phi(i-1,j,k,t0)+phi(i+1,j,k,t0)+phi(i,j-1,k,t0)+ &
12             phi(i,j+1,k,t0)+phi(i,j,k-1,t0)+phi(i,j,k+1,t0) )
13         enddo
14       enddo
15     enddo
16   !$OMP end parallel do
17   enddo
18   i=t0 ; t0=t1; t1=i  ! swap arrays
19 enddo
```

restrictions (for instance, the address to which the data is written must be aligned to a SIMD width address boundary), but in many cases the compiler is able to employ these instructions when allowed or directed to do so. In case of the Jacobi smoother, the `phi(i,j,k,t1)` array in the inner loop is a candidate for non-temporal stores. There is a source code directive which acts as a hint for the Intel compiler to employ NT stores if it is safe, i.e., if the alignment constraint can be met.

Naively one would expect a speedup with respect to standard stores of 50% in this case, since the code balance would be 16 bytes/LUP due to the missing write-allocate. It turns out, however, that the memory interface is less efficient with NT stores, so the input from the microbenchmarking must be modified: With NT stores, the STREAM copy bandwidth goes down from 40 GB/s to 36 GB/s on a Sandy Bridge socket, leading to a performance prediction of 2250 MLUP/s for the Jacobi smoother. The measurement yields 1930 MLUP/s, which is about 86% of the prediction, and 25% faster than the version with standard stores. Hence, this optimization does not quite live up to the expectations.

A closer look at the STREAM benchmarks and the scaling of the smoother performance reveals at least part of the problem: Figure 5.2a shows the bandwidth scaling behavior of the STREAM copy benchmark within a Sandy Bridge chip with all four combinations of standard stores, NT stores, 2.7 GHz clock speed, and "turbo mode" (see Sect. 2.1.7). As expected from the ECM model, the clock speed has significant influence on the bandwidth in the non-saturated case (and also, albeit smaller, in the saturated case, which was already shown in Fig. 4.4). This particular Sandy Bridge processor can run at up to 3.5 GHz when only one or two cores are

tical purposes it can be assumed that the cache is ignored.

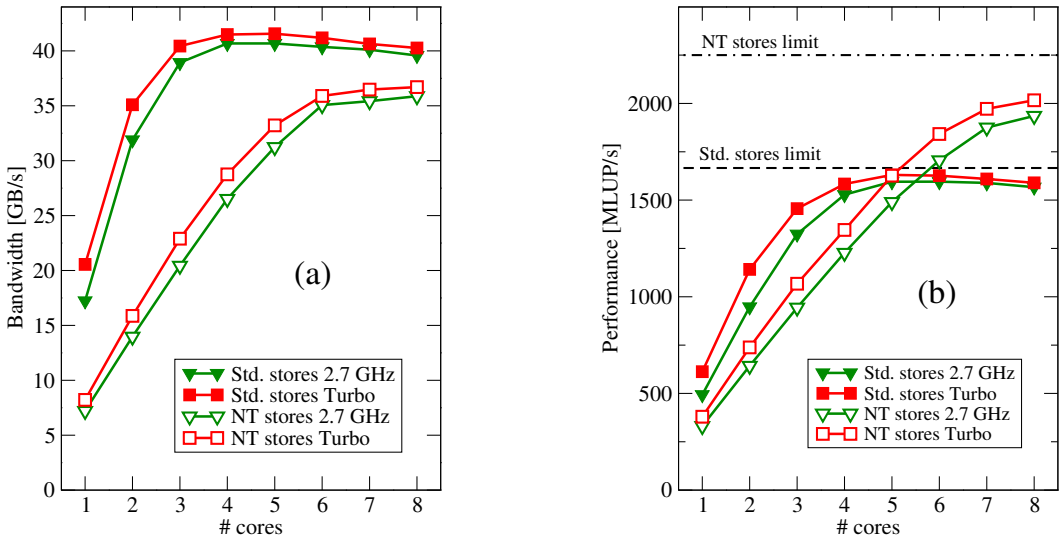


Figure 5.2: (a) Bandwidth scaling of the STREAM copy benchmark across the cores of one Sandy Bridge socket, for a fixed clock speed of 2.7 GHz and turbo mode (triangles vs. squares), with and without non-temporal stores (open vs. filled symbols). The bandwidth number is the actual bandwidth over the memory bus, including write-allocate transfers. (b) Performance scaling of the Jacobi smoother, same parameters.

used, and still at up to 3.1 GHz with a full socket.[3] With NT stores the impact is smaller but still visible. The main conclusion from Fig. 5.2a is that the STREAM benchmark performance saturates across the cores, regardless of whether turbo mode is used or not. The degradation in saturated bandwidth with NT stores is clearly visible, but the effective bandwidth available to the application code is still larger with NT stores. This is only true in the saturated case, however; NT stores show no benefit in the non-saturated regime.

The Jacobi code, on the other hand, shows no clear saturation with NT stores at 2.7 GHz (open triangles in Fig. 5.2b), while saturation is easily reached with standard stores (filled triangles). Although the ECM model is of not much help for NT stores, the general conclusion that fewer cores are needed for saturation at higher clock speeds is still valid. Accordingly, we see a 5% improvement in full-socket performance with NT stores when using turbo mode, for an overall 90% of the prediction. There is still some headroom left, so a yet higher clock frequency or a larger number of cores would result in somewhat better performance.

A summary of the improvements in comparison with the different performance models on a single socket can be seen in Fig. 5.3a.

Optimization 4: Proper ccNUMA placement

A second CPU socket doubles the available memory bandwidth, so we expect a doubling of the performance when running with 16 threads. However, the measurement shows a slowdown to about 1610 MLUP/s. This problem can again be investigated using hardware performance metrics: All the memory traffic in the system goes to the ccNUMA domain at the first socket, and half of those transfers are initiated from the second socket. Hence, there seems to

[3] This data can be obtained with `likwid-powermeter` from the LIKWID tool suite.

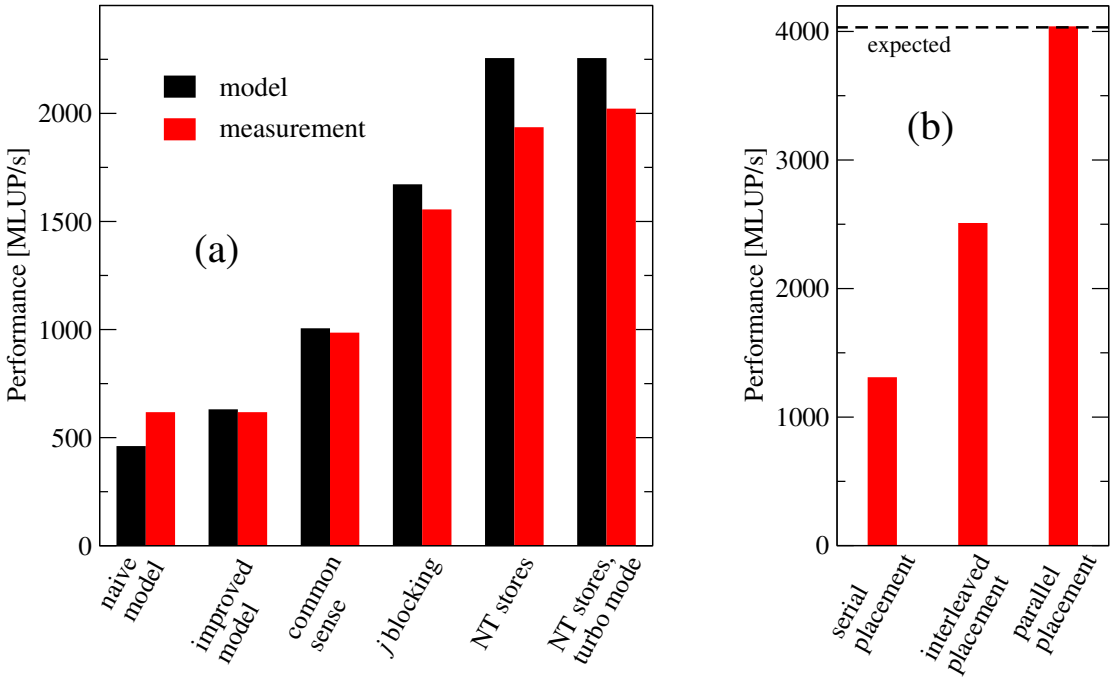


Figure 5.3: Summary of the modeling and optimization results for the performance engineering process applied to the 3D Jacobi smoother on (a) one Sandy Bridge socket and (b) on the whole node (two sockets). Note the change of scale between the two graphs.

be a strong ccNUMA locality problem. Before trying to fix this in the code, one can double-check by running the code with interleaved page placement across the ccNUMA nodes (using `numactl -i 0,1` as a wrapper), which boosts the performance to 3070 MLUP/s. Interleaved page placement ensures at least some parallel data access, although half of all memory requests go to the remote domain. If a loop runs faster with interleaved pages, this is a clear indication of a ccNUMA placement problem.

Fixing the placement problem completely is simple here and involves applying the ”Golden Rule of ccNUMA” [21], a.k.a. the “first-touch principle:” After allocation the arrays must be initialized in parallel, and in the exact same way as they are accessed later in the solver loop. Listing 5.4 shows the correct allocation and initialization code. Note that the middle loop in this nest is blocked with the same j-block size `bj` as the solver loop. This is to ensure that there is absolutely no difference in the data access pattern. The initialization of the boundary layers (for which at least one of the Cartesian indices is zero) must be done after the parallel first touch.

After this change the performance goes up to 4030 MLUP/s on two sockets, i.e., perfect scaling is achieved. A summary of the ccNUMA-related improvements can be seen in Fig. 5.3b.

Jacobi smoother summary

The steps taken above for modeling and optimization of the Jacobi smoother are by no means new. Stencil update schemes have been a subject of intense study over the last decade [49, 50, 51, 52, 53, 54, 9, 55, 10], and there are many more optimizations that can be applied to the serial and parallel code such as temporal blocking or communication hiding (in the distributed-

Listing 5.4: Initialization code for proper ccNUMA page placement. The code for initializing the boundaries on the faces of the domain must come after the parallel first touch, and is omitted.

```
1  double precision, dimension(:,:,:,:) :: phi
2
3  allocate(phi(0:imax+1,0:jmax+1,0:kmax+1,0:1))  ! loop nest over blocks
4  do js=1,jmax,bj
5    ! sweep one block
6  !$OMP parallel do schedule(static)
7    do k = 1,kmax
8      do j = js,min(jmax,js+bj-1)
9        do i = 1,imax
10         phi(:,j,k,:) = ...
11       enddo
12     enddo
13   enddo
14 !$OMP end parallel do
15 enddo
16 ! boundary initialization omitted
```

memory case) [21]. The example was chosen to show the usefulness of a model-guided approach to optimizations. First, runtime profiling was used to get an impression of the importance of different code parts (first vs. second loop). A roofline model was established with input from code analysis, machine characteristics (pipeline throughput), and microbenchmarking (STREAM). Multiple adaptions of the model were necessary, first to get an agreement with the measurements at all (refined roofline model of the first loop), and later to predict the outcome of code optimizations (common sense, spatial blocking, NT stores). Hardware metrics were used to validate or refute models and hypotheses (memory bandwidth saturation, ccNUMA locality problems), and the microbenchmarks were adapted to the requirements of the application code (standard vs. NT stores). Hence, all the components of the performance engineering process sketched in Fig. 5.1 were covered in this simple case study, some of them even multiple times.

5.2 Identification of performance patterns on the node level [5]

While the performance engineering process laid out in the previous section is a useful guideline for practical work with code analysis and optimizations, some details are still missing, especially about how the performance model is built and refined, and how optimizations interact with the model. It turns out that it is very helpful to think about modeling and optimization in terms of *performance patterns*. Patterns aid in building, refining, and validating performance models, and in predicting the outcome of possible optimizations. This section develops the notion of node-level performance patterns.

5.2.1 Hardware performance metrics

Hardware performance monitoring (HPM) is available in every modern microprocessor design. It allows for the measurement of many (sometimes hundreds) of metrics that are related to the way code is executed on the hardware. Although many of those metrics are unimportant for

the developer writing numerical simulation code, some of them can be very useful in assessing resource utilization and general performance properties, and can thus be used to validate performance models. A large variety of tools exist, from basic to advanced, that allow easy access to HPM data, and some of them even give optimization advice derived from the measurements.

Fortunately, although there is considerable variation in the kinds of hardware events that are available on different processors (even from the same manufacturer), a rather small subset of them is sufficient to identify the prevalent performance problems in serial and parallel code. These are available on all modern processor designs. We call a specific combination of hardware event counts and possible other sources of information a "signature." Together with information about runtime performance behavior and code properties, signatures indicate the presence of so-called "performance patterns," which help to assess the quality of code and, most importantly, identify relevant bottlenecks to enable a structured approach to performance optimizations.

5.2.2 likwid-perfctr

Given sufficient experience, simple and lightweight tools are often adequate to accomplish the goals described above. Hence the restriction to x86 architectures under the Linux OS and the use of the `likwid-perfctr` tool from the LIKWID tool suite [25, 26]. LIKWID[4] is a collection of command line programs that facilitate performance-oriented program development and production in x86 multicore environments under Linux. The concept of event sets with connected derived metrics, which is implemented in `likwid-perfctr` by means of performance groups, fits well to the signature approach presented here.

5.2.3 Performance patterns and event signatures

This section describes performance patterns that have been found to be most useful when analyzing scientific application codes on multicore-based nodes. Other application domains may have different issues, but the basic principle could still be applied. The categorization is to some extent arbitrary, and some patterns are frequently found together. Each of those patterns can be mapped to one or more "signatures," which consist of a combination of performance behavior (scalability, sensitivity to problem size, etc.) and a particular pattern in raw or derived hardware metrics. While the former is often independent of the underlying architecture, the latter is very hardware-specific. Ideally a tool should provide these event sets and derived metrics in a similar way on all supported processor architectures. `likwid-perfctr` [25, 26] tries to support this by "performance groups." Table 5.1 maps each performance pattern to its signatures in the performance behavior and to the relevant anomalies in hardware metrics (together with the `likwid-perfctr` performance group, if available). In some cases the signature also involves information from other sources such as microbenchmarks or static code analysis, since some HPM signatures may be easily misinterpreted. Note that general optimization hints are problematic, even if they are based on patterns; optimization is only possible through a thorough code review together with a suitable performance model.

In the following, each pattern is described in detail.

[4] "Like I Knew What I'm Doing"

Pattern		Signature: Performance behavior	Signature: HPM (and `likwid-perfctr` group(s))
Bandwidth saturation		saturating speedup across cores sharing a data path	bandwidth meets BW of suitable streaming microbenchmark (`MEM`, `L3`)
Limited instruction throughput	**Pipeline saturation**	throughput at design limit	low CPI, 1:1 ratio of cy to specific instruction counts (`FLOPS_*`, `DATA`, `CPI`)
	Pipelining hazards	in-core throughput far from design limit, performance insensitive to data size	(large) integral ratio of cy to specific instruction counts, high CPI (`FLOPS_*`, `DATA`, `CPI`)
	Control flow issues		high branch rate, high branch miss ratio (`BRANCH`)
Inefficient data access	**Strided access**	simple BW models far too optimistic	low cache hit ratio, frequent cache line evicts/replacements
	Erratic access		see above, plus low BW utilization (latency) (`CACHE`, `DATA`, `MEM`)
Microarchitectural anomalies		large discrepancy from simple performance models	very hardware-specific, e.g., memory aliasing stalls, conflict misses, unaligned LD/ST, requeue events. Code review required, with architectural features in mind.
False cache line sharing		very low speedup, or slowdown / discrepancy from model only in parallel case	frequent (remote) evicts (`CACHE`)
Bad ccNUMA page placement		bad/no scaling across locality domains, better performance w/ interleaved placement	unbalanced bandwidth on memory interfaces / high remote traffic (`MEM`)
Load imbalance		saturating/sub-linear speedup	different amount of "work" across cores (`FLOPS_*`); instruction count is not reliable!
Synchronization/communication overhead		speedup going down as more cores are added / no speedup with small problem sizes / cores busy but low performance	large non-"work" instruction count (growing with number of cores used) / Low CPI (`FLOPS_*`, `CPI`)
Code composition issues	**Instruction overhead**	low application performance, good scaling across cores, performance insensitive to problem size	low CPI near theoretical limit / large non-FP instruction count (constant vs. number of cores) (`FLOPS_*`, `DATA`, `CPI`)
	Expensive instructions		large CPI
	Ineffective instructions		scalar instructions dominating in data-parallel loops (`FLOPS_*`, `CPI`)

Table 5.1: Performance patterns and corresponding signatures for parallel code on multicore systems. Color code: maximum resource usage (green), hazards (red), work-related patterns (blue).

Bandwidth saturation

Whenever the bandwidth of a shared data path is limited, there is a natural bound to scalability. Most frequently this happens on the main memory interface or the (usually shared) outer-level cache (OLC). Even if an algorithm is perfectly parallelizable in theory (in the sense of Amdahl's Law (3.5)), bandwidth saturation can set a limit to its scalability, as was shown in Fig. 3.9. The roofline model can be used to predict whether the bandwidth bottleneck applies in the saturated case (see Sect. 3.2), while the ECM model describes multicore scaling for streaming kernels (see Sect. 3.4).

Using hardware performance monitoring, bandwidth saturation can be detected by measuring the actual bandwidth utilization of a data path. If the measured bandwidth is close to the maximum value, which can be obtained by a suitable streaming microbenchmark such as STREAM [24], this is an indication that bandwidth limitations play some role.

Limited instruction throughput

There is always a limit for the overall number of instructions that can be executed per cycle on a core, independent of their types. Even if a code does not hit this limit, it could still suffer from a bottleneck in a specific execution port, such as LOAD or MULT (see Fig. 2.1). Depending on whether the pipeline(s) is/are filled or not, one can distinguish three cases: Pipeline saturation, pipelining hazards, and control flow issues.

Pipeline saturation If an execution unit is at its throughput limit, this is indicated by a 1:1 ratio of core cycles and executed instructions on this unit. As a consequence, the CPI value is typically good (low).

Pipelining hazards True data dependencies, i.e., dependencies that cannot be resolved by register renaming, cause pipeline bubbles, which further diminish the throughput. In this case there is often an exact integral ratio (larger than one) between core cycles and executed instructions in the affected pipeline. Due to the additional latencies incurred by the badly pipelined instructions, performance is rather insensitive to the location of the data. The CPI value is typically high, and a simple in-core performance model based on pipeline throughput will be far too optimistic.

Control flow issues This pattern is closely related to pipelining hazards, but the affected pipeline is usually not a single one of the core pipelines but the overall fetch/decode/execute pipeline. The HPM signature is also similar, but there is usually no integer ratio between clock cycles and instructions. Control flow issues arise, e.g., when the branch prediction hardware can not work or when branches depend on the result from previous instructions and the resulting bubbles cannot be filled by other useful work.

In all three cases, HPM or even simple timing measurements can reveal the bottleneck. In general, if code execution performance is limited by instruction throughput on a single pipeline, there is a clear bottleneck on the core level. This simplifies the execution part of ECM modeling, and usually leads to clear indications of what should be done to improve performance. If, e.g., the limiting resource in a loop on an Intel Sandy Bridge core is the LOAD port throughput

on double-precision floating-point data, and all loads in the loop are of scalar type, the core executes two LOADs per cycle. Execution may be sped up by using AVX load instructions instead. This will double the data throughput since now one full-width AVX LOAD (32 bytes) can be performed per cycle. See also the "code composition" pattern below. Control flow issues can sometimes be resolved by reformulating the algorithm so that branches are eliminated from inner loops [21].

Inefficient data access

Cache-based architectures require contiguous data accesses to make efficient use of bandwidth due to the cache line concept. One needs to distinguish strided access from erratic access, since the latter often prevents the efficient use of hardware or software prefetching. In both cases will simple bandwidth-based performance models assuming unit stride be too optimistic.

Strided access Strided data access is often caused by inappropriate data structures or badly ordered loop nests, and is one of the most frequent causes for low data transfer efficiency between cache levels and to/from memory. The cache miss ratio is lower than for unit stride, but can be easily predicted by taking the low cache line utilization into account. A bandwidth-based model can usually be made to work if the prefetching mechanisms can accommodate the strided accesses.

Erratic access If the access pattern is not just strided but erratic (e.g., caused by an indexed array access as in the sparse matrix-vector multiply kernel described in Sect. 3.3.2), automatic or compiler-based prefetch mechanisms may fail, incurring not only loss of effective bandwidth but also exposing memory latency. Bandwidth-based models are hard to reconcile with truly erratic data access, and it is hard or impossible to predict cache miss ratios.

Microarchitectural anomalies

This is a very architecture-specific pattern, which may have different manifestations depending on the hardware. The measured performance will deviate strongly from any model based on "simple" architectural features, such as the ECM model. Typical examples for anomalies are false store forward aliasing, unaligned data accesses or instruction code, a shortage of load/store buffers, cache thrashing due to insufficient cache associativity, bank conflicts in cache or memory, violation of pairing rules,[5] limited reorder buffer size, a limited number of concurrent prefetch streams, etc. Correspondingly, the HPM signature is also very hardware-specific.

False cache line sharing

Different threads accessing a cache line (and at least one of them modifying it) lead to frequent evictions and reloads, impacting performance a lot. False sharing is usually easy to identify using HPM, since frequent remote cache line evicts will occur and speedup will be small or

[5] This effect is specific to the Intel Xeon Phi coprocessor [56]. On this in-order two-way superscalar architecture, pairs of instructions may be scheduled in the same cycle on the same core, but there are limitations on which types can be paired. If the order of instructions in the machine code is not in accordance with the pairing rules, the maximum throughput goes down to one instruction per cycle.

even smaller than one. Once spotted, false sharing is usually simple to remedy by well-known code optimizations such as padding or privatization [21].

Bad page placement on ccNUMA

All modern multi-socket servers are of ccNUMA type. Memory-bound codes must implement proper first-touch page placement in order to profit from the bandwidth advantages that ccNUMA provides. The two main problems with bad page placement are nonlocal data accesses and bandwidth contention, with load imbalance as a possible secondary effect.

Bad ccNUMA page placement is only a problem for memory-bound code, and usually leads to small or no speedup across ccNUMA domains. HPM measurements will report a large volume of non-local traffic across the inter-domain NUMA network, and probably an unbalanced utilization of the memory interfaces.

Load imbalance

Load balancing issues are an impediment for parallel scalability, and hence performance, and they should be resolved first. There are many possible sources of load imbalance, but all include some sort of synchronization or coordination between workers. As an example, a global barrier, i.e., a synchronization point that requires all workers to arrive before any work can proceed, often makes load imbalance manifest. But even without global synchronization, differences in other overhead such as point-to-point communication can lead to imbalanced execution of useful work. This well-known effect has been demonstrated on multicore systems in [2] for the important sparse matrix-vector multiply operation.

Load imbalance does not lead to a drop in performance when more workers are added, unless there are other factors such as communication overhead. Using hardware performance monitoring, this pattern is readily identified by a different amount of "work" performed by different workers. This depends crucially on whether "work" is a well-defined concept in the code; e.g., floating-point operations are usually a good metric in this context, but the number of executed instructions is not, because typical synchronization and communication overhead tends to lead to tight spin-waiting loops, which execute lots of instructions but do not count as "work."

Note that a non-negligible sequential, i.e., non-parallelizable part in an algorithm, and the corresponding limitation of speedup (Amdahl's Law) is only a special case of load imbalance, even if "speedup" is obtained not by using more identical workers but by putting part of the problem onto an accelerator.

Synchronization overhead

Barriers at the end of parallel loops or locks protecting shared resources may have a large performance impact if the workload between such synchronization points is too small. This pattern may also incur secondary effects like load imbalance or code composition issues.

Synchronization overhead typically grows with the number of participating workers, so it is often a fundamental obstacle for strong but also weak scalability [21]. Especially with strong scaling, adding workers inevitably leads to slower execution at some point. In the worst case,

any parallelization, even with only two workers, will slow down the program. HPM measurements typically show a large number of instructions that are not directly associated with application code, and a low CPI value.

There is a broad consent in the supercomputing community that global synchronization (of which global, collective communication is a variant) must be avoided by all means if algorithms are ever going to be exascale-ready [57].

Communication overhead

Communication overhead is usually seen as separate from synchronization overhead, although they are certainly related. Whenever different parts of a system have to communicate in order to work cooperatively, some overhead is to be expected. These parts may even be close together (such as accelerator hardware and the associated host system). Simple point-to-point communication can often be described by a latency-bandwidth model, even in non-trivial cases like halo exchange. An abundance of communication overhead generally manifests itself in a non-linear speedup, especially with strong scaling. The details are very problem-specific, however (see [21] for an overview). The metric signature of communication overhead is similar to synchronization overhead: many non-essential instructions and low CPI.

Code composition issues

Often, the machine code comprises an instruction mix that is inadequate to solve the problem efficiently. A possible symptom of a bad instruction mix is an over-optimistic algorithm-based performance model (which only considers the minimum required amount of work and resources). The difference between prediction and measurement is typically larger when the data set is close to the core, i.e., if it fits in a cache. Consequently, inefficient code execution often manifests itself in an insensitivity of performance to the problem size, just like pipeline hazards and control flow problems. Again we can distinguish several cases:

Instruction overhead General-purpose instruction overhead is caused by inefficient compiler code, which often occurs in over-abstracted C++ frameworks, or with programming languages that are inappropriate for generating efficient low-level code. In this case a code or runtime HPM analysis reveals that the execution bottleneck lies in an abundance of machine instructions that do not do actual "work," such as index arithmetic, inter-register moves, branches, etc. The CPI will typically be low, indicating "good" utilization of the pipelines. Note that synchronization constructs (barriers, locks) and communication overhead (waiting for messages) can have the same effect because the CPU often ends up in tight spin-waiting loops.

Expensive instructions The use of expensive operations like divide and square root can have the opposite effect: The CPI value will be large, since most of the time is spent in long-latency, badly pipelined execution units. This can be observed in the single-core data in Fig. 3.9, where the substitution of the multiply by a divide operation causes a drop in performance of almost a factor of two. Note that modern x86 processors have special instructions for optimized, low-latency, pipelined divides with reduced precision (11 bits instead of 23 in single precision) [23]. These can be used if performance is crucial and accuracy is secondary. See Chapter 6 for an application example.

Ineffective instructions Another source of inefficient low-level code is a low degree of SIMD vectorization with algorithms that are actually (or can be formulated as) data-parallel. The measured CPI value in these cases will be generally low, i.e., many instructions per cycle are executed. If the HPM architecture supports it, scalar and SIMD operations can be counted independently, giving a clear indication of the code composition.

5.2.4 Pattern categorization

The patterns described above help in identifying the relevant bottlenecks of a given loop, and may point to code that is "particularly slow," and could be improved. Slow code has important consequences for scalability. It follows from the ECM model that more cores will be needed to saturate a bandwidth bottleneck if the code runs slower on a single core (see Fig. 3.9b). If there is no chip-level bottleneck (or if it cannot be exhausted), slowing down the code will also impact the large-scale (multi-node) performance; as a side effect, communication and synchronization overhead will be less important and speedup (not performance) will improve. See also the discussion on "slow computing" in [21].

A loose categorization of all identified patterns is shown by the color code in Table 5.1:

- *Maximum resource usage.* Pipeline or bandwidth saturation may be seen as "positive" patterns, since it is not possible to exhaust the respective resource any further. This does not mean, however, that there are no other resources that could be used. For instance, if the ADD pipeline is at its limit but the loop does not execute MULT instructions, fusing loops could lead to a good utilization of both pipelines at the same time.

- *Hazards.* A "hazard" is a condition that leads to sub-optimal utilization of the hardware due to the particular way code is executed or data is accessed. The exact impact of such a pattern is often not directly related to an algorithm but to a specific implementation on specific hardware.

- *Work inefficiency.* The "work" to be done as defined by the algorithm may be executed in a way that prevents efficient use of the hardware. These patterns address general issues and tend to be rather hardware-independent.

It is evident that the "negative patterns" in the last two categories are not clearly separated.

5.3 Patterns and models: Performance engineering refined

Although the patterns described above are useful in the performance engineering work flow, it is still not entirely clear how the interaction of patterns, performance models, and optimizations works in practice. In Fig. 5.1 the details of pattern usage are hidden in the "Performance model" box.

The examples have so far shown that, once a performance model has been built, it can be used for two purposes:

1. If the model "works," i.e., if it can be validated using performance measurements (and, if applicable, HPM data), it describes the relevant bottleneck of the loop correctly. Then it can guide optimizations by predicting the possible benefit of a code change. This prediction can have one of two consequences:

(a) The optimized code is limited by the same bottleneck as the original code (i.e., the same pattern applies).

One example for this was the application of spatial blocking to the 3D Jacobi smoother in Sect. 5.1.2: Before the optimization, *memory bandwidth saturation* was identified as the relevant performance pattern. Blocking the j loop led to a reduction of the code balance from 40 bytes/LUP to 24 bytes/LUP, but the pattern (and thus the bottleneck) stayed the same.

(b) The optimized code hits another bottleneck, which implies a shift to another pattern.

This case was encountered with the divide-accumulate kernel in Sect. 3.3.3: While the original code was core-bound for all but very small problem sizes on the Westmere chip due to the long-latency divides in the loop kernel (*expensive instructions* pattern), the optimized version was strongly memory-bound at large problem sizes (*memory bandwidth saturation* pattern). With a working set fitting in the L1 cache we could potentially expect arithmetic peak for the optimized code (*pipeline saturation* pattern), but we have estimated that the OpenMP overhead will dominate the runtime (*synchronization overhead* pattern).

2. If there is a discrepancy between the performance measurement and the model, the model has "failed." There are two possible reasons for such a mismatch:

(a) The wrong pattern was used for building the model, i.e., the relevant bottleneck was not identified correctly. Fixing this issue either implies a change of pattern, and probably building a new model, or a code optimization that keeps the pattern and makes the performance "fit" to it.

The latter case was observed with the bad (or rather non-existent) scaling across sockets for the 3D Jacobi smoother, where *memory bandwidth saturation* was expected but *bad ccNUMA page placement* was encountered. Fixing this problem by proper parallel first touch initialization shifted the model back to *memory bandwidth saturation*.

The failed roofline modeling of the single-threaded vector triad in main memory (Sect. 3.4.1) led to the development of the ECM model, which can encompass several patterns, depending on the code characteristics. More complex examples for this case will be covered in Chapters 6 and 7.

(b) The pattern was correct, but the input to the model was wrong. Since a model has several inputs (code analysis, microbenchmarks, and machine characteristics), there are several options, such as adjusting the assumed resource requirements of the code, choosing a different microbenchmark, or correcting a probably unjustified assumption about machine characteristics.

In the 3D Jacobi smoother example in Sect. 5.1.2, the first attempt at roofline modeling failed because the amount of data traffic over the memory interface was in fact lower than estimated, although the pattern (*bandwidth saturation*) and even the data path (*memory*) was correct. In this case the code analysis was the problem.

In Sect. 3.3.2 we also studied the sparse matrix-vector multiplication, in which common lore suggests that *erratic access* should be the relevant pattern. The analysis showed that this is not

necessarily the case, and that *memory bandwidth saturation*may also apply. The "severity" of the erratic access pattern could be determined by "reverse modeling," i.e., by measuring certain performance properties of the kernel and then fixing the free parameter α, which describes the overall volume of the data traffic caused by the access to the right-hand side. This example shows that it is not always just a single pattern that applies, but that several patterns may overlap (which is, by design, often the case when the ECM model is used).

Figure 5.4 shows a refined performance engineering cycle, with all activities exposed that employ patterns. The profiling component was omitted for brevity.

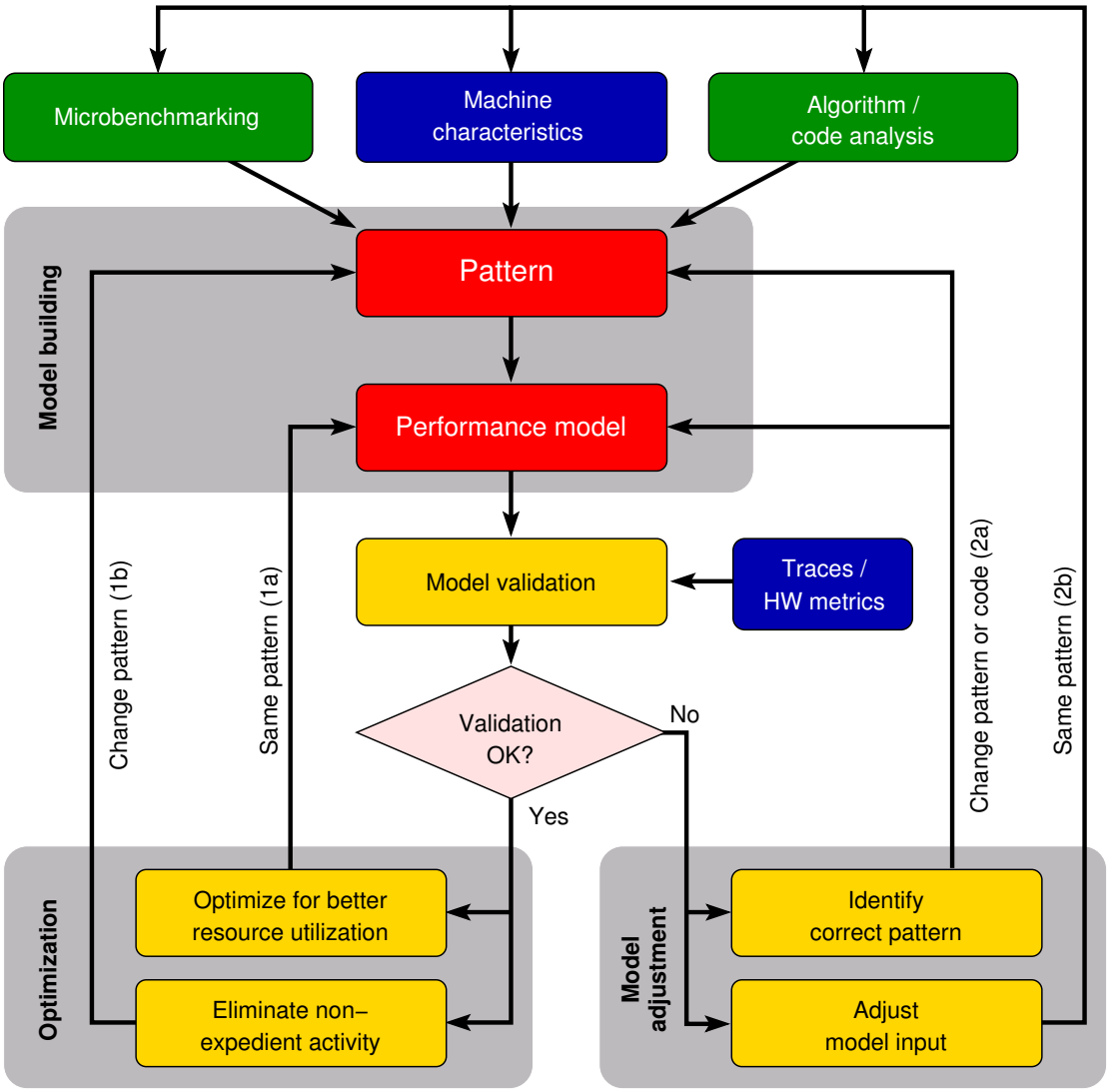


Figure 5.4: A refined performance engineering process, with pattern-related activities exposed. The labels (1a . . . 2b) correspond to the list items in Sect. 5.3.

Part II

Applications

Chapter 6

A medical image reconstruction algorithm [6]

6.1 Introduction

6.1.1 Computed tomography

Computed tomography (CT) [58] is an established technology to non-invasively determine a three-dimensional (3D) structure from a series of projections of an object. Beyond its classic application area of static analysis in clinical environments the use of CT has accelerated substantially in recent years, e.g., toward material science or time-resolved scans supporting interventional cardiology. The numerical volume reconstruction scheme is a key component of modern CT systems and is known to be very compute-intensive. Acceleration through special-purpose hardware such as FPGAs [59] is a typical approach to meet the constraints of real-time processing. Integrating nonstandard hardware into commercial CT systems adds considerable costs both in terms of hardware and software development, as well as system complexity. From an economic view the use of standard x86 processors would thus be preferable. Driven by Moore's law the compute capabilities of standard CPUs have now the potential to meet the requested CT time constraints.

The volume reconstruction step for recent C-arm systems with flat panel detector can be considered a prototype for modern clinical CT systems. Interventional C-arm CTs, such as the one sketched in Fig. 6.1, perform the rotational acquisition of 496 high resolution X-ray projection images (1248×960 pixels) in 20 seconds [60]. This acquisition phase sets a constraint for the maximum reconstruction time to attain real-time reconstruction. In practice filtered backprojection (FBP) methods such as the Feldkamp algorithm [61] are widely used for performance reasons. The algorithm consists of 2D pre-processing steps, backprojection, and 3D post-processing. Volume data is reconstructed in the backprojection step, making it by far the most time-consuming part [59]. It is characterized by high computational intensity, nontrivial data dependencies, and complex numerical evaluations but also offers an inherent embarrassingly parallel structure. In recent years hardware-specific optimization of the Feldkamp algorithm has focused on GPUs [62, 63, 64, 65, 66] and IBM Cell processors [67, 68]. For GPUs in particular, large performance gains compared to CPUs were reported [63] or documented by the standardized RABBITCT benchmark [69, 70]. Available studies with standard CPUs indicate that large servers are required to meet GPU performance [71]. RABBITCT is an open competi-

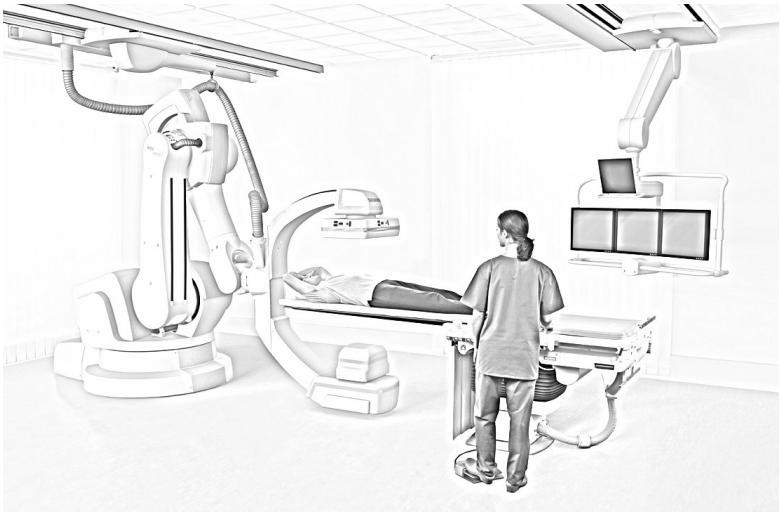


Figure 6.1: C-arm system illustration (Axiom Artis Zeego, Siemens Healthcare, Forchheim, Germany).

tion benchmark based on C-arm CT images of a rabbit (see Fig. 6.2). It allows to compare the manifold of existing hardware technologies and implementation alternatives for reconstruction scenarios by applying them to a fixed, well-defined problem. This chapter concentrates on the optimal implementation of the FBP algorithm on multicore processors. See [6] for a detailed account of related work.

This chapter highlights, in condensed form, the aspects of Ref. [6] that are related to the performance engineering approach: Starting from a first rough analysis of the code, which points to a strongly memory-bound problem on modern multicore chips, obvious optimizations such as work reduction and SIMD vectorization are applied. Using the optimized code as a baseline, an ECM performance model is built which leads to the conclusion that the algorithm is memory-bound only on older, bandwidth-starved processors, but not on modern CPUs like the Intel Sandy Bridge. Consequently, blocking or unrolling techniques only pay off when bandwidth limitation applies. Performance results are presented for a number of modern and older Intel multicore CPUs.

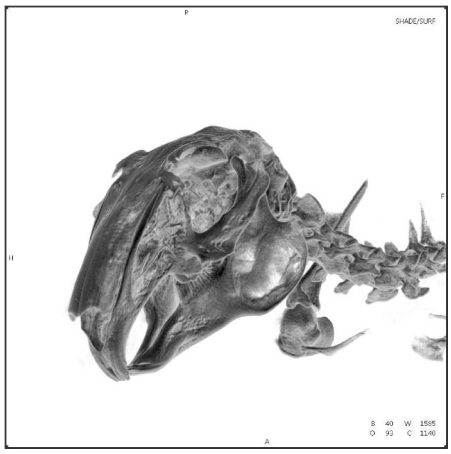


Figure 6.2: Volume rendering based on the reconstruction of 2D X-ray projections of a rabbit.

6.2 Experimental testbed

A selection of modern Intel x86-based multicore processors (see Table 6.1) was chosen to test the performance potential of optimized implementations of the algorithm. All of these chips feature a large outer level cache, which is shared by two (Core 2 Quad "Harpertown"), four (Sandy Bridge), six (Westmere EP), or ten cores (Westmere EX). The maximum number of cores sharing an outer level L2/L3 cache is called an "L2/L3 group."

With the introduction of the Core i7 architecture the memory subsystem of Intel processors was redesigned to allow for a substantial increase in memory bandwidth, at the price of introducing ccNUMA on multisocket servers. At the same time Intel also relaunched simultaneous multithreading (SMT) with two threads per physical core. The Sandy Bridge processor is equipped with a new instruction scheduler, supports the new AVX SIMD instruction set extension, and has a new last level cache subsystem (which was already present in Nehalem EX). The 10-core Intel Westmere EX is not mainly targeted at HPC clusters but reflects the performance maximum for x86 shared-memory nodes. A summary of the most important processor features is presented in Table 6.1. Note that the Sandy Bridge model used here is a desktop variant, while the other processors are of the server ("Xeon") type. Table 6.1 also contains bandwidth measurements for a simple update benchmark:

```
1 for(int i=0; i<N; ++i)
2   a[i] = s * a[i];
```

This benchmark reflects the data streaming properties of the reconstruction algorithm and is thus better suited than STREAM [72] as a baseline for a quantitative performance model.

Since most of the performance-critical code was written in assembly language, the choice of compiler is marginal (the Intel compiler in version 12.0 was used). Thread affinity, hardware performance monitoring, and low-level benchmarking was implemented via the LIKWID tool suite [73, 74], using the tools `likwid-pin`, `likwid-perfctr`, and `likwid-bench`, respectively.

6.3 The algorithm

6.3.1 Theory

The RABBITCT dataset consists of $N = 496$ projection images I_n acquired by a C-arm system. The projections are already pre-processed and filtered. Hence, only the backprojection step is considered in the presented work. Each projection image is accompanied by a projection matrix $A_n \in \mathbb{R}^{3\times4}$ [75, 76]. It encodes the complete projection geometry, including reproducible deviations from the ideal Feldkamp geometry [76]. Using A_n, the perspective projection of an arbitrary point $\vec{x} = (x,y,z)^T$ in 3D space onto the point $\vec{p}$ in the u-v image plane of the n-th view can be expressed as [77]

$$\tilde{\vec{p}}_n \cong A_n \tilde{\vec{x}}, \tag{6.1}$$

Table 6.1: Test machine specifications. The cacheline size is 64 bytes for all processors and cache levels. The update benchmark results were obtained with the likwid-bench tool.

Microarchitecture	Intel Harpertown	Intel Westmere	Intel Westmere EX	Intel Sandy Bridge
Model	Xeon X5482	Xeon X5670	Xeon E7- 4870	Core i7-2600
Label	HPT	WEM	WEX	SNB
Clock [GHz]	3.2	2.66 (2.93 turbo)	2.40	3.4 (3.5 turbo)
sockets/cores/threads	2/8/-	2/12/24	4/40/80	1/4/8
SIMD extension	SSE3	SSE4.2	SSE4.2	AVX
SIMD register [bytes]	16	16	16	32
Socket L1/L2/L3	4×32k/2×6M/-	6×32k/6×256k/12M	8×32k/8×256k/30M	4×32k/4×256k/8M
Bandwidths [GB/s]:				
Theoretical socket BW	12.8	32.0	34.2	21.3
Update (1 thread)	5.9	15.2	8.3	16.5
Update (socket)	6.2	20.3	24.6	17.3
Update (node)	8.4	39.1	98.7	-

where $\tilde{\vec{x}} = (\vec{x}^T, 1)^T$ and $\tilde{\vec{p}} = (u, v, 1)^T w$. Note that the equality is in homogeneous coordinates and therefore up to scale. For convenience, we further define

$$u_n(\vec{x}) = \tilde{\vec{p}}_{n,0}/w_n(\vec{x}), \tag{6.2}$$
$$v_n(\vec{x}) = \tilde{\vec{p}}_{n,1}/w_n(\vec{x}), \text{and} \tag{6.3}$$
$$w_n(\vec{x}) = \tilde{\vec{p}}_{n,2}. \tag{6.4}$$

The N filtered projection images I_n are backprojected into the volume F. The value of a voxel at position $\vec{x} = (x, y, z)$ is determined as

$$F(\vec{x}) = \sum_{n=1}^{N} \frac{1}{w_n(\vec{x})^2} \cdot I_n(u_n(\vec{x}), v_n(\vec{x})). \tag{6.5}$$

Since equation (6.1) is only defined up to scale, A_n can be constructed such that the scaling factor $\tilde{\vec{p}}_{n,2}$ corresponds to the distance weight w in the backprojection formula (6.5) [76].

Note that in practice one deals with image data that has a finite pixel resolution. The projection of a voxel will in general not hit one pixel of the 2D CT image exactly. Therefore, the projection value is computed by bilinear interpolation of the four closest pixels.

6.3.2 Code analysis

The basic backprojection algorithm (as provided by the RabbitCT framework [70], see Listing 6.1) is usually implemented in single precision (SP) and exhibits a streaming access pattern for most of its data traffic. One volume reconstruction uses 496 CT images (denoted by `I`) of 1248×960 pixels each (`ISX`×`ISY`). The volume size is 256^3 mm^3. `MM` is the voxel size and changes depending on the number of voxels. The most common resolution in present clinical

Listing 6.1: Voxel update loop nest for the plain backprojection algorithm. This gets executed for each projection `I`. All variables are of type `float` unless indicated otherwise. The division into parts (see text) is only approximate since there is no 1:1 correspondence to the SIMD-vectorized code.

```
1  wz = offset_z;
2  for(int z=0; z<L; z++, wz+=MM) {
3   wy = offset_y;
4
5   for (int y=0; y<L; y++, wy+=MM) {
6     wx = offset_x;
7     valtl=0.0f; valtr=0.0f;
8     valbl=0.0f; valbr=0.0f;
9
10  // Part 1 ---------------------------
11    for (int x=0; x<L; x++, wx+=MM) {
12      uw = (A[0]*wx+A[3]*wy+A[6]*wz+A[9]);
13      vw = (A[1]*wx+A[4]*wy+A[7]*wz+A[10]);
14      w  = (A[2]*wx+A[5]*wy+A[8]*wz+A[11]);
15
16      u = uw * 1.0f/w; v = vw * 1.0f/w;
17
18      int iu = (int)u, iv = (int)v;
19
20      scalu = u - (float) iu;
21      scalv = v - (float) iv;
22  // Part 2 ---------------------------
23      if (iv>=0 && iv<ISY) {
24        if (iu>=0 && iu<ISX)
25          valtl = I[iv*ISX + iu];
26        if (iu>=-1 && iu<ISX-1)
27          valtr = I[iv*ISX + iu+1];
28      }
29
30      if (iv>=-1 && iv<ISY-1) {
31        if (iu>=0 && iu<ISX)
32          valbl = I[(iv+1)*ISX + iu];
33        if (iu>=-1 && iu<ISX-1)
34          valbr = I[(iv+1)*ISX + iu+1];
35      }
36  // Part 3 ---------------------------
37      vall = scalv*valbl + (1.0f-scalv)*valtl;
38      valr = scalv*valbr + (1.0f-scalv)*valtr;
39      fx   = scalu*valr  + (1.0f-scalu)*vall;
40
41      VOL[z*L*L + y*L + x] += 1.0f/(w*w)*fx;
42    } // x
43  } // y
44 } // z
```

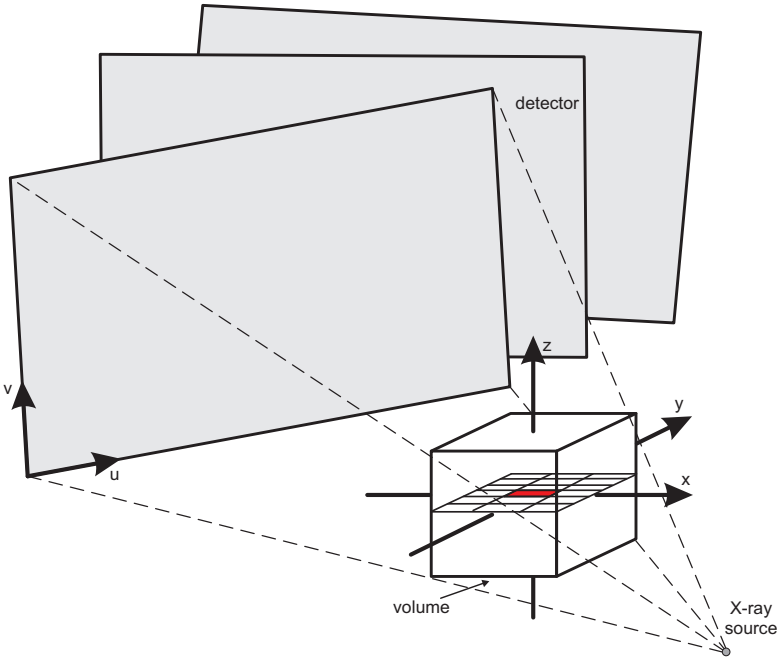


Figure 6.3: Setup geometry for generating the CT projection images. The size of the volume is always $256^3\,\mathrm{mm}^3$, but the number of voxels may vary. The *x*-*y*-*z* space represents the volume while the *u*-*v* plane represents a filtered projection.

applications is 512 voxels in each direction (denoted by the problem size `L`). The algorithm computes the contributions to each voxel across all projection images, and the reconstructed volume is stored in array `VOL`. Voxel coordinates (indices) are denoted by `x`, `y`, and `z`, while pixel coordinates are called `u` and `v`. See Fig. 6.3 for the geometric setup.

The aggregate size of all projection images is about 2.4 GB. One voxel sweep incurs a data transfer volume consisting of the loads from the projection image and an update operation (`VOL[i]+=s`, see line 41 in Listing 6.1) to the voxel array. The latter causes 8 bytes of traffic per voxel and results (for problem size 512^3) in a data volume of 1 GB, or 496 GB for all projections. The traffic caused by the projection images is not easy to quantify since it is not a simple stream; it is defined by a "beam" of locations slowly moving over the projection pixels as the voxel update loop nest progresses. It exhibits some temporal and spatial locality since neighboring voxels are projected on proximate pixels of the image, but there may also be multiple streams with large strides. Nevertheless, the above estimates suggest that the memory traffic caused by the projection images is small compared to the updates to the voxel volume. On the computational side, the basic version of this algorithm performs 13 additions, 5 subtractions, 17 multiplications, and 3 divides.

6.3.3 Simple performance models

Based on this knowledge about data transfers and arithmetic operations one can build a roofline model for a rough upper performance bound on the compute node. The arithmetic limitation results in 20 cycles per vectorized update (four and eight inner loop iterations for SSE and AVX, respectively), assuming full vectorization, and a throughput of one divide per cycle. This takes into account that all architectures under consideration can execute one addition and one multiplication per cycle and assumes that the pipelined `rcpps` instruction can be employed for the divisions (see Sect. 6.4.1 for details) and shares an execution port with the multiply

instructions. Knowing the number of cycles c per vectorized update, the most optimistic in-core performance (in voxel updates per time unit) is

$$P_{\max} = \frac{f \cdot s \cdot n}{c} \,, \tag{6.6}$$

where f is the clock frequency, s is the SIMD width, and c is the number of cores per node.

The performance limitation due to data transfers is given by $I \cdot b_S$, where I is the computational intensity of one update per eight bytes (see above), and b_S is the node memory bandwidth as measured with the synthetic update benchmark described in Sect. 6.2 (see Table 6.1). The following table shows upper performance bounds for a full reconstruction based on in-core and bandwidth limitations on the four systems in the testbed (full nodes; see Table 6.1 for label definitions):

	HPT	WEM	WEX	SNB
$P_{\max}$ [GUP/s]	5.12	7.03	19.2	5.60
$I \cdot b_S$ [GUP/s]	1.05	4.89	11.2	2.16

Performance is given in billions of voxel updates per second (GUP/s),[1] where one "update" represents the reconstruction step of one voxel using a single image. The large expected performance for the single socket (quad-core) Sandy Bridge under the in-core arithmetic limitation is caused by its wide AVX vector size and its fast clock speed.

Above predictions indicate a strongly memory-bound situation, but it will be shown later that they are far from accurate: It is much too optimistic to assume perfectly independent instructions and perfect SIMD vectorization. Moreover, counting only "useful" work, i.e., arithmetic operations, is wrong since this algorithm is nontrivial to vectorize due to the scattered load of the projection image data; it therefore involves many more non-arithmetic instructions. A more careful analysis will lead to a completely different picture, and further optimizations can change the bottleneck analysis considerably.

In order to have a better view on low-level optimizations we divide the algorithm into three parts:

1. Geometry computation: Calculate the index of the projection of a voxel in pixel coordinates
2. Load four corner pixel values from the projection image
3. Interpolate linearly for the update of the voxel data

6.3.4 Algorithmic optimizations

The first optimizations for a given algorithm must be on a hardware-independent level. Beyond elementary steps like moving invariant computations out of the inner loop body and reducing the divides to one reciprocal (thereby reducing the flop count to 31), a main optimization is to minimize the workload. Voxels located at the corners and edges of the volume are not visible on every projection, and can thus be "clipped off" and skipped in the inner loop. This is not a

[1] SI prefixes are used, i.e., 1 GUP/s stands for 10^9 updates per second. This is inconsistent with a large part of the literature on medical image reconstruction, where "G" is used as a binary prefix for $2^{30} \approx 1.074 \cdot 10^9$ [78]

new idea, but the approach presented here improves the work reduction from 24% [79] to nearly 39%.

The basic building block for all further steps is the update of a consecutive line of voxels in x direction, covered by the inner loop level in Listing 6.1. This is called the "line update kernel." The geometry, i.e., the position of the first and the last relevant voxel for each projection image and line of voxels is precomputed. This information is specific for a given geometric setup, so it can be stored and used later during the backprojection loop. Reading this extra data from memory incurs an additional transfer volume of $512^2 \times 496 \times 4$ bytes=496 MB (assuming 16-bit indexing), which is negligible compared to the other traffic. The advantage of line-wise clipping is that the shape of the clipped voxel volume is much more accurately tracked than with the blocking approach described in [79].

The conditionals (lines 23 and 30 in Listing 6.1), which ensure correct access to the projection image, involve no measurable overhead for the scalar case due to the hardware branch prediction. However, for vectorized code they are potentially costly since an appropriate mask must be constructed whenever there is the possibility that a SIMD vector instruction accesses data outside the projection [79]. To remove this complication, separate buffers are used to hold suitably zero-padded copies of the projection images, so that there is no need for vector masks. The additional overhead is far outweighed by the performance advantage for fully vectorized code execution. The conditionals are also effectively removed by the clipping optimization described above, but we need a code version without clipping for validating our performance model later.

Note that a similar effect could be achieved by peeling off scalar loop iterations to make the length of the inner loop body a multiple of the SIMD vector size and ensure aligned memory access. However, this may introduce a significant scalar component especially for small problem sizes and large vector lengths.

6.4 Single core optimizations

For all further optimizations an implementation of the line update kernel in C is chosen as the performance baseline, with all algorithmic optimizations from Sect. 6.3.4 already applied.

6.4.1 SIMD vectorization

No current compiler is able to efficiently vectorize the backprojection algorithm, so the code was implemented directly in x86 assembly language. Using SIMD intrinsics could ease the vectorization but adds some uncertainties with regard to register scheduling and hence does not allow full control over the instruction code. All data is aligned to enable packed and aligned loads/stores of vector registers (16 or 32 bytes with one instruction).

The line update kernel operates on consecutive voxels. For part 1 of the algorithm classic vectorization, i.e., working on multiple voxels at the same time, is straightforward (see Sect. 6.3.3). This part is arithmetically limited and fully benefits from the increased register width. The divide is replaced by a reciprocal. SSE provides the fully pipelined `rcpps` instruction for an approximate reciprocal with reduced accuracy (11 bits) compared to a full divide (24 bits). This approximation is sufficient for this algorithm, and results in an accuracy similar to GPGPU implementations. The integer type cast (line 18) is implemented via the vectorized hardware rounding instruction `roundps`, which was introduced with SSE4.

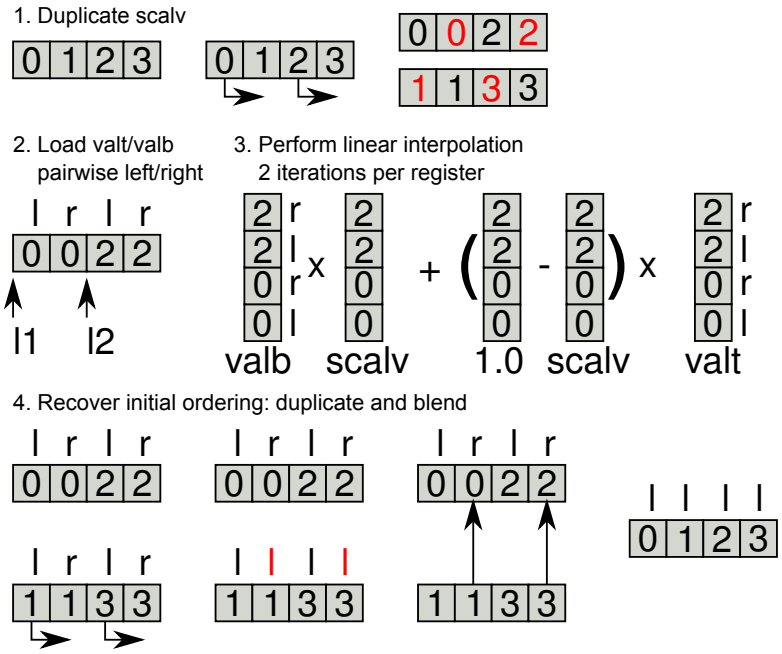


Figure 6.4: Vectorization of part 2 of the algorithm: The data is loaded pairwise into the vector registers. The interpolation of iterations 0,2 and 1,3 are computed simultaneously. Afterwards the results must be reordered for the second interpolation step.

Part 2 of the algorithm cannot be directly vectorized. The projection value is computed by bilinear interpolation of the four closest pixels (top left (`valtl`), top right (`valtr`), and bottom left (`valbl`), bottom right (`valbr`)). As illustrated in Fig. 6.5, `valtl` and `valtr` as well as `valbl` and `valbr` can be loaded in pairs. Moreover, the classic vectorization approach of part 1 – operating on multiple voxels at the same time – cannot be retained here since neighboring voxels will in general not be projected onto consecutive pixels. Fig. 6.4 shows the steps involved in vectorizing part 2 and the first linear interpolation in more detail. For the sake of simplicity, we consider a vector of four voxels with indices 0–3, but the scheme can be easily extended to wider vectors. Since the pixel coordinates from step 1 are already in a vector register, the index calculation for, e.g., `iv*ISX+iu` and `(iv+1)*ISX+iu` (lines 25, 27, 32, and 34 in Listing 6.1) uses packed SIMD instructions. We compute the first part of the interpolation for voxels 0 and 2 simultaneously, and then for voxels 1 and 3. Therefore, we duplicate the

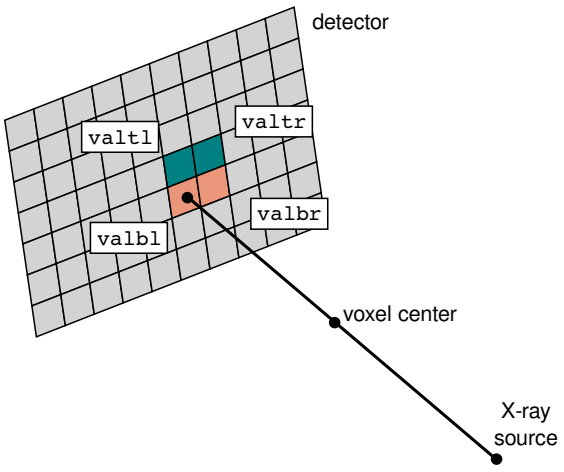


Figure 6.5: Projection of a voxel center onto the detector. The labeled (four) pixels are used for bilinear interpolation.

weighting vector `scalv` such that one copy contains the weights for voxels 0 and 2 (twice), and the other one for voxels 1 and 3 (step 1 in Fig. 6.4). With one load at the index of `valtl` we implicitly load `valtr` into the second vector element. Again we create two vectors, one containing the top left and right pixel values for voxels 0 and 2, and one for 1 and 3 (step 2 in Fig. 6.4). The same is done for the bottom values. The resulting vectors are called `valt` and `valb`, respectively, in Fig. 6.4. Note that the cost for this construct increases with wider SIMD registers because two load operations are required per voxel. Next, we multiply those vectors with the corresponding `scalv` vector in step 3 of Fig. 6.4 and get vectors with the interpolation result in v direction. The elements are still alternating left and right, and for voxels 0 and 2, or 1 and 3. Therefore, in step 4, we reorder the elements to get a vector containing only the left values for all voxels and one containing the right values. This scheme is implemented using the SSE3 instructions `movsldup` and `movshdup` for duplication of the `scalv` values. The final reordering to enable classic vectorization in part 3 of the algorithm also uses those instructions, together with `blendps`, which interleaves the values to bring them into the correct order again. The conversion of the index into a general purpose register, which is needed for addressing the load of the data and the scattered pairwise loads, is costly in terms of necessary instructions. Moreover, the runtime increases linearly with the width of the registers due to the pairwise loads. This implies that the whole operation is limited by instruction throughput.

We consider two SSE implementations, which only differ in part 2 of the algorithm. Version 1 (V1) converts the floating point values in the vector registers to four quadwords and stores the result back to memory (cache, actually). Single index values are then loaded to general purpose registers one by one. Version 2 (V2) does not store to memory but instead shifts all values in turn to the lowest position in the SSE register, from where they are moved directly to a general purpose register using the `cvtss2si` instruction.

The remainder of part 3 – the second part (u direction) of the bilinear interpolation, and the voxel update – is again trivially vectorizable and fully benefits from wider SIMD registers.

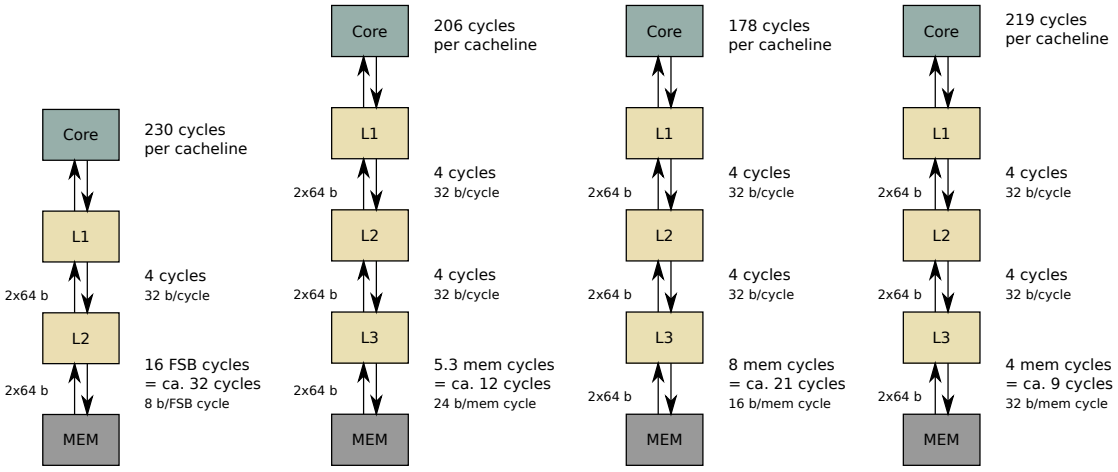
Note that any further inner loop unrolling beyond what is required by SIMD vectorization would not show any benefit due to register shortage; however, as will be shown later, SMT can be used to achieve a similar effect.

6.4.2 AVX implementation

In theory, the AVX instruction set extension doubles the peak performance per core as compared to SSE. The backprojection cannot fully benefit from this advantage because the number of required instructions increases linearly with the register width in part 2 of the algorithm. For arbitrary SIMD vector lengths a hardware gather operation would be required to prevent this part from becoming a severe bottleneck.[2]

The limited number of AVX instructions that natively operate on 256-bit registers prohibits more sophisticated variants of part 2; only the simple version (V1) could be implemented. A register-to-register variant would be possible only at the price of a much larger instruction count, so this alternative was not considered. Despite these shortcomings, an improvement of 25% could be achieved with the AVX kernel on Sandy Bridge (see Sect. 6.7 for detailed performance results).

[2]The Intel Xeon Phi coprocessor and the Intel Haswell processor do have such instructions. See [80] for a detailed analysis of the backprojection algorithm on the Intel Xeon Phi.



(a) Harpertown (b) Westmere (c) Sandy Bridge (SSE) (d) Westmere EX

Figure 6.6: ECM model analysis: Runtime contributions from instruction execution and necessary cache line transfers. The total data volume in bytes is indicated on the left of each group of arrows. On the right we show the data transfer capabilities between hierarchy levels and the resulting transfer time in core cycles. In-core execution times are measured values from Table 6.2, scaled to a complete cache line.

6.5 In-depth performance analysis

6.5.1 ECM performance model

As shown in Sect. 6.3.3, a simple roofline model analysis based on arithmetic instruction throughput and the memory bottleneck alone can be done for the backprojection algorithm. It predicts a strong bandwidth limitation on all considered architectures, but a simple measurement of the utilized memory bandwidth proves that the memory bandwidth is far from saturated in most cases. Hence, the ECM model introduced in Sect. 3.4 is employed to arrive at a more complete picture.

The starting point for all further analysis is the single-thread runtime spent executing instructions with data loaded from L1 cache, since this is what determines $P_{\max}$. Due to the complexity of the loop body, the Intel Architecture Code Analyzer (IACA) [38] was used to analytically determine the runtime of the loop body. This tool determines the runtime of the loop body in cycles, calculating either raw throughput (no dependencies) or the critical path length, according to the architectural properties of the processor under the assumption that all data resides in the L1 cache. It supports Westmere and Sandy Bridge (including AVX) as target architectures. The results for Westmere are shown in the following table for the two SSE kernel variants described above (all entries except μops are in core cycles):

	Issue port								
Variant	0	1	2	3	4	5	TP	μOPs	CP
V1	15	21	**24**	3	3	19	**24**	85	54
V2	20	**27**	16	1	1	20	**27**	85	71

Execution times are calculated separately for all six issue ports (0...5). (A μop is a RISC-like

"micro-instruction;" x86 processors perform an on-the-fly translation of machine instructions to μops, which are the "real" instructions that get executed by the core.) Apart from the raw throughput (TP) and the total number of μops the tool also reports a runtime prediction taking into account latencies on the critical path (CP). Based on this prediction V1 should be faster than V2 on Westmere. However, the measurements in Table 6.2 show the opposite result. The high pressure on the load issue port (2) together with an overall high pressure on all ALU issue ports (0, 1, and 5) seems to be decisive. In V2 the pressure on port 2 is much lower, although the overall pressure on all issue ports is slightly larger.

Below we report the results for the Sandy Bridge architecture with SSE and AVX. The pressure on the ALU ports is similar, but due to the doubled SSE load performance Sandy Bridge needs only half the cycles for the loads in kernel V1. V1 is therefore faster than V2 on Sandy Bridge (see Table 6.2).

	Issue port								
Variant	0	1	2	3	4	5	TP	μOPs	CP
V1 SSE	16	**20**	14	13	3	19	**20**	85	56
V2 SSE	20	**26**	9	8	1	21	**26**	85	72
V1 AVX	18	20	22	21	6	**30**	**30**	114	90

So far we have assumed that all data resides in the L1 cache. The data transfers required to bring cachelines into L1 and back to memory are modeled separately. We assume that there is no overlap between data transfers and instruction execution. This is true at least for the L1 cache: It can either communicate with L2 to load or evict a cacheline, or it can deliver data to the registers, but not both at the same time. As a first approximation we also (pessimistically) assume that this "no-overlapping" condition holds for all caches, and that a data transfer between any two adjacent levels in the memory hierarchy does not overlap with anything else. Since the smallest transfer unit is a 64-byte cacheline, the analysis will from now on be based on a full "cacheline update" (16 four-byte voxels), which corresponds to four (two) inner loop iterations when using SSE (AVX).

We only consider the data traffic for voxel updates; the image data traffic is negligible in comparison, hence we assume that all image data comes from L1 cache. It takes two cycles to transfer one cacheline between adjacent cache levels over the 256-bit unidirectional data path. Every modified line must eventually be evicted, which takes another two cycles. Figure 6.6 shows a full analysis, in which the core execution time for a complete cacheline update is based on the measured cycles from Table 6.2. On the three architectures with L3 cache the simplification is made that the "Uncore" part (L3 cache, memory interface, and QuickPath interconnect) runs at the same frequency as the core, which is not strictly true but does not change the results significantly. It was shown for the Nehalem-based architectures (Westmere and Westmere

	HPT	WEM	WEX	SNB
V1 SSE	62.6	61.6	59.6	44.4
V2 SSE	57.4	51.5	54.7	50.0
V1 AVX				76.2

Table 6.2: Measured execution times (one core) in cycles for one iteration of the SIMD-vectorized kernel (i.e., 4 or 8 voxel updates) with all operands residing in L1 cache.

EX) that they can overlap instruction execution with reloading data from memory to the last level cache [81]. Hence, the model predicts that the in-core execution time is much larger than all other contributions, which makes this algorithm limited by instruction throughput for single core execution. On Sandy Bridge, the AVX kernel requires 76.2 cycles for one vectorized loop iteration (eight updates). This results in 152 cycles instead of 178 cycles (SSE) for one cacheline update.

Based on the runtime of the loop kernel we can now estimate the total required memory bandwidth for multithreaded execution if all cores on a socket are utilized, and also derive the expected performance (we consider the full volume without clipping):

	HPT	WEM	WEX	SNB	SNB (AVX)
BW/core [GB/s]	1.7	1.9	1.5	2.5	3.0
BW/socket [GB/s]	**6.8**	11.2	11.6	10.0	12.0
Perf. [GUP/s]	0.85	1.42	1.45	1.25	1.51

We conclude that the multithreaded code is bandwidth-limited only on Harpertown, since the required socket bandwidth is above the practical limit given by the update benchmark (see Table 6.1). All other architectures are below their data transfer capabilities for this operation and should show no benefit from further bandwidth-reducing optimizations (see Sect. 6.6.2).

6.5.2 ILP optimization and SMT

At this point the analysis still neglects the possible benefit from Simultaneous multi-threading. As described in Sect. 2.1.5, SMT can improve the pipeline utilization for codes that suffer from dependencies, long-latency loads, instruction scheduling issues, or resource contention. At the same time it is important to understand that there can be no benefit if all threads running on the same core compete for a shared resource like, e.g., a request queue.

As shown in the previous section, our implementation of the backprojection algorithm exhibits a strong discrepancy between the IACA "throughput" and "critical path" predictions. Due to the complex loop body, register dependencies are unavoidable, resulting in many pipeline bubbles. Outer loop unroll and jam (interleaving two outer loop iterations in the inner body) is out of the question due to register shortage, but SMT can do a similar job and provide independent instruction streams using independent register sets. Since there is no shared resource apart from the core pipelines, running two threads on the two virtual cores of each physical core is expected to reduce the cycles taken for the cacheline update. However, the effect of using SMT is difficult to estimate quantitatively. See Sect. 6.7 below for complete parallel results.

6.6 OpenMP parallelization

OpenMP parallelization of the algorithm is straightforward and works with all optimizations discussed so far. For the thread counts and problem sizes under consideration here it is sufficient to parallelize the loop that iterates over all voxel volume slices (loop variable `z` in Listing 6.1). However, due to the clipped-off voxels at the edges and corners of the volume, simple static loop scheduling with default chunk size leads to a strong load imbalance. This can be easily corrected by using block-cyclic scheduling with a small chunk size (e.g., `static,1`).

Images are produced one by one during the C-arm rotation, and could at best be delivered to the application in batches. Since the reconstruction should start as soon as images become available, a parallelization across images was not considered.

As shown in Sect. 6.5, the socket-level performance analysis does not predict strong benefits from bandwidth-reducing optimizations except on the Harpertown platform. However, since one can expect to see more bandwidth-starved processor designs with a more unbalanced ratio of peak performance to memory bandwidth in the future, we still consider bandwidth optimizations important for this algorithm. Furthermore, ccNUMA architectures have become omnipresent even in the commodity market, making locality and bandwidth awareness mandatory. In the following sections we will describe a proper ccNUMA page placement strategy for voxel and image data, and a blocking optimization for bandwidth reduction. The reason why we present those optimizations in the context of shared-memory parallelization is that they become relevant only in the parallel case, since bandwidth is not a problem on all architectures for serial execution (see Sect. 6.5.1).

6.6.1 ccNUMA placement

The reconstruction algorithm uses essentially two relevant data structures: the voxel array and the image data arrays. Upon voxel initialization one can easily employ first-touch initialization, using the same OpenMP loop schedule (i.e., access pattern) as in the main program loop. This way each thread has local access (i.e., within its own ccNUMA domain) to its assigned voxel layers, and the full aggregate bandwidth of a ccNUMA node can be utilized.

Although the access to the projection image data is much less bandwidth-intensive than the memory traffic incurred by the voxel updates, ccNUMA page placement was implemented here as well. As mentioned in Sect. 6.3.4, the padded projection buffers are explicitly allocated and initialized in each locality domain, and a local copy is shared by all threads within a domain. Since the additional overhead for the duplication is negligible, this ensures conflict-free local access to all image data. The time taken to copy the images to the local buffers is included in the runtime measurements.

6.6.2 Blocking/unrolling

In order to reduce the pressure on the memory interface we use a simple blocking scheme for the outer loop over all images: Projections are loaded and copied to the padded projection buffers in small chunks, i.e., b images at a time. The line update kernel (see Sect. 6.4) for a certain pair of (y,z) coordinates is then executed b times, once for each projection. This corresponds to a b-way unrolling of the image loop and a subsequent jam into the next-to-innermost voxel loop (across the y voxel coordinate). At the problem sizes studied here, all the voxel data for this line can be kept in the L1 cache and reused $b-1$ times. Hence, the complete volume is only updated in memory $496/b$ instead of 496 times. Relatively small unrolling factors between 2 and 8 are thus sufficient to reduce the bandwidth requirements to uncritical levels even on "starved" processors like the Intel Harpertown.

This optimization is so effective that it renders proper ccNUMA placement all but obsolete; we will thus not report the benefit of ccNUMA placement in our performance results, although it is certainly performed in the code.

6.7 Results

In order to evaluate the benefit of our optimizations we have benchmarked different code versions with the 512^3 case on all test machines. RABBITCT includes a benchmarking application, which takes care of timing and error checking. It reports total runtime in seconds for the complete backprojection. We performed additional hardware performance counter measurements using the likwid-perfctr tool. likwid-perfctr can produce high-resolution timelines of counter data and useful derived metrics on the core and node level without changes to the source code. Unless stated otherwise we always report results using two SMT threads per core. For all architectures apart from Sandy Bridge the line update kernel version V2 was used. On Sandy Bridge results for the SSE kernel V1 as well as for the AVX port of the V1 kernel are presented.

6.7.1 Validation of analytical predictions

To validate the predicted performance of the analytic model (see Sect. 6.5), single-socket runs were performed without the clipping optimization and SMT. Blocking was used on the Harpertown platform only, to ensure that execution is not dominated by memory access. The following table shows the measured performance and the deviation against the model prediction:

	HPT	WEM	WEX	SNB	SNB (AVX)
Perf. [GUP/s]	0.75	1.20	1.30	1.11	1.28
deviation	-13.3%	-18.3%	-11.5%	-12.6%	-18.0%

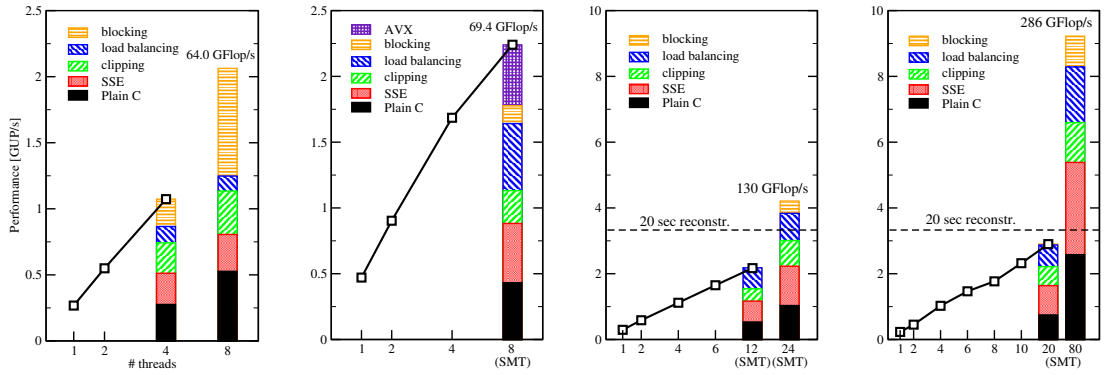
This demonstrates that the model has a reasonable predictive power. It has been confirmed that the contribution of data transfers indeed vanishes against the core runtime, despite the fact that the total transfer volume is high and a first rough estimate based on data transfers and arithmetic throughput alone (Sect. 6.3) predicted a bandwidth limitation of this algorithm on all machines.

As a general rule, the IACA tool can provide a rough estimate of the innermost loop kernel runtime via static code analysis. Still it is necessary to further enhance the machine model to improve the accuracy of the predictions. Especially the ability of the out of order scheduler to exploit superscalar execution was overestimated and has led to qualitatively wrong predictions.

Note that this example is an extreme case with all data transfers vanishing against core runtime. However, the approach also works for bandwidth-limited codes, as was shown in [4].

6.7.2 Parallel results

Figures 6.7 (a)–(d) display a summary of all performance results on node and socket levels, and parallel scaling inside one socket for the best version on each architecture. All machines show nearly ideal scaling inside one socket when using physical cores only. With SMT, the benefit is considerable on Sandy Bridge (33%) and Westmere (31%), and a little smaller on Westmere EX (25%). The large effect on Sandy Bridge may be attributed to a higher number of bubbles in the pipeline, as indicated by the larger discrepancy between the “throughput” and “critical path” cycles in the AVX loop kernel (see Sect. 6.5.2). Scalability from one to all sockets of the node is also close to perfect for the multisocket machines, with the exception of Westmere EX, on which there is a slight load imbalance due to 80 threads working on only 512 slices.



(a) Harpertown (b) Sandy Bridge (c) Westmere (d) Westmere EX

Figure 6.7: Scalability and performance results for the 512^3 test case on all platforms. In-socket scalability was tested using the best version of the SIMD-vectorized line update kernel on each system (AVX-V1 on Sandy Bridge, SSE-V2 on all others). The practical performance goal for complete reconstruction (20 seconds runtime, corresponding to 3.33 GUP/s) is indicated as a dashed line. GF/s numbers have been computed assuming 31 flops per optimized (scalar) inner loop iteration. Note the scale change between the left and right pairs of graphs.

Depending on the architecture, SSE vectorization boosts performance by a factor of 2–3 on the socket level. As explained earlier (see Sect. 6.4), part 2 of the algorithm prohibits the optimal speedup of 4 because its runtime is linear in the SIMD vector length. Work reduction through clipping alone shows only limited effect due to load imbalance, but this can be remedied by an appropriate cyclic OpenMP scheduling, as described in Sect. 6.6. This kind of load balancing not only improves the work distribution but also leads to a more similar access pattern to the projection images across all threads.

Cache blocking has little to no effect on all architectures except Harpertown, as predicted by the analysis.

The benefit of AVX on Sandy Bridge falls short of expectations for the same reason as in the SSE case. Still it is remarkable that the desktop Sandy Bridge system outperforms the dual-socket Harpertown server node, which features twice the number of cores at a similar clock speed. Both Westmere and Westmere EX meet the performance requirements of at most 20 sec for a complete volume reconstruction. The Westmere EX node is, however, not competitive due to its unfavorable price to performance ratio. It is an option if absolute performance is the only criterion.

6.8 Conclusion

6.8.1 Summary of results

Several algorithmic and low-level optimizations for a CT backprojection algorithm were demonstrated on current Intel x86 multicore processors. Highly optimizing compilers were not able to deliver useful SIMD-vectorized code. The chosen implementation is thus based on assembly language and vectorized using the standard instruction set extensions SSE and AVX. The results

show that commodity hardware can be competitive with special-purpose hardware clinically relevant 512^3 voxel case at the same level of accuracy. Nonpipelined divide instructions (`divps`) or a fast pipelined version (`rcpps`) with subsequent Newton-Raphson iteration proved to be equivalent in terms of accuracy and performance. Compared to a pure reciprocal they provide better accuracy at a 10% performance penalty. The standard dual-socket server system Intel Westmere EP is easily able to beat the 20 s limit for the full backprojection, reaching 15.8 s. Preliminary tests on an Intel Sandy Bridge EP Xeon platform (8 cores per socket) showed that runtimes close to the GPU results are in reach for modestly priced dual-socket servers.

It was shown that it is necessary to consider all aspects of processor and system architecture in order to reach best performance, and that the effects of different optimizations are closely connected to each other. The benefit of the AVX instruction set on Sandy Bridge was limited due to the lack of a gathered load and the small number of instructions that natively operate on the full SIMD register width. This relevant algorithm can achieve very good efficiency on commodity processors and it would be a natural step to further improve performance with a distributed memory implementation. At higher resolutions, which are used in industrial applications, multicore systems are frequently the only choice (apart from expensive custom solutions).

6.8.2 Reassessment in view of performance patterns

The first shot at performance modeling for the backprojection used the roofline model, based on arithmetic throughput and memory bandwidth as the applicable bottlenecks. The model predicted *bandwidth limitation* as the relevant pattern on all architectures at hand, which could easily be ruled out by a simple runtime measurement. After some basic optimizations, notably work reduction and SIMD vectorization, an ECM model was set up. Using the output from the IACA tool as the in-core baseline, the conclusion was that in-core execution was the bottleneck even with a full socket on all but the very bandwidth-starved Intel Harpertown platform. A combination of patterns applied, from *pipelining hazards* (dependencies along the critical code path) to *instruction overhead* and *ineffective instructions* (SIMD-incompatible gather operations). Loop blocking was applied but only effective – as expected by the patterns – on the Harpertown CPU. The work reduction optimization mentioned above resulted in a strong *load imbalance* pattern with OpenMP-parallel code, which was identified by mediocre scalability across cores and confirmed by HPM measurements. The imbalance could be easily fixed by choosing an appropriate OpenMP loop schedule. In the end, the performance model was well within a 20% margin on all four architectures.

Chapter 7

A performance- and energy-optimized lattice-Boltzmann fluid solver [7]

Algorithms with low computational intensity show interesting performance and power consumption behavior on multicore processors. This has been demonstrated in Chapter 4 using the simple vector triad and Jacobi smoother benchmarks. The lattice-Boltzmann method (LBM) is widely used in computational fluid dynamics, and a prototype for many other memory-bound algorithms. It has gained popularity due to its ease of implementation and suitability for complex geometries. Despite its seeming simplicity, optimizing LBM on recent hardware platforms and for different application cases has been the subject of intense research in the last ten years [82, 83, 84, 85, 86, 47, 87, 88, 89, 90, 91, 92, 93]. In this chapter, a specific version of the LBM is used to show if and how single-chip performance and power characteristics can be generalized to the highly parallel case.

After a thorough analysis of a sparse-lattice two-relaxation-time (TRT) LBM implementation on the Intel Sandy Bridge processor, we use the ECM model and the multicore power model to describe the intra-chip saturation characteristics of the code and its optimal operating point in terms of energy to solution as a function of the propagation method [93], the clock frequency, and the SIMD vectorization. These findings are then extrapolated to the multi-node level on SuperMUC, where the energy-saving potential of various optimizations is quantified. One surprising result of this analysis is that the memory-bound nature of the LBM algorithm is partly lost when communication plays a significant role, and that it is then even more important to select the optimal operating point (number of cores, clock speed) to get minimal energy to solution. Adding a power capping condition will complicate matters and may make otherwise sensible decisions for an operating point inaccessible. It is found that simplistic measures often applied by users and computing centers, such as setting a low clock speed for memory-bound applications, have limited impact.

7.1 Introduction

7.1.1 Related work

Performance modeling and prediction especially in the context of LBM are an ongoing research topic of many groups in engineering and computer science [94, 95]. Auto-tuning was used, e.g.,

in [96] for a magnetohydrodynamics LBM. The "ILBDC"[1] LBM code used here has been optimized previously [97] and its sustained performance is close to predictions from the roofline model [93]. Research in the direction of energy-saving hardware and software mechanisms focuses on models and algorithms for dynamic voltage and frequency scaling (DVFS) and dynamic concurrency throttling (DCT) [44].

The unique combination of the ECM model and the multicore power model used here allows a new view on energy consumption issues of LBM and other memory-bound algorithms. The observation that MPI (inter- and intra-node) communication must be viewed as a highly frequency-dependent overhead adds a new twist.

7.1.2 The lattice-Boltzmann method

The lattice-Boltzmann method is an algorithm for computational fluid dynamics (CFD). Instead of discretizing and solving the Navier-Stokes equations, which contain macroscopic entities such as pressure and velocity, the LBM is based on the Boltzmann equation, which describes the temporal evolution of a time-dependent *single-particle distribution function* (PDFs) $f(\vec{x},\vec{\xi},t)$. The PDF quantifies the probability density for finding a particle at position $\vec{x}$ with microscopic velocity $\vec{\xi}$. The Boltzmann equation for f is

$$\frac{\partial f}{\partial t}+\vec{\xi}\frac{\partial f}{\partial \vec{x}}+\vec{F}\frac{\partial f}{\partial \xi}=\mathcal{Q}(f,f)\,, \tag{7.1}$$

where $\vec{F}=\vec{K}/m$, with $\vec{K}$ being external forces exerted on particles with mass m. The left-hand side of this equation describes advection processes, while the right-hand side is a *collision integral*, whose dependence on (f,f) is written to clarify that only two-particle collisions are considered. Eq. (7.1) is thus an integral-differential equation. Since the collision integral makes solving this equation very complex, approximations have been developed, which keep important properties such as energy and momentum conservation but are simple enough to be applied in practice. One prominent example is the Bhatnagar-Gross-Krook operator [98]

$$\mathcal{Q}^{\mathrm{BGK}}(f,f)=-\frac{1}{\tau}\left(f-f^{(0)}\right)\,, \tag{7.2}$$

where the *relaxation time* τ quantifies how quickly the local thermodynamic equilibrium (Maxwell-Boltzmann) distribution $f^{(0)}$ can be attained. Macroscopic quantities can be obtained from the PDF by calculating moments. For pressure and velocity one has, e.g.,

$$\rho(\vec{x},t) = \iiint_{-\infty}^{\infty}\vec{\xi}^{\,0}f(\vec{x},\vec{\xi},t)\,\mathrm{d}\vec{\xi} \tag{7.3}$$

$$\rho(\vec{x},t)\vec{u}(\vec{x},t) = \iiint_{-\infty}^{\infty}\vec{\xi}^{\,1}f(\vec{x},\vec{\xi},t)\,\mathrm{d}\vec{\xi}\,. \tag{7.4}$$

The lattice-Boltzmann equation is obtained through a discretization of the velocity space and the spatial and temporal derivatives in the Boltzmann equation (7.1), which leads to [99]

$$f_i(\vec{x}+\vec{e}_i\Delta t,t+\Delta t)=f_i(\vec{x},t)+\Omega(f_i(\vec{x},t),f_i(\vec{x},t))\,. \tag{7.5}$$

[1] International Lattice-Boltzmann Development Consortium

Since there is now a number of discrete velocity vectors $\vec{e}_i$, there is one PDF f_i for each of them. The basic properties of the so obtained discrete phase space are usually labeled by DnQm, where n is the number of spatial dimensions and m is the number of discrete velocity directions. In the following we will concentrate on the D3Q19 model, which is a three-dimensional with 19 discrete PDFs. Macroscopic quantities can be obtained by discrete sums over the PDFs.

The right-hand side of (7.5) contains the *collision operator* Ω, which takes the role of the collision integral in (7.1). Beyond a straightforward discretization of the BGK operator (7.2), several other schemes for the collision operator have been devised, which enhance the stability of the algorithm. The variant used here is the (TRT) approximation of the collision process, which [100, 99, 101, 102] is based on the evolution operator

$$\begin{aligned} \Omega_i^{\mathrm{TRT}}(\rho(\vec{x},t),\vec{u}(\vec{x},t)) &= \lambda_e(f_i^+ - f_i^{\mathrm{eq}+}) + \lambda_D(f_i^- - f_i^{\mathrm{eq}-}), \\ \text{with} \qquad f_i^{\pm} &= \frac{1}{2}\left[f_i(\vec{x},t) \pm f_{\bar{\imath}}(\vec{x},t)\right] \\ \text{and} \qquad f_i^{\mathrm{eq}\pm} &= \frac{1}{2}\left[f_i^{\mathrm{eq}}(\rho(\vec{x},t),\vec{u}(\vec{x},t)) \pm f_{\bar{\imath}}^{\mathrm{eq}}(\rho(\vec{x},t),\vec{u}(\vec{x},t))\right] . \end{aligned} \tag{7.6}$$

Here, i and $\bar{\imath}$ denote opposite directions, so that $\vec{e}_{\bar{\imath}} = -\vec{e}_i$. In the low Mach number limit, an appropriate discretized equilibrium distribution function is [102]

$$f_i^{\mathrm{eq}}(\rho,\vec{u}) = w_i \left\{ p + \left[\frac{3}{c^2}\vec{c}_i\vec{u} + \frac{9}{2c^4}(\vec{c}_i\vec{u})^2 - \frac{3}{2c^2}\vec{u}\vec{u} \right] \right\} . \tag{7.7}$$

The weights w_i depend on the model (i.e., DnQm) and the direction i.

7.1.3 Implementation options and data traffic analysis for LBM

Lattice-Boltzmann methods have become a popular approach in computational fluid dynamics. However, they are also an interesting field of study for computer science, as the core algorithm uses a stencil-like access pattern with vector instead of scalar data, resulting in many concurrent memory streams and no reuse of data in a single iteration. The LBM is straightforward to parallelize, in shared memory as well as distributed memory. In the latter case, domain decomposition and simple halo-based next-neighbor communication is employed.

The *ILBDC* code [97] uses a D3Q19 lattice model and implements the two-relaxation-time (TRT) collision operator (7.6). All calculations are performed in double-precision floating point arithmetic. The algorithm with the D3Q19 model can be viewed as a 19-point stencil in 3-D accessing only nearest neighbors, but has two important differences to common stencil algorithms: (i) each lattice node consists not only of one, but of 19 values (the PDFs); (ii) each PDF read from or updated in memory is only accessed again in the next time step, which prevents data reuse (unless complex temporal blocking schemes are employed).

The performance of a given LBM approach depends at least on the data layout and memory access patterns, the scale of arithmetic operations (i.e., how well numerical expressions are simplified and combined to avoid unnecessary operations), and their degree of SIMD vectorization. A thorough overview of popular propagation step implementations and their memory access characteristics can be found in [88, 93].

It seems natural to store the PDFs in a 4-D array with an additional Boolean array, which is used to distinguish fluid and solid nodes. This is known as the *marker-and-cell* approach. However, LBM simulations of domains with a large fraction of solid nodes can benefit from

a *sparse representation* of the domain [82, 83, 86, 87, 91, 92], where only the fluid nodes are kept in a 1-D vector. Indirect accesses to PDFs of neighboring nodes are then required and accomplished through an adjacency list (`IDX`), which represents the topological connections of the nodes. ILBDC uses such a sparse representation.

For updating one node, optimized implementations read one PDF from each of the 19 surrounding neighbors (streaming step), compute updated values (collision step), and write the results to the PDFs of the local node. This is known as the *pull* scheme [47]. It is implemented in the ILBDC code together with a *structure-of-arrays* (SoA) data layout where all PDFs of a direction are stored consecutively in memory before the next direction follows.

We choose fluid-only lattice site updates as the sensible unit of work. To work around the data dependency problems of a combined stream-collide step, two lattices are often used, one as the source and one as the destination. Then, a fluid lattice-node update (FLUP) requires 19 PDF loads, 19 additional PDF loads because of write-allocate transfers, 19 PDF stores and 18 `IDX` loads of the adjacency list. Assuming double-precision floating-point numbers (eight bytes) for PDF and four-byte integers for `IDX`, the total number of bytes that must be transferred between CPU and memory for one FLUP is $3 \times 19 \times 8$ (PDF) $+ 18 \times 4$ (`IDX`) bytes $= 528$ bytes. With non-temporal store instructions the write-allocates are avoided and the data is directly written from the processor into memory, bypassing the cache hierarchy. The number of bytes required for one FLUP decreases then to $2 \times 19 \times 8$ (PDF) $+ 18 \times 4$ (`IDX`) bytes $= 376$ bytes. Current standard processors have difficulties sustaining the full memory bandwidth with 19 concurrent write streams, in particular if they consist of non-temporal stores. As a remedy, a blocking/stripmining scheme can be applied so that a node's PDFs are read in chunks and updated values are stored in a small temporary buffer, which should be small enough to fit in the L1 cache. From this buffer, two directions of the updated PDFs at a time are written with non-temporal stores to the destination lattice. We call this implementation *pull-split no-NT* or *pull-split NT*, depending on whether normal stores or non-temporal stores are used.

BAILEY'S *AA pattern* [90] for the PDF access allows using one single lattice only (instead of separate source and destination grids) while maintaining the possibility to update all cells in any order and in parallel. It was originally conceived for optimizing LBM on GPGPU platforms, but can be applied on multicore processors as well. The iterations over the lattice are divided into even and odd time steps. During an even step only PDFs of the current node are accessed in each lattice site update. At the following odd step only PDFs of the neighboring nodes are accessed, which requires indirect addressing in our case of the sparse representation. With this update scheme only stores to locations in memory occur which have previously been read. No write-allocate will be performed as the data to be updated already resides in the cache. We use an optimized version where the even time step is completely SIMD-vectorizable (SSE/AVX), which can easily be accomplished as all PDFs are accessed consecutively and no indirect access is required. In the odd time step a partial vectorization is performed, which can avoid the indirect addressing and allows for vectorized execution of consecutively stored chunks of PDFs. Nodes that cannot be treated in this way are updated without SIMD vectorization (i.e., in scalar mode). The fraction of nodes which can be updated with SIMD operations depends on the geometry used for the simulation. During even time steps $2 \times 19 \times 8$ (PDF) bytes $= 304$ bytes per FLUP are required. In the odd time step the number of bytes needed for one FLUP depends on the fraction of vectorizable updates. The lower bound is the case when all updates can be vectorized. Here only $2 \times 19 \times 8$ bytes $= 304$ bytes are required. The upper bound is reached when all updates must be scalar and indirect accesses are required, which results in $2 \times 19 \times$

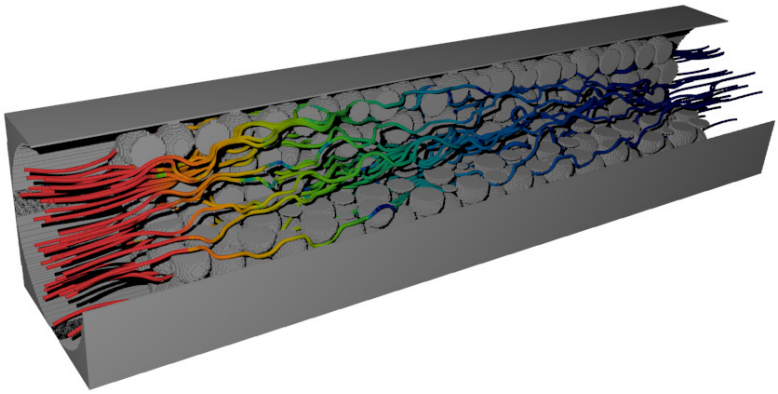


Figure 7.1: Visualization of the packed bed reactor geometry used in the benchmarks.

8 (PDF) + 18 × 4 (`IDX`) bytes = 376 bytes.

In order to have full control over the code vectorization, all performance-critical parts were implemented using SIMD compiler intrinsics.

7.1.4 Test bed and benchmark cases

Since SuperMUC does not easily allow an arbitrary frequency setting, and turbo mode is inaccessible at all, all single-node benchmark tests were run on a standalone Sandy Bridge EP node (see Sect. 2.4.1) with the same type of CPU and otherwise similar characteristics as one SuperMUC node Each MPI process was explicitly pinned to its physical core using `sched_setaffinity()` within the code. All benchmarks were run inside a single island to guarantee that communication is performed through the fully non-blocking fat tree.

Two geometries were selected for the benchmarks. The first is an empty channel consisting only of fluid nodes except for the walls. The second geometry is a *packed bed reactor,* i.e., a tube filled with spheres (see Fig. 7.1). It represents a real world application case for flow simulation with this type of code. Both geometries have dimensions of $4000 \times 80 \times 80$ nodes, resulting in $25 \cdot 10^6$ fluid nodes (≈ 3.8 GB lattice + 1.8 GB adjacency list) and $19 \cdot 10^6$ fluid nodes (≈ 2.9 GB lattice + 1.4 GB adjacency list) for the channel and the reactor geometry, respectively.

With these dimensions both geometries fit into the NUMA locality domain of one socket on SuperMUC. For strong scaling runs the reactor geometry was enlarged to $8000 \times 160 \times 160$ nodes, as the smaller lattice fits in the L3 caches of 128 compute nodes and above. The large reactor geometry consists of $157 \cdot 10^6$ fluid nodes and requires around 24 GB of memory for the lattice and around 11 GB for the adjacency list.

The ILBDC code is purely MPI-parallel. All single-node measurements were thus performed with intra-node MPI only; no significant changes are expected from a hybrid MPI/OpenMP version, but this will be investigated in the future. Details abut the MPI parallelization can be found in Sect. 7.5.1.

7.2 Chip-level performance and scaling

As a motivation for a thorough performance analysis, Fig. 7.2 shows the intra-socket scaling of the empty channel test case with the AA pattern for the two "extremal" clock frequencies of 2.7 GHz and 1.2 GHz, respectively, in three variants: AVX-vectorized (full 256-bit loads/stores),

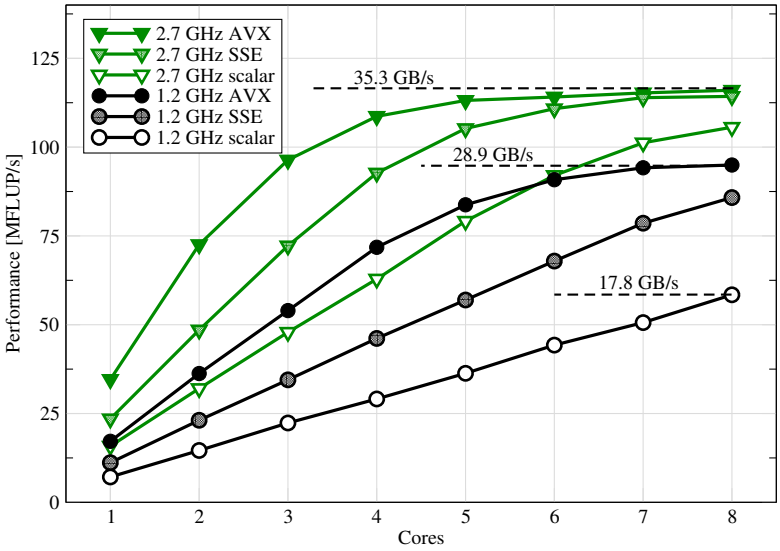


Figure 7.2: Intra-socket strong scaling of the AA pattern LBM implementation for an empty channel, comparing AVX, SSE, and scalar code at the clock frequencies of 2.7 GHz (triangles) and 1.2 GHz (circles). The corresponding memory bandwidths are indicated for selected cases.

SSE-vectorized, and scalar. The data indicates that SIMD vectorization has a large impact in the serial case; in fact, the serial performance differs by more than a factor of two between the AVX and the scalar code. At 2.7 GHz, the gap closes as the number of cores is increased. On the full socket the scalar code is hardly 10% slower than the AVX variant. The latter, however, reaches the same level already with four cores, which opens an opportunity for saving energy by leaving cores idle.

At 1.2 GHz, the situation in the serial case is similar, but on a lower level. The single-core performance of all code variants is roughly proportional to clock speed. However, only the AVX-vectorized code shows a saturation pattern, while the SSE and scalar variants scale linearly up to eight cores without reaching a bandwidth barrier. Hence, lack of vectorization ("slow code") cannot be compensated by using more cores in this case. Moreover, the maximum memory bandwidth is correlated with the core clock frequency and varies by about 20% across the full frequency range [42].

Figure 7.3 shows a socket-level performance comparison of the scalar and vectorized AA pattern implementation with the pull-split pattern for both application cases (empty channel vs. packed reactor) at a clock speed of 2.7 GHz. Although there is a large fraction of obstacles in the packed reactor geometry, their presence hardly influences the performance, independent of the propagation pattern. This shows the clear superiority of the sparse lattice representation in the ILBDC code over the simple marker-and-cell approach. When a large fraction of the cells are obstacles, marker-and-cell inevitably loses performance because of a low vectorization ratio, leading to late (or no) saturation. We also see that the pull-split pattern is not competitive, since it cannot by far saturate the memory bandwidth of the chip.

The intention of applying the ECM model is to gain deeper insight into this performance behavior, and to pave the way for a practically useful energy consumption analysis.

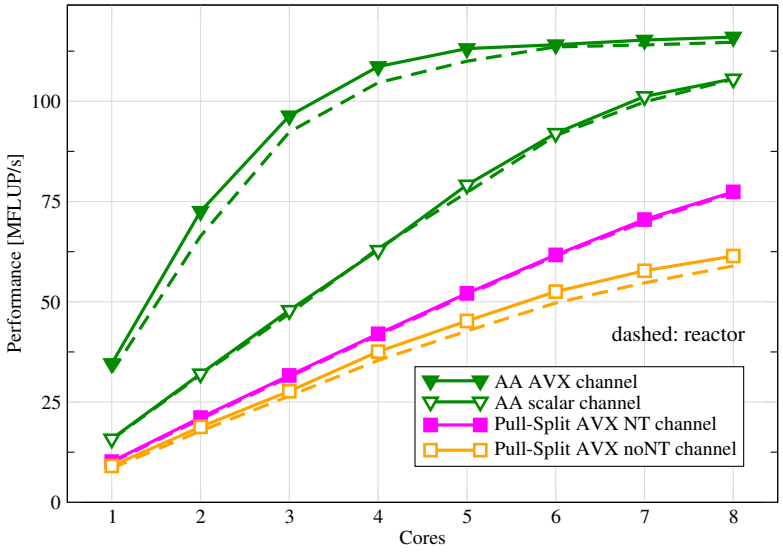


Figure 7.3: Intra-socket scaling at 2.7 GHz: AA pattern in AVX and scalar variants (triangles) and pull-split pattern with AVX vectorization with and without non-temporal stores (squares), for the empty channel application case (solid lines). The performance numbers for the packed reactor case are shown with dashed lines.

7.3 ECM model for the ILBDC code

7.3.1 In-core analysis

An IACA [38] throughput analysis for the AA pattern kernel shows that the ADD port of the SNB core is the sole bottleneck of core execution for all variants (scalar, SSE, AVX), as well as for even and odd time steps, and that one loop iteration (four updates with AVX, two with SSE, one for scalar) should take about 135 cycles. In contrast, a critical path analysis reports somewhat longer execution times due to dependencies in the instruction and data flow. The critical path depends on the type of time step and has a maximum length of 163 cycles (even, AVX), 212 cycles (odd, AVX), 160 cycles (even, scalar), and 187 cycles (odd, scalar). This prediction roughly coincides with direct measurements, which we will use as an input in the following (160, 212, 158, and 160 cycles, respectively). These numbers must be multiplied by two (for AVX) or eight (for scalar) for getting execution times for one unit of work, i.e., a cache line (see the table in Fig. 7.4).

The analysis for the packed reactor case is surprisingly similar: The even time step does not change at all, since no index access is required. In the odd time step, even when assuming no potential for vectorization (as would be the case for an extremely porous geometry) there is ample room for hiding the additional loads for the index array due to the bottleneck on the ADD port. This step is necessarily scalar, however, so the execution time is about four times longer per unit of work. The actual impact of this slowdown depends on the fraction of vectorizable updates. In the applications covered here, this fraction is roughly 97% for the empty channel and 92% for the packed reactor case, leading to a very small performance penalty for the latter, which was already observed in Fig. 7.3. Hence, only the empty channel case will be considered for the rest of the chip-level analysis.

7.3.2 Data transfers and saturation behavior on the chip

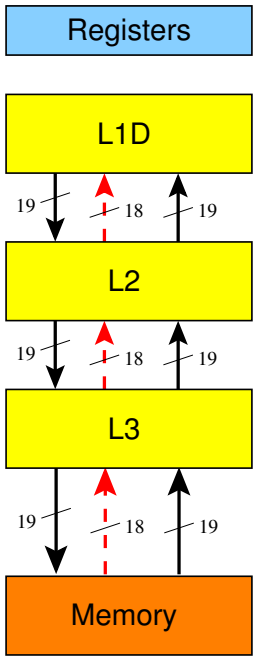
The ECM model requires the maximum attainable memory bandwidth as an input parameter. It is known that this value depends on the number of parallel read/write streams as well as the CPU

Listing 7.1: Parallel multi-stream update benchmark with 19 streams. `N` is chosen such that the arrays do not fit in any cache.

```
1 double a01[N], a02[N],..., a19[N], s=2.0;
2 #pragma omp parallel for
3  for(int i=0; i<N; ++i) {
4    a01[i] = s * a01[i];
5    a02[i] = s * a02[i];
6       ...
7    a19[i] = s * a19[i];
8  }
```

clock speed. From a data transfer perspective, the AA-pattern implementation of the D3Q19 LBM algorithm reads 19 arrays from memory, modifies their contents, and writes them back. In order to get the maximum memory bandwidth on the socket we hence use a parallel multi-stream array update benchmark (see Listing 7.1). It is designed to mimic the data streaming behavior of the LBM algorithm.

Figure 7.5 shows the achieved memory bandwidth on one SNB socket with varying number of threads (cores) and clock frequencies between 1.2 GHz and 2.7 GHz (plus turbo mode). As



	even	odd
AVX	320 cy	424 cy
scalar	1264 cy	1280 cy

2 19 2 cy [+ 18 cy] = 76 cy [+ 18 cy]

2 19 2 cy [+ 18 cy] = 76 cy [+ 18 cy]

(2 19 64) B 2.7 Gcy/s / 36 GB/s) [+ 43 cy]
= 182 cy [+ 43 cy] @ 2.7 GHz
or
(2 19 64) B 1.7 Gcy/s / 33 GB/s) [+ 30 cy]
= 125 cy [+ 30 cy] @ 1.7 GHz

Figure 7.4: Single-core ECM model of the AA propagation pattern for D3Q19 LBM (eight FLUPs). Even and odd time steps have different in-core timings. One arrow represents the number of full cache line transfers indicated; dashed arrows stand for half-wide (32-byte) transfers and are required for loading the adjacency information in the odd time step when vectorization is not possible. One half-wide cache line transfer takes one cycle. Numbers in square brackets denote contributions from the adjacency list, and can be ignored for the empty channel case.

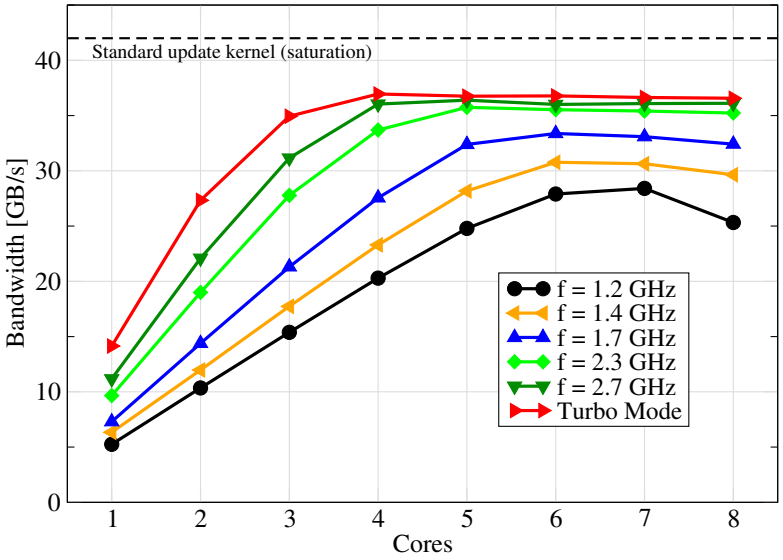


Figure 7.5: Multi-stream update benchmark performance scaling on one SNB socket with different CPU frequency settings. 19 update streams were run per thread. The dashed line indicates the maximum achievable bandwidth with a simple single-array update kernel.

predicted by the ECM model, the single-thread performance is proportional to the clock speed, and the saturation point is shifted to larger thread counts as the clock speed decreases: While saturation is reached near three cores with turbo mode, up to six cores are needed at the lowest frequencies. Due to the large number of read/write streams, the maximum bandwidth is significantly lower than with a standard single-stream update kernel (dashed line in Fig. 7.5). At the same time, the maximum (saturation) memory bandwidth drops by about 25% over the whole frequency range; there is another substantial drop when using the full socket (eight cores) at the lowest frequency. As of now there is no conclusive explanation for these latter effects. They do, however, influence the considerations on energy dissipation, which will be discussed in Sect. 7.4. In the following, the maximum bandwidths as measured at the respective frequencies will be used as an input to the ECM model in order to calculate the number of cycles required to transfer cache lines between memory and L3 cache.

Figure 7.4 shows the complete ECM model analysis at 2.7 and 1.7 GHz, respectively. The cycle counts in square brackets are contributions from loading the adjacency information (dashed arrows), and can be ignored for the empty channel case. The achievable memory bandwidth (36 GB/s and 33 GB/s, respectively) and the clock speed enter the model when calculating the cycles for data transfers to and from main memory. Data transfers between adjacent cache levels are assumed occur at 32 bytes per cycle, so these cycle counts are independent of the clock frequency. The various execution and data transfer times may be combined in different ways to arrive at a performance prediction for the serial program:

1. The most conservative (worst case) assumption is that none of those contributions overlap with each other, so that the execution time is equal to their sum (e.g., 320+2·76+182=654 cycles for the even time step with AVX at 2.7 GHz).

2. The most optimistic assumption is that the cycles in which the L1 cache is occupied by loads and stores from the core cannot be used for reloads and evicts to L2, but all other contributions do overlap.

3. Lastly one may assume that the pure in-core execution part (everything except loads and

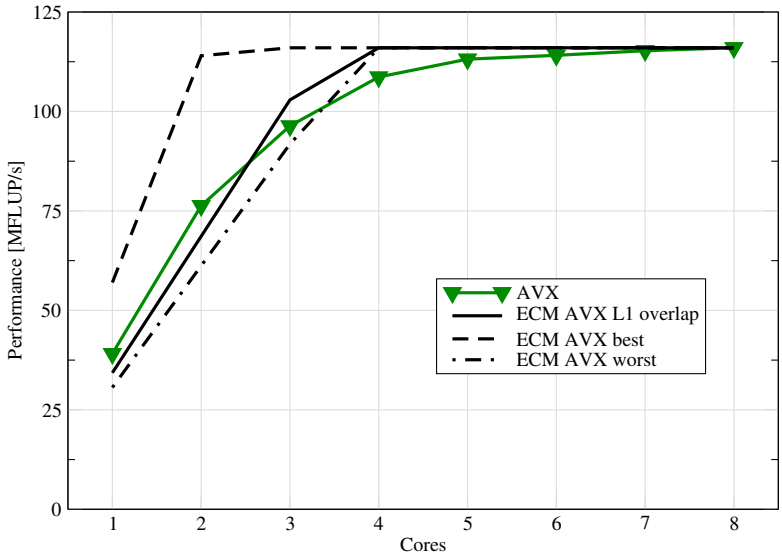


Figure 7.6: Performance of the AVX implementation of the AA pattern (empty channel) at 2.7 GHz (triangles). The ECM model predictions for AVX with full overlap assumption (dashed line), no overlap (dotted-dashed line), and partial overlap at L1 (solid line) are shown for comparison.

stores) can overlap with loads and evicts from/to the L2 cache, but that there is no overlap beyond that.

None of these assumptions coincides with the roofline model, which requires the achievable memory bandwidth for each number of cores as an input parameter. The ECM model only requires the maximum (saturated) bandwidth, and predicts the scaling.

7.3.3 Validation of the performance model

Figure 7.6 shows a comparison of the measured performance for the AVX-vectorized AA pattern implementation with the three models described above. Apart from the region around the saturation point (3–4 cores), the third assumption provides the best fit to the data.

It was already shown in Fig. 7.3 that the pull-split propagation pattern (with and without NT stores) is not competitive since it cannot saturate the memory bandwidth, although the NT version has almost the same computational intensity as the AA pattern. This failure can mainly be attributed to the fact that the pull-split variant cannot be efficiently SIMD-vectorized on the Sandy Bridge architecture due to the indirect access in every lattice site update. More specifically, the loop which loads the neighboring distribution functions and stores intermediate results into temporary buffers is scalar. The pull-split pattern will thus be ignored from now on, and the focus of the following discussion will be on the AA pattern.

7.4 Power model

Many applications in computational science are memory-bound on modern processors, LBM being but a prominent example. The prevalent questions arising in this context are (i) How can a parallel code be run so that its overall energy consumption until a solution is reached can be minimized, preferably under the constraint of constant time to solution? and (ii) How can a parallel computer be operated in a production environment so that overall power dissipation is minimized or kept below a given maximum?

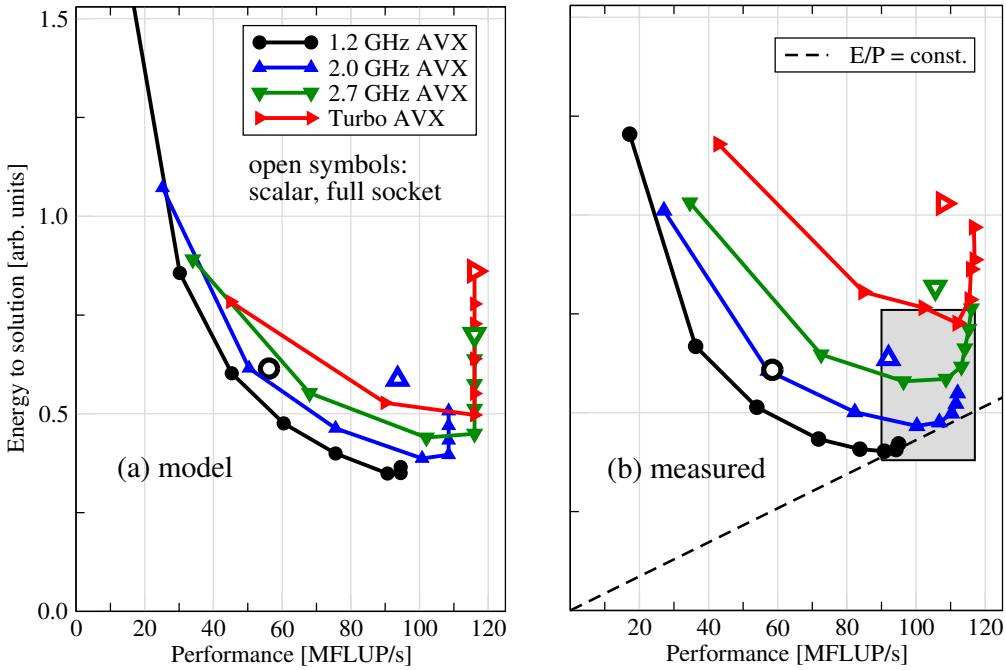


Figure 7.7: Energy to solution vs. performance ("Z-plot") of the AVX-vectorized LBM AA pattern implementation (empty channel case) of one SNB socket for different clock frequencies (lines and filled symbols). The number of cores used is the parameter along each data set. (a) Predictions by the ECM performance model and the chip-level power model. (b) Measured data. For comparison, the big open symbols mark the energy and performance of the scalar code on a full socket. The shaded area is the region defined by absolute minimum energy and saturated performance for he AVX versions. The dashed line is the line of constant energy-delay product that hits the saturation point of the lowest-frequency run.

We concentrate on the first question here, and employ the ECM model together with the multicore power model developed in Chapter 4. For simplicity we neglect the linear power coefficient W_1 in (4.4), since it is usually small compared to W_0 and W_2.

7.4.1 Energy to solution for the LBM solver on the chip

The ECM model and the power model enable a combined analysis of the energy and performance properties of the LBM algorithm. It is useful to put energy and performance data in a single graph, which we call a "Z-plot."[2] Energy to solution is plotted versus performance, with the number of cores used as a parameter within a data set for a specific frequency, SIMD vectorization variant, propagation method, or other property. This has been done in Fig. 7.7a for three different clock frequencies and turbo mode, using the AA pattern in AVX and scalar variants. In turbo mode, each data point was computed using the maximum allowed frequency for each number of active cores. The corresponding measurements are shown in Fig 7.7b. Note that we always show energy to solution in arbitrary units, but the values shown are coherent for

[2]The "Z" goes back to Dr. Thomas **Z**eiser, who first had the idea to present performance and energy data in this way.

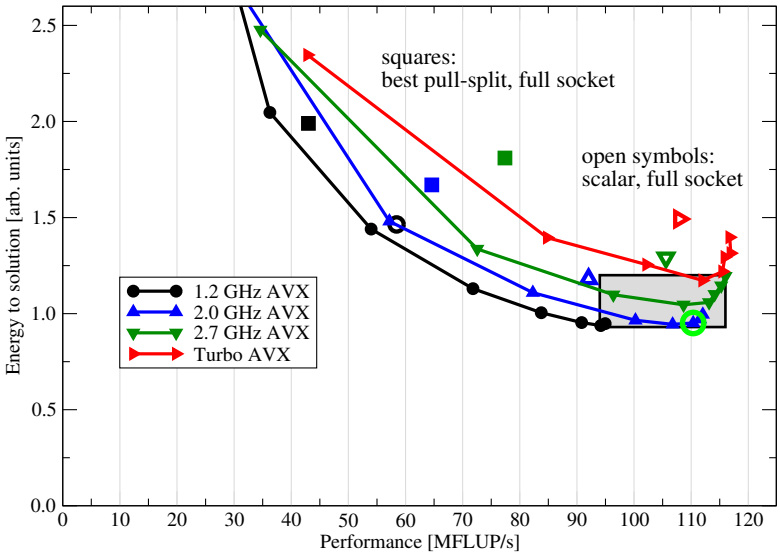


Figure 7.8: Same data as in Fig. 7.7b but with a power baseline of 50 W added to the socket. The circle marks a possible operating point for almost minimal energy with a tolerable loss in performance. For reference, the best pull-split data (vectorized, full socket) for 1.2, 2.0, and 2.7 GHz is also shown (filled squares).

a specific problem size (geometry and number of iterations).

The models are able to describe the qualitative features of energy and performance. The observed deviations are caused by (i) the inability of the ECM model to accurately describe the performance behavior in the vicinity of the saturation point, (ii) the inaccuracy in determining W_2 and W_0, and (iii) the approximation of linear power behavior with respect to core count even with saturated codes like LBM at higher clock speeds. In addition, turbo mode does not fit perfectly into the model (4.4) since the SNB chip can operate beyond its thermal design power (TDP) for a limited amount of time [41]. This is why the deviation from the measurements is especially large with turbo mode (right-pointing triangles in Fig. 7.7). Looking at the minimum energy point with respect to clock frequency and number of cores in the regime where performance is not saturated, we see that this point moves to smaller frequency as the core count goes up, as described by (4.11).

In general, all other things being equal, a faster sequential code (AVX instead of scalar) saves energy. Comparing energy to solution for the AVX codes at their respective saturation points, we can identify an "optimization space" (shaded area in Fig. 7.7b), in which the desired optimal operating point should be found. Depending on the emphasis one wants to put on energy minimization vs. maximum performance, this point may be in the lower left corner of the area. In this case one would use all cores at the lowest frequency (1.2 GHz, filled circles) and sacrifice about 20% of performance compared to the right edge of the area, which is defined by the saturation point at higher frequencies (2.0 GHz to turbo mode). Another clear conclusion is that turbo mode is of no good use for the LBM implementations studied here, neither from a performance nor from an energy point of view.

There is no single, well-defined criterion for identifying the optimal operating point on the chip level. One may certainly employ cost models such as the energy-delay product (ratio of energy and performance), but this is only one possible choice. For reference we have included a line of constant energy-delay product in Fig. 7.7b. From the data we have collected, using 5–6 cores at 2.0–2.3 GHz seems to provide a good compromise between performance loss and energy consumption ("as far on the lower right as possible").

While the model and the measurements yield a consistent picture on the chip level, it is clear

that the chip contributes only a (however significant) part to the overall power consumption of a compute node. As mentioned in the derivation of the power model, the rest of the system should be taken into account when assessing the real energy demand for running an application. We do this by setting $W_0 = 73\,\mathrm{W}$ for the chip-level baseline power, which amounts to roughly 300 W of node power (assuming two-socket nodes). This is also the value measured during a LINPACK run on SuperMUC [103]. With this change we can offset the energy measurements from Fig. 7.7 to arrive at the data shown in Fig. 7.8.

As expected, the modified baseline power leads to a reduction of the vertical spread between the measurements for different clock frequencies. While it was possible with the chip-level (i.e., small) W_0 to have a situation where energy to solution was heavily influenced by frequency and SIMD vectorization even at a specific performance level (with a spread of up to $2\times$ within the optimization space shown in Fig. 7.7), a large W_0 reduces the spread to about 25%. Hence, a large baseline power favors the "race to idle" principle where the most influential parameter is performance; optimizations that favor a larger saturation performance (such as the AA propagation pattern, or blocking schemes which increase the computational intensity) have the most potential for saving energy. In addition, optimized clock speed and a reduction of the number of cores used can yield second-order but still significant savings. Within the transformed optimization space (shaded area in Fig. 7.8) we can identify a possible optimal operating point at about 2.0 GHz and six cores, with almost minimal energy to solution and a performance loss of about 6% compared to the highest possible saturation level. In comparison to a naive strategy of running on all cores with turbo mode enabled and a scalar kernel, more than one third of the energy can be saved.

The "race to idle" principle with respect to maximum code performance is evident from a comparison with the energy-performance data for the pull-split pattern (filled squares) in the best variant (SSE or AVX vectorized, non-temporal stores, full socket) at three different frequencies in Fig. 7.8: The pull-split pattern can neither compete with AA in the performance nor in the energy dimension. Using AA, almost a factor of two in energy and 30–40% of runtime can be saved in comparison to pull-split.

7.5 Highly parallel LBM simulations

7.5.1 MPI parallelization in ILBDC

ILBDC uses an MPI parallelization with a static load balancing scheme. The sparse representation of the lattice is cut into equally sized chunks, so that each MPI rank receives the same number of fluid nodes (probably off by one). The interfaces of such generated partitions can be arbitrarily formed with different numbers of partition neighbors, as the simple cutting of the sparse representation does not consider any topological information. However, in the case of the channel and reactor benchmark geometries this method results only in a 1-D decomposition, where each rank only needs to exchange ghost PDFs with its two direct neighbors. A more extensive description of this approach can be found in [104]. We distribute the ranks linearly across the compute nodes, so that consecutive ranks are located nearby on the same node. In case of strong scaling the communication volume of a process stays constant, since each rank only gets a smaller segment of the long geometries when the number of processes goes up.

The packed bed reactor geometry was used for all the multi-node experiments, since it is the application scenario that is relevant in practice. We have shown earlier that the node-level

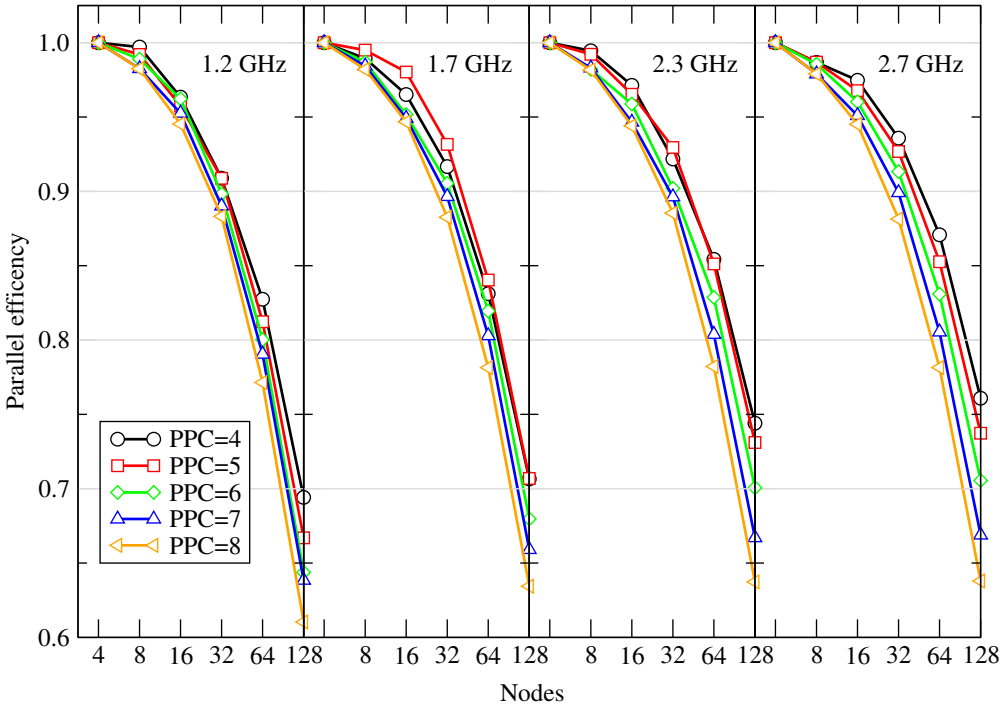


Figure 7.9: Parallel efficiency of the large packed bed reactor application case ($8000 \times 160 \times 160$ lattice nodes) for different frequency settings and different number of processes per chip (PPC) on up to 128 nodes of SuperMUC. The efficiency calculation was based on the four-node performance baseline.

performance (and thus power) properties are very similar to the empty channel case. All multi-node measurements were conducted on a single island of SuperMUC. Note also that "turbo mode" cannot be activated on SuperMUC, so we stick to the fixed frequencies of 1.2, 1.7, 2.3, and 2.7 GHz in the following.

7.5.2 Performance and energy at strong scaling

Parallel efficiency and communication performance

All variants of the AA pattern scale well up to 32 nodes (512 cores) at all frequencies, and parallel efficiency only starts to degrade below 90% beyond that point. Scaling experiments were performed on up to 128 nodes (2048 cores), since this is where some variants start to show efficiencies as low as 60%. We assume a sensible limit of 50–60% of parallel efficiency for production use in a computing center environment. A lower efficiency, which must be regarded as a waste of resources, should be justified by special needs, for instance when large aggregate memory is required. In Fig. 7.9 we show the parallel efficiency of the strong scaling runs versus the number of nodes at the four chosen frequencies and with between four and eight processors per chip (PPC). Since the application case is too large to fit on a single node, all efficiency numbers were normalized to the four-node run.

Usually one would expect the parallel efficiency to increase as the node-level performance goes down, because communication and synchronization overheads become less important

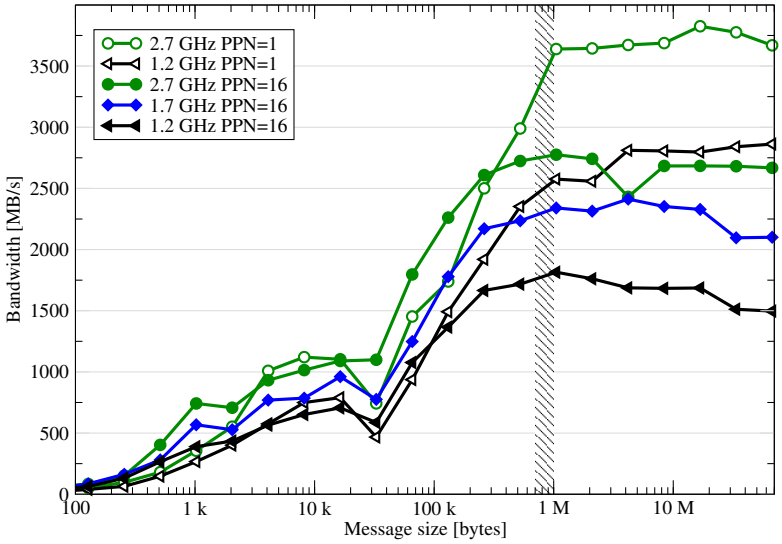


Figure 7.10: IMB sendrecv benchmark on two SuperMUC nodes at 1.2 and 2.7 GHz with 16 (filled symbols) and one process per node (open symbols). MPI ranks were mapped to cores for minimum inter-node traffic. The shaded area indicates the range of message sizes for the application test case (reactor).

when the pure compute time goes up. On SuperMUC, the opposite is the case: The minimum parallel efficiency (at 128 nodes) varies between 76% and 63% (depending on the number of cores per chip) for 2.7 GHz, but between 69 and 61% at 1.2 GHz. We conclude that there must be a frequency-dependent factor which impedes scalability whenever communication overhead plays a significant role.

In order to explore the reasons for this effect we have conducted experiments with "sendrecv" from the Intel MPI benchmark suite (IMB) [105], since it mimics the ringshift-like halo-exchange communication pattern of the ILBDC code. Each MPI process exchanges data with its neighbors: `MPI_Sendrecv(to right neighbor, from left neighbor)`. The benchmark reports the available communication bandwidth per process. In Fig. 7.10 we show the results for two SuperMUC nodes in the two corner cases of one process (PPN=1) and 16 processes per node (PPN=16) for the two extremal frequencies of 1.2 and 2.7 GHz. The placement of the MPI ranks was done in the same way as for the ILBDC benchmarks: Neighboring ranks were "packed" to the same node to minimize inter-node traffic.

Although both scenarios show a dependence of the effective MPI bandwidth on the clock speed, this is especially pronounced at PPN=16, and we see a breakdown of about 35% in communication bandwidth within the region of message sizes relevant for the ILBDC packed reactor benchmark (shaded area). Moreover, the bandwidth of the FDR-10 IB interface cannot be saturated even at the highest frequency setting with PPN=16. We attribute both effects to the dominance of intra-node communication, which has a strong dependence on clock speed. In contrast, the saturated LBM performance with the AA pattern and AVX vectorization only drops by about 20% over the whole frequency range (see Fig. 7.2). This explains the stronger breakdown of parallel efficiency at strong scaling for low clock speed and for a large number of cores per chip.

Energy and performance at scale

The question remains whether one can extrapolate the findings about energy to solution and performance from the chip to the multi-node level, and especially whether single-core optimizations, notable SIMD vectorization, have a similar impact. Figure 7.11 shows aggregated

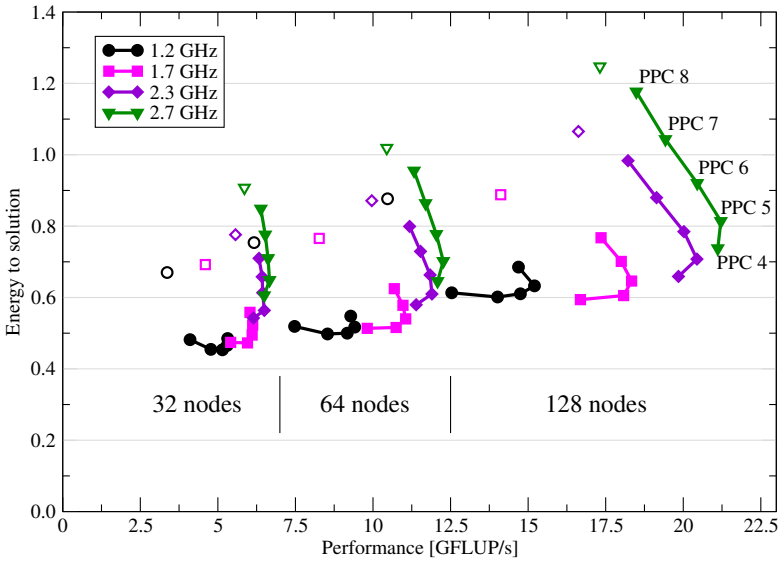


Figure 7.11: Multi-node energy to solution vs. performance for the AA pattern AVX LBM implementation (large reactor case) across clock speeds and node counts. The parameter along each curve is the number of processes per chip (4 … 8). For comparison, the open symbols show data for the scalar implementation on full sockets.

socket-level energy (as measured via RAPL) vs. performance with AVX for the three node counts (32, 64, and 128) at which parallel efficiency is between 90 and 60%. Along each curve, the number of processes per chip is increased from four to eight, and the highest energy point at the top of each curve is at PPC=8. For reference the corresponding lowest-energy data points for the scalar implementation are included (open symbols). The overall rise in energy to solution with growing node count is a trivial consequence of the decreasing parallel efficiency.

The most striking difference to the chip-level results is the notable performance degradation after the saturation point, especially at the larger node counts (32 and 64). It is caused by the drop in ringshift bandwidth (as described in the previous section) with growing PPC, and directly leads to a fast rise in energy to solution, much steeper than would be expected by the power model without communication component. Hence, it is even more crucial in the highly parallel case to select the optimal operating point, since each expendable core costs an over-proportional amount of energy: At 128 nodes and 2.7 GHz, the reduction in energy consumption when going from the full socket to the saturation point is over 40%, but only about 25% on a single chip (see the 2.7 GHz data in Fig. 7.7b).

The strong disadvantage of scalar execution can also be seen on the highly parallel level (open symbols in Fig. 7.11 show the “naive” operating point of PPC=8 for this case). Since more processes are needed to reach saturation – if this is possible at all –, the slowdown at larger PPC contributes strongly to the low performance and high energy consumption. As a consequence, a well-vectorized LBM code is instrumental for optimal energy to solution, particularly in the highly parallel case when communication plays a noticeable (but not dominant) role.

The question remains how these findings change if a realistic baseline power is used. Figure 7.12 shows the same data as Fig. 7.11 but with 100 W of constant power added per node. The results are very similar to the chip-level discussion in Sect. 7.4.1 above: All differences in energy to solution are damped by the larger idle power, but there is still more than 30% gain between a naive scalar code run with PPC=8 and the possible optimal operating point (marked in Fig. 7.12) with PPC=4 and 2.3 GHz. In contrast to the case where only the chip power is considered, the lowest frequency setting of 1.2 GHz is very unfavorable: The large performance degradation together with the communication bandwidth breakdown problem and the large base-

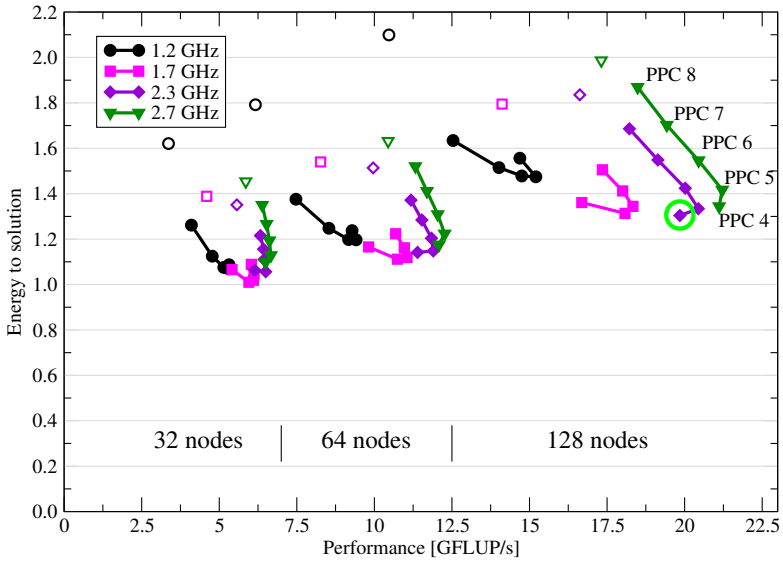


Figure 7.12: Same data as in Fig. 7.11 but with a realistic baseline power of 100 W added per node. The green circle marks a possible optimal operating point.

line power prohibit the use of very small frequencies, even if energy to solution were the only relevant metric. On the other hand, energy is practically constant between 1.7 and 2.7 GHz when the best PPC value is chosen, but performance is boosted by 15% at 128 nodes.

Power capping

Up to now we have only considered energy or time to solution, which are certainly important factors for current and future high-performance systems and their users. However, designing data centers with minimum overhead for infrastructure is another crucial goal: Since the dynamic power of processors and systems may vary significantly, power supply and cooling must be planned to accommodate the "hottest" operating point, which is almost never reached. Sig-

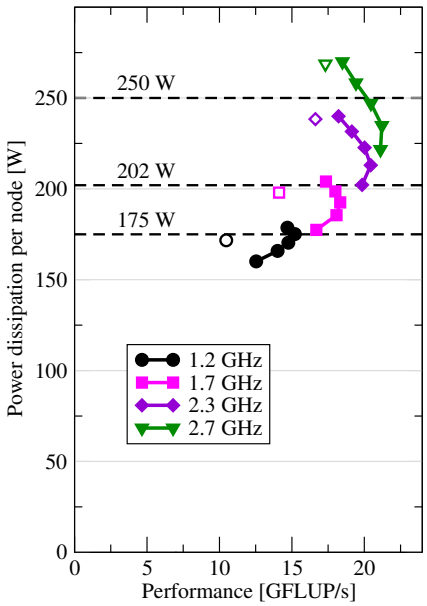


Figure 7.13: Power dissipation (per node, including full baseline power) of the AA pattern AVX LBM implementation (filled symbols) at 128 nodes across clock speeds and number of cores per chip (PPC=4...8 from top to bottom along each curve). For comparison the full-socket scalar implementation is also shown. Different power caps are indicated by horizontal lines.

nificant cost savings are possible if the power dissipation of a system is capped to a value that is acceptable for most of the applications, probably to the point where capping can have precedence over lowest energy to solution. For each particular application, the operating point should then be chosen so as to get best performance within the power cap.

Figure 7.13 shows the performance and power dissipation per node for a 128-node run of the AA pattern LBM implementation with different PPC values and clock speeds. A possible power cap of 202 W can be met by either running with seven cores at 1.2 GHz, with six cores at 1.7 GHz, or with four cores at 2.3 GHz; the latter corresponds to the suggested optimal operating point in Fig. 7.12. Comparing the options closest to the cap (1.7 and 2.3 GHz, respectively), the faster clock speed is clearly favorable since it provides about 8% more performance. However, using five cores instead of four will exceed the power cap by more than 10 W at a minor performance gain of less than 3%. The full-socket scalar run (open square) is also below the cap, albeit at an unacceptable performance loss.

Of course, such considerations depend very much on the particular value of the cap: At a cap of 250 W, for instance, there is almost no restriction and one can go for maximum performance (five cores at 2.3 GHz). At a very stringent cap of 175 W there is no choice but to run with seven cores at 1.2 GHz, which is certainly far from the optimum in terms of performance and energy. This kind of power capping would be too tight.

In conclusion, staying inside a power cap requires the same awareness of the power and performance properties of an application as optimizing for performance and/or energy, but can lead rule out or allow certain operating points which would or would not be chosen without power capping.

7.6 Conclusion

7.6.1 Summary of results

The scalar and AVX-vectorized single-core performance and intra-chip saturation of an LBM implementation with AA propagation pattern was successfully modeled using the ECM model, and compared to the popular "pull-split" propagation model. The superiority of the AA pattern in terms of performance and memory bandwidth saturation was demonstrated, and it was shown that "best possible" performance is achieved on the chip with properly AVX-vectorized code, meaning that bandwidth saturation is reached at the lowest number of cores.

The energy consumption of the LBM algorithm with AA propagation pattern was then modeled on the chip level for a range of clock frequencies. Together with the ECM model a coherent picture of the performance and power properties of the LBM algorithm on the chip was gained, and good qualitative agreement was achieved with measurements. A region of optimal operating points w.r.t. clock speed and number of cores was identified. The system's baseline power (power consumption of everything apart from the CPUs, i.e., memory, chip sets, network, disks, etc.) was taken into account and shown to have a damping influence on the differences in energy consumption (as predicted by the power model). Even then, potential energy savings of up to 50% could be achieved compared to a naive operating point with the inferior pull-split propagation model. Single-thread code performance and the selection of an optimal number of cores per chip (the latter depending on the former) were shown to have the largest influence on energy consumption.

In highly parallel LBM runs, a loss in parallel efficiency was observed when the CPU clock speed was reduced. This unexpected result could be explained by a strong dependence of effective inter-node and intra-node MPI communication bandwidth on the clock speed. The effective bandwidth also shows a strong negative correlation with the number of MPI processes per node. Hence, non-negligible MPI communication introduces a core-bound component into the performance characteristics of the LBM algorithm. As a consequence, minimal energy to solution in the highly parallel case depends even more strongly on the proper choice of the operating point, especially on the number of cores per chip (and thus the single-thread performance). Possible power-capping conditions may modify this decision, especially when they are very loose or very stringent. In any case will a simple non-reflective reduction of the clock speed reduce performance and consume more energy at the same time.

7.6.2 Reassessment in view of performance patterns

The LBM algorithm is traditionally assumed to be limited by *memory bandwidth saturation* on all processor architectures. This assumption was shown to be valid with the ILBDC code on the modern Sandy Bridge processor only when properly SIMD vectorized. With scalar code, or with a propagation pattern that inhibits vectorization because of the sparse lattice representation, memory bandwidth saturation could not be achieved and *limited instruction throughput* or *ineffective execution* applies. The fact that the scalability across cores is much better in this case is of no significance.

The ECM model and the power model were shown to be in line with measurements across a range of clock frequencies. Going to strong scaling across nodes, MPI overhead added a *code composition* pattern, which made the code partially core-bound, along with consequences for energy consumption. Just as in the case of the backprojection algorithm studied in Chapter 6, a combination of patters applies. However, these are not encountered in the same loop here but in different code parts (lattice updates vs. MPI communication).

Chapter 8

Conclusion

This chapter summarizes the main points of this treatise and gives an outlook to possible future research. Note that a precise account of all contributions can be found in Sect. 1.3.

8.1 Summary

This work demonstrates the use and usefulness of performance models embedded in an iterative, structured performance engineering process when assessing, predicting, and optimizing implementations of algorithms in computational science. Using the process, domain scientists can arrive at a well-defined notion of the meaning of "best performance" instead of blindly applying code changes in hope for performance improvements. One central idea of the process is that the *failure* of a model should be embraced as something that challenges assumptions and paves the way for new insights. Although the process was developed specifically for node-level performance engineering, its principles are universal.

After a brief coverage of computer architecture in Chapter 2, the principles of white-box performance modeling on the chip level were presented in Chapter 3. White-box performance modeling uses abstractions in different levels of sophistication to describe the interaction of software with hardware. The prime example for this approach is the well-known roofline model, which predicts the performance of loops by reducing the interaction to two possible bottlenecks: in-core performance and data transfer bandwidth. Hence, the model is *resource driven*, because it is the exhaustion of either one of those bottlenecks which determines the runtime of a loop. It is also the simplest model that assumes the notion of high performance computing as *computing at a bottleneck*. The ability of the roofline model to predict performance and to guide performance analysis and optimization were demonstrated using simple examples.

The roofline model builds on four critical assumptions (*bottleneck*, *overlap*, *saturation*, and *streaming*), which limit its applicability but also allow for a clear account of when and why the model will probably fail. The ECM model can be regarded as a refinement of the roofline model. It only keeps the streaming assumption and *predicts* the occurrence of saturation effects, overlapping of execution and data transfers, and the hitting of bottlenecks by taking the time contribution of data transfers throughout the memory hierarchy into account. It is the only approach to date which uses a simplified machine model for the prediction of the single-core performance and scaling properties of loop kernels on a multicore chip. Since not all details of a microarchitecture are known (or obtainable), the ECM model is not entirely predictive. It

rather provides a prediction interval, in which the measured performance should be found. A comparison of measurement and prediction then hints at possible refinements.

In Chapter 4 a phenomenological multicore power model was developed, which can be used to select the optimal operating point in terms of clock frequency and number of cores used for minimum energy to solution. This model is not an immediate part of the performance engineering process, but it can be expected that CPU cycles will soon not be the only cost function that scientific users on large-scale systems have to take into account. One of the main prerequisites of the model is that code performance is the paramount influencing variable for energy consumption; all other measures are subordinate (*code race to idle* principle). The model distinguishes *scalable* from *saturating* code. The latter hits a bottleneck when the number of cores is increased, while the former does not. They both show very different energy consumption behavior with respect to the parameters, which makes optimizing for energy in complex applications a challenge. For scalable code, the model predicts that energy is minimized when using all cores at an optimal frequency, which may or may not be actually available in the system. This frequency depends on the ratio of static to dynamic power consumption, and can thus be large when the system is "hot" (*clock race to idle* principle). There are some interesting consequences for system design in this limit: Depending on the static (or baseline) power, one can distinguish design space limits for "hot" and "cool" systems, which can be identified with Cray's "oxen and chickens." For saturating code, the situation is simpler, because the minimal energy strategy is to use as many cores as required for saturation, at the lowest possible frequency. Baseline and dynamic power do not play a role.

Chapter 5 described the design of the pattern-based structured node-level performance engineering process. The main goal of this approach is to take guesswork out of high-performance code development. Starting from a first version without performance patterns it was shown how to apply the process to the textbook example of an OpenMP-parallel three-dimensional Jacobi smoother. Already at this stage the process is useful enough to be applied to realistic problems, but a significant enhancement is added by *performance patterns*. A pattern is identified by its *signature*, a combination of observed performance behavior (e.g., intra-chip scalability or dependence of performance on problem size) and hardware performance metrics. A performance model can then be built for every loop in the code from the correctly identified pattern (or combination of patterns) and input from code analysis, hardware characteristics, and probably microbenchmarking. Hardware performance metrics are used for validating or disproving the model. If the model is valid, optimizations can be targeted, which may or may not change the applicable pattern(s). In any case, the model gives a prediction of the possible performance benefit. If the model is not valid, it must be adjusted, either by changes in the input data or by choosing an entirely different pattern.

Chapter 6 described the application of performance modeling and engineering to a computed tomography backprojection algorithm on current Intel x86 multicore processors. A first attempt with the roofline model predicted a strong bandwidth limitation, but measurements stayed far behind this expectation. A more thorough code inspection (after some "common sense" and low-level optimizations) revealed that the bandwidth limitation did not apply but that code execution in the core is the actual bottleneck. In view of this insight it could be shown that the CM performance model is able to describe the performance and scalability features of the algorithm, and that only one out of the our considered processor architectures is bandwidth-limited for this algorithm. Consequently, a popular optimization often applied to loop nests (spatial blocking) was proven to be relevant only on this architecture. The analysis also uncovered a major

problem with the AVX SIMD instruction set on Intel Sandy Bridge processors, which lacks a *gather* instruction. Hence, the benefit from AVX over SSE was not as large as anticipated. Future Intel designs will feature gather instructions and thus promise a significant performance improvement. Finally, using the optimized code on a standard two-way Intel Xeon server the clinical upper runtime limit of twenty seconds could be met without reverting to special-purpose hardware.

In Chapter 7, the scalar and AVX-vectorized single-core performance and intra-chip saturation of a lattice-Boltzmann (LBM) flow solver implementation with AA propagation pattern was successfully modeled using the ECM model, and compared to the popular "pull-split" propagation model. "Best possible" performance in terms of bottleneck exhaustion was achieved on the chip. The chip-level energy consumption of the best implementation was then modeled using the multicore power model for a range of clock frequencies. Combining the two models, a region of optimal operating points with respect to clock speed and number of active cores could be identified. The system's baseline power had the expected damping influence on the differences in energy consumption. Single-thread code performance and the selection of an optimal number of cores per chip were shown to have the largest influence on energy consumption. Extrapolating these results to highly parallel strong scaling runs yielded the interesting observation that a core-bound component was introduced by the MPI communication overhead, sharpening the identification of the optimal operating point. Hence, the guidelines developed on the chip level were not invalidated, but the opposite was the case: Saving energy with tolerable loss in performance leaves very little room for variation in the tunable parameters. Finally, power-capping measures often imposed by computing centers were discussed, and it was shown that these do not have much influence on the choice of the optimal operating point if they are not uselessly stringent. Not that, although these results were obtained specifically for an LBM algorithm, they are expected to be generally applicable to many bandwidth-bound scenarios.

8.2 Outlook

Many possible options exist for extending the concepts developed here to a broader context.

The ECM model does not accurately describe the performance characteristics of bandwidth-bound code near the saturation point, especially when there is hardly any overlap in the cache hierarchy. This discrepancy must be studied further, although it does not significantly change the general applicability of the model. One crucial prerequisite for the model is that latency effects can be ignored, which is certainly not true for all loop structures, sparse matrix-vector multiply being the most prominent example. It would be worthwhile extending the model toward latency-influenced data accesses, so that the corresponding performance penalties can be estimated. This will especially be interesting on platforms having less advanced out-of-order and latency-hiding mechanisms, such as the IBM Blue Gene/Q processor or the Intel Xeon Phi Coprocessor platform, where the streaming assumption is frequently invalid.

The multicore power model assumes perfect load balancing and a constant clock speed across all cores of a chip. In the light of upcoming processor generations, which have the ability to set core-individual clock frequencies, the model will have to be revised. Under unbalanced load the power dissipation of "idle" cores depends very much on the details of the programming model. The power model should be extended to accommodate this situation.

While it is generally applicable to any performance analysis and optimization effort in com-

putational science, the structured performance engineering process was formulated specifically for node-level issues on multicore processors. However, it was intentionally not specified which particular models should be used. Extending it to other setups such as accelerators or massively parallel machines is mainly a matter of identifying the relevant patterns in these cases, which might be complex. For instance, MPI communication overhead alone is prone to several typical performance issues, which are nowadays identified using tools, but which should be embedded in the performance engineering process. It is also to be expected that there will be a mixture of applicable patterns in complex codes *especially* at the large scale.

Bibliography

[1] G. Hager, J. Treibig, J. Habich and G. Wellein. *Exploring performance and power properties of modern multicore chips via simple machine models*. Accepted for publication in Concurrency and Computation: Practice and Experience . http://arxiv.org/abs/1208.2908

[2] G. Schubert, H. Fehske, G. Hager and G. Wellein. *Hybrid-parallel sparse matrix-vector multiplication with explicit communication overlap on current multicore-based systems*. Parallel Processing Letters **21**, (2011) 339–358.

[3] M. Kreutzer, G. Hager, G. Wellein, H. Fehske and A. R. Bishop. *A unified sparse matrix data format for modern processors with wide SIMD units* Submitted. http://arxiv.org/abs/1307.6209

[4] J. Treibig and G. Hager. *Introducing a performance model for bandwidth-limited loop kernels*. In: R. Wyrzykowski, J. Dongarra, K. Karczewski and J. Wasniewski (eds.), *Parallel Processing and Applied Mathematics*, vol. 6067 of *Lecture Notes in Computer Science* (Springer Berlin / Heidelberg). ISBN 978-3-642-14389-2, 615–624, (2010).

[5] J. Treibig, G. Hager and G. Wellein. *Performance patterns and hardware metrics on modern multicore processors: Best practices for performance engineering*. In: I. Caragiannis, M. Alexander, R. Badia, M. Cannataro, A. Costan, M. Danelutto, F. Desprez, B. Krammer, J. Sahuquillo, S. Scott and J. Weidendorfer (eds.), *Euro-Par 2012: Parallel Processing Workshops*, vol. 7640 of *Lecture Notes in Computer Science* (Springer Berlin Heidelberg). ISBN 978-3-642-36948-3, 451–460, (2013). http://dx.doi.org/10.1007/978-3-642-36949-0_50

[6] J. Treibig, G. Hager, H. G. Hofmann, J. Hornegger and G. Wellein. *Pushing the limits for medical image reconstruction on recent standard multicore processors*. Int. J. High Perform. Comp. Appl. **27(2)**, (2013) 162–177.

[7] M. Wittmann, G. Hager, T. Zeiser and G. Wellein. *An analysis of energy-optimized lattice-Boltzmann CFD simulations from the chip to the highly parallel level* Submitted. http://arxiv.org/abs/1304.7664

[8] M. Gen and R. Cheng. *Genetic Algorithms and Engineering Optimization* (John Wiley & Sons), 1999. ISBN 978-0471315315.

[9] M. Wittmann, G. Hager, J. Treibig and G. Wellein. *Leveraging shared caches for parallel temporal blocking of stencil codes on multicore processors and clusters*. Parallel Processing Letters **20(4)**, (2010) 359–376. http://dx.doi.org/10.1142/S0129626410000296

[10] A. Schäfer and D. Fey. *A predictive performance model for stencil codes on multicore cpus*. In: M. Daydé, O. Marques and K. Nakajima (eds.), *High Performance Computing for Computational Science - VECPAR 2012*, vol. 7851 of *Lecture Notes in Computer Science* (Springer Berlin Heidelberg). ISBN 978-3-642-38717-3, 451–466, (2013).

[11] S. W. Williams, A. Waterman and D. A. Patterson. *Roofline: An insightful visual performance model for floating-point programs and multicore architectures*. Tech. Rep. UCB/EECS-2008-134, EECS Department, University of California, Berkeley, Oct 2008. http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-134.html

[12] R. W. Hockney and I. J. Curington. *$f_{1/2}$: A parameter to characterize memory and communication bottlenecks*. Parallel Computing **10(3)**, (1989) 277–286.

[13] W. Schönauer. *Scientific Supercomputing: Architecture and Use of Shared and Distributed Memory Parallel Computers* (Self-edition), 2000. http://www.rz.uni-karlsruhe.de/~rx03/book

[14] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman and M. Gittings. *Predictive performance and scalability modeling of a large-scale application*. In: *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '01 (ACM, New York, NY, USA). ISBN 1-58113-293-X, 37–37, (2001).

[15] F. Petrini, D. J. Kerbyson and S. Pakin. *The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q*. In: *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing* (IEEE Computer Society, Washington, DC, USA). ISBN 1-58113-695-1, 55, (2003).

[16] P. F. Spinnato, G. van Albada and P. M. Sloot. *Performance modeling of distributed hybrid architectures*. IEEE Trans. Parallel Distrib. Systems **15(1)**, (2004) 81–92.

[17] D. J. Kerbyson and P. W. Jones. *A performance model of the Parallel Ocean Program*. Int. J. High Perform. Comp. Appl. **19**, (2005) 261–276.

[18] S. Hammond, G. Mudalige, J. Smith, S. Jarvis, J. Herdman and A. Vadgama. *WARPP: A toolkit for simulating high performance parallel scientific codes*. In: *2nd International Conference on Simulation Tools and Techniques (SIMUTools09)*. (2009). http://eprints.dcs.warwick.ac.uk/38/

[19] M. Burtscher, B.-D. Kim, J. Diamond, J. McCalpin, L. Koesterke and J. Browne. *PerfExpert: An easy-to-use performance diagnosis tool for HPC applications*. In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10 (IEEE Computer Society, Washington, DC, USA). ISBN 978-1-4244-7559-9, 1–11, (2010).

[20] D. Schmidl, C. Iwainsky, C. Terboven, C. H. Bischof and M. S. Müller. *Towards a performance engineering workflow for OpenMP 4.0*. In: *Proc. International Conference on Parallel Computing (ParCo 2013)*. Accepted.

[21] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers* (CRC Press, Inc., Boca Raton, FL, USA), 1st ed., 2010. ISBN 978-1439811924.

[22] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach* (Morgan Kaufmann), 4th ed., 2006. ISBN 978-0123704900.

[23] *Intel 64 and IA-32 architectures optimization reference manual*, April 2012. http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf.

[24] J. D. McCalpin. *STREAM: Sustainable memory bandwidth in high performance computers*. Tech. rep., University of Virginia, Charlottesville, VA, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/

[25] J. Treibig, G. Hager and G. Wellein. *LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments*. In: *PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures* (IEEE Computer Society, Los Alamitos, CA, USA), 207–216, (2010).

[26] J. Treibig. *Likwid: Linux tools to support programmers in developing high performance multi-threaded programs*. http://code.google.com/p/likwid/

[27] G. M. Amdahl. *Validity of the single processor approach to achieving large scale computing capabilities*. In: *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, Spring Joint Computer Conference* (ACM, New York, NY, USA), 483–485, (1967).

[28] G. Wellein, G. Hager, A. Basermann and H. Fehske. *Exact Diagonalization of Large Sparse Matrices: A Challenge for Modern Supercomputers*. In: *CD CUG Summit 2001, Indian Wells, USA*. (2001).

[29] G. Wellein, G. Hager, A. Basermann and H. Fehske. *Fast sparse matrix-vector multiplication for TeraFlop/s computers*. In: J. Palma *et al.* (eds.), *High Performance Computing for Computational Science — VECPAR2002, LNCS 2565* (Springer-Verlag, Berlin, Heidelberg). ISBN 3-540-00852-7, 287–301, (2003).

[30] G. Schubert, G. Hager and H. Fehske. *Performance limitations for sparse matrix-vector multiplications on current multicore environments*. In: S. Wagner *et al.* (eds.), *High Performance Computing in Science and Engineering, Garching/Munich 2009* (Springer-Verlag, Berlin, Heidelberg), (2010). To appear. http://arxiv.org/abs/0910.4836

[31] S. Williams, L. Oliker, R. W. Vuduc, J. Shalf, K. A. Yelick and J. Demmel. *Optimization of sparse matrix-vector multiplication on emerging multicore platforms*. Parallel Computing **35(3)**, (2009) 178–194.

[32] M. Mohiyuddin, M. Hoemmen, J. Demmel and K. Yelick. *Minimizing communication in sparse matrix solvers*. In: *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (ACM, New York, NY, USA). ISBN 978-1-60558-744-8, 1–12, (2009).

[33] N. Bell and M. Garland. *Implementing sparse matrix-vector multiplication on throughput-oriented processors*. In: *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (ACM, New York, NY, USA). ISBN 978-1-60558-744-8, 1–11, (2009).

[34] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A. Basermann and A. Bishop. *Sparse matrix-vector multiplication on GPGPU clusters: A new storage format and a scalable implementation*. In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1696–1702, (2012).

[35] *pOSKI: parallel optimized sparse kernel interface*. http://bebop.cs.berkeley.edu/poski

[36] E. Cuthill and J. McKee. *Reducing the bandwidth of sparse symmetric matrices*. In: *Proceedings of the 1969 24th national conference (ACM '69), ACM, New York, NY, USA*. 157–172, (1969).

[37] *The EPCC OpenMP Microbenchmarks*. http://www2.epcc.ed.ac.uk/computing/research_activities/openmpbench/openmp_index.html

[38] *Intel architecture code analyzer*. http://software.intel.com/en-us/articles/intel-architecture-code-analyzer.

[39] M. A. Suleman, M. K. Qureshi and Y. N. Patt. *Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs*. SIGARCH Comput. Archit. News **36(1)**, (2008) 277–286. ISSN 0163-5964.

[40] J. D. McCalpin. *Memory bandwidth and machine balance in current high performance computers*. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, December 1995. http://tab.computer.org/tcca/NEWS/DEC95/dec95_mccalpin.ps

[41] E. Rotem, A. Naveh, A. Ananthakrishnan, D. Rajwan and E. Weissmann. *Power-management architecture of the Intel microarchitecture code-named Sandy Bridge*. IEEE Micro **32**, (2012) 20–27. ISSN 0272-1732.

[42] R. Schöne, D. Hackenberg and D. Molka. *Memory performance at reduced CPU clock speeds: An analysis of current x86_64 processors*. In: *Proceedings of the 2012 USENIX conference on Power-Aware Computing and Systems*, HotPower'12 (USENIX Association, Berkeley, CA, USA), 9–9, (2012). http://dl.acm.org/citation.cfm?id=2387869.2387878

[43] J. W. Choi, D. Bedard, R. Fowler and R. Vuduc. *A roofline model of energy*. In: *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. ISSN 1530-2075, 661–672, (2013).

[44] D. Li, B. R. de Supinski, M. Schulz, D. S. Nikolopoulos and K. W. Cameron. *Strategies for energy efficient resource management of hybrid programming models*. IEEE Transactions on Parallel and Distributed Systems **99(PrePrints)**. ISSN 1045-9219.

[45] C. Navarrete, C. Guillen, W. Hesse and M. Brehm. *Optimizing the energy-to-solution on SandyBridge systems*. inSiDE – Innovatives Supercomputing in Deutschland **10(2)**, (2012) 62–65. http://www.autotune-project.eu/sites/default/files/Materials/Papers/inSiDE_autumn2012.pdf

[46] S. Donath. *On Optimized Implementations of the Lattice-Boltzmann Method on Contemporary High Performance Architectures*. Bachelor thesis, Universität Erlangen-Nürnberg, Department Informatik, 2004.

[47] G. Wellein, T. Zeiser, S. Donath and G. Hager. *On the Single Processor Performance of Simple Lattice Boltzmann Kernels*. Comput. & Fluids **35**, (2006) 910–919.

[48] I. Steiner. Intel, private communication.

[49] D. Wonnacott. *Using time skewing to eliminate idle time due to memory bandwidth and network limitations*. In: *Proc. 14th International Parallel and Distributed Processing Symposium (IPDPS 2000)*. 171–180, (2000).

[50] M. Frigo and V. Strumpen. *Cache oblivious stencil computations*. In: *ICS '05: Proceedings of the 19th annual international conference on Supercomputing* (ACM, New York, NY, USA). ISBN 1-59593-167-8, 361–366, (2005).

[51] M. Frigo and V. Strumpen. *The memory behavior of cache oblivious stencil computations*. J. Supercomput. **39(2)**, (2007) 93–112. ISSN 0920-8542.

[52] 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**, (2009) 129–159.

[53] G. Wellein, G. Hager, T. Zeiser, M. Wittmann and H. Fehske. *Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization*. Annual International Computer Software and Applications Conference (COMPSAC09) **1**, (2009) 579–586. ISSN 0730-3157.

[54] J. Treibig, G. Wellein and G. Hager. *Efficient multicore-aware parallelization strategies for iterative stencil computations*. Journal of Computational Science **2(2)**, (2011) 130–137. ISSN 1877-7503. Simulation Software for Supercomputers. http://www.sciencedirect.com/science/article/pii/S1877750311000172

[55] R. de la Cruz and M. Araya-Polo. *Towards a multi-level cache performance model for 3D stencil computation*. Procedia Computer Science **4(0)**, (2011) 2146 – 2155. Proceedings of the International Conference on Computational Science, ICCS 2011. http://www.sciencedirect.com/science/article/pii/S1877050911002936

[56] J. Jeffers and J. Reinders. *Intel Xeon Phi Coprocessor High Performance Programming* (Morgan Kaufmann), 2013. ISBN 978-0124104143.

[57] W. Gropp and M. Snir. *Programming for exascale computers*. Computing in Science Engineering **PP(99)**, (2013) 1–1. ISSN 1521-9615.

[58] A. Kak and M. Slaney. *Principles of Computerized Tomographic Imaging* (SIAM), 2001.

[59] B. Heigl and M. Kowarschik. *High-speed reconstruction for C-arm computed tomography*. In: *9th International Meeting on Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine* (www.fully3d.org, Lindau), 25–28, (2007).

[60] N. Strobel and et al. *3D Imaging with Flat-Detector C-Arm Systems*. In: *Multislice CT* (Springer, Berlin / Heidelberg), 3rd ed. ISBN 978-3-540-33125-4, 33–51, (2009).

[61] L. Feldkamp, L. Davis and J. Kress. *Practical Cone-Beam Algorithm*. Journal of the Optical Society of America **A1(6)**, (1984) 612–619.

[62] K. Mueller and R. Yagel. *Rapid 3D cone-beam reconstruction with the Algebraic Reconstruction Technique (ART) by utilizing texture mapping graphics hardware*. Nuclear Science Symposium, 1998. Conference Record. **3**, (1998) 1552–1559.

[63] K. Mueller, F. Xu and N. Neophytou. *Why do Commodity Graphics Hardware Boards (GPUs) work so well for acceleration of Computed Tomography?* In: *SPIE Electronic Imaging Conference*, vol. 6498 (San Diego), 64980N.1–64980N.12, (2007).

[64] H. Scherl, B. Keck, M. Kowarschik and J. Hornegger. *Fast GPU-Based CT Reconstruction using the Common Unified Device Architecture (CUDA)*. In: E. C. Frey (ed.), *Nuclear Science Symposium Conference Record, 2007. NSS '07. IEEE*, vol. 6 (Honolulu, HI). ISSN 1082-3654, 4464–4466, (2007).

[65] Okitsu, Ino and Hagihara. *High-performance cone beam reconstruction using cuda compatible gpus*. Par. Comp. **36**, (2010) 129–141.

[66] E. Papenhausen, Z. Zheng and K. Mueller. *GPU-Accelerated Back-Projection Revisited: Squeezing Performance by Careful Tuning*. Workshop on High Performance Image Reconstruction (HPIR), 2011.

[67] H. Scherl, M. Koerner, H. Hofmann, W. Eckert, M. Kowarschik and J. Hornegger. *Implementation of the FDK algorithm for cone-beam CT on the cell broadband engine architecture*. In: J. Hsieh and M. J. Flynn (eds.), *SPIE Medical Imaging Conference Proc.*, vol. 6510 (SPIE), 651058, (2007).

[68] M. Kachelrieß, M. Knaup and O. Bockenbach. *Hyperfast parallel-beam and cone-beam backprojection using the CELL general purpose hardware*. Medical Physics **34(4)**, (2007) 1474–1486.

[69] C. Rohkohl, B. Keck, H. G. Hofmann and J. Hornegger. *RabbitCT—An Open Platform for Benchmarking 3-D Cone-beam Reconstruction Algorithms*. Medical Physics **36(9)**, (2009) 3940–3944. http://link.aip.org/link/?MPH/36/3940/1

[70] *RabbitCT Benchmark*. http://www.rabbitct.com/.

[71] H. G. Hofmann, B. Keck, C. Rohkohl and J. Hornegger. *Comparing Performance of Many-core CPUs and GPUs for Static and Motion Compensated Reconstruction of C-arm CT Data*. Medical Physics **38(1)**, (2011) 3940–3944.

[72] *The Stream Benchmark*. http://www.streambench.org/, Mar 2011.

[73] J. Treibig, G. Hager and G. Wellein. *LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments*. 2012 41st International Conference on Parallel Processing Workshops **0**, (2010) 207–216. ISSN 1530-2016.

[74] *LIKWID performance tools*. http://code.google.com/p/likwid

[75] N. Navab, A. Bani-Hashemi, M. Nadar, K. Wiesent, P. Durlak, T. Brunner, K. Barth and R. Graumann. *3D Reconstruction from Projection Matrices in a C-Arm Based 3D-Angiography System*. In: W. Wells, A. Colchester and S. Delp (eds.), *Medical Image Computing and Computer-Assisted Interventation MICCAI 98*, vol. 1496 of *Lecture Notes in Computer Science* (Springer Berlin / Heidelberg). ISBN 978-3-540-65136-9, 119–129, (1998). 10.1007/BFb0056194. http://dx.doi.org/10.1007/BFb0056194

[76] K. Wiesent, K. Barth, N. Navab, P. Durlak, T. Brunner, O. Schuetz and W. Seissler. *Enhanced 3-D-reconstruction algorithm for C-arm systems suitable for interventional procedures*. IEEE Transactions on Medical Imaging **19(5)**, (2000) 391–403.

[77] R. Hartley and A. Zissermann. *Multiple View Geometry in Computer Vision, 2nd Edition* (Cambridge University Press, Cambridge), 2004.

[78] I. Goddard, A. Berman, O. Bockenbach, F. Lauginiger, S. Schuberth and S. Thieret. *Evolution of computer technology for fast cone beam backprojection*. In: *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, vol. 6498. (2007).

[79] H. G. Hofmann, B. Keck and J. Hornegger. *Accelerated C-arm Reconstruction by Out-of-Projection Prediction*. In: T. M. Deserno, H. Handels, H.-P. Meinzer and T. Tolxdorff (eds.), *Bildverarbeitung für die Medizin 2010* (Berlin). ISBN 978-3-642-11967-5, 380–384, (2010). http://www5.informatik.uni-erlangen.de/Forschung/Publikationen/2010/Hofmann10-ACR.pdf.

[80] J. Hofmann. *Performance Evaluation of the Intel Many Integrated Core Architecture for 3D Image Reconstruction in Computed Tomography*. Master's thesis, Universität Erlangen-Nürnberg, Department Informatik, 2013.

[81] J. Treibig, G. Hager and G. Wellein. *Complexities of performance prediction for bandwidth-limited loop kernels on multi-core architectures*. In: S. W. et al. (ed.), *High Performance Computing in Science and Engineering, Garching/Munich 2009* (Springer, Berlin / Heidelberg, Garching/Munich). ISBN 978-3642138713, 3–12, (2010).

[82] M. Schulz, M. Krafczyk, J. Tölke and E. Rank. *Parallelization strategies and efficiency of CFD computations in complex geometries using lattice Boltzmann methods on high performance computers*. In: M. Breuer, F. Durst and C. Zenger (eds.), *High Performance Scientific and Engineering Computing Proceedings of the 3rd International FORTWIHR Conference on HPSEC, Erlangen, March 12-14, 2001*, vol. 21 of *Lecture Notes in Computational Science and Engineering* (Springer-Verlag, Berlin, Heidelberg), 115–122, (2002).

[83] C. Pan, J. F. Prins and C. T. Miller. *A high-performance lattice Boltzmann implementation to model flow in porous media*. Computer Physics Communications **158(2)**, (2004) 89–105.

[84] T. Pohl, F. Deserno, N. Thürey, U. Rüde, P. Lammers, G. Wellein and T. Zeiser. *Performance evaluation of parallel large-scale lattice Boltzmann applications on three supercomputing architectures*. In: *SC '04: Proceedings of*

the 2004 ACM/IEEE conference on Supercomputing. (2004). http://www.sc-conference.org/sc2004/schedule/index.php?module=Default&action=ShowDetail&eventid=13#2.

[85] T. Zeiser, G. Wellein and P. Lammers. *Is there still a need for tailored HPC systems or can we go with commodity off-the-shelf clusters — Some comments based on performance measurements using a lattice Boltzmann flow solver*. InSiDE – Innovatives Supercomputing in Deutschland **2(2)**, (2004) 10–15.

[86] J. Wang, X. Zhang, A. G. Bengough and J. W. Crawford. *Domain-decomposition method for parallel lattice Boltzmann simulation of incompressible flow in porous media*. Phys. Rev. E **72(1)**, (2005) 016706.

[87] M. Bernaschi, S. Succi, M. Fyta, E. Kaxiras, S. Melchionna and J. Sircar. *MUPHY: A parallel high performance MUlti PHYsics/Scale code*. In: *IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008.* 1–8, (2008).

[88] K. Mattila, J. Hyvaluoma, J. Timonen and T. Rossi. *Comparison of implementations of the lattice-Boltzmann method*. Computers & Mathematics with Applications **55(7)**, (2008) 1514–1524.

[89] T. Zeiser. *Simulation und Analyse von durchströmten Kugelschüttungen in engen Rohren unter Verwendung von Hochleistungsrechnern*. Ph.D. thesis, Universität Erlangen-Nürnberg, Technische Fakultät, 2008.

[90] P. Bailey, J. Myre, S. Walsh, D. Lilja and M. Saar. *Accelerating lattice Boltzmann fluid flow simulations using graphics processors*. In: *International Conference on Parallel Processing 2009 (ICPP'09)*. 550–557, (2009).

[91] D. Vidal, R. Roy and F. Bertrand. *On improving the performance of large parallel lattice Boltzmann flow simulations in heterogeneous porous media*. Computers & Fluids **39(2)**, (2010) 324–337.

[92] J. Zudrop, H. Klimach, M. Hasert, K. Masilamani and S. Roller. *A fully distributed CFD framework for massively parallel systems*. In: *Cray Users Group Conference 2011*. (2012). April 29 to May 3, Stuttgart, Germany. https://cug.org/proceedings/attendee_program_cug2012/includes/files/pap136.pdf

[93] M. Wittmann, T. Zeiser, G. Hager and G. Wellein. *Comparison of different propagation steps for lattice Boltzmann methods*. Computers & Mathematics with Applications **65(6)**, (2013) 924–935.

[94] A. Peters, S. Melchionna, E. Kaxiras, J. Lätt, J. K. Sircar, M. Bernaschi, M. Bisson and S. Succi. *Multiscale simulation of cardiovascular flows on the IBM Bluegene/P: Full heart-circulation system at red-blood cell resolution*. In: *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking and Storage, SC 2010, New Orleans, LA, USA, November 13-19, 2010* (IEEE), 1–10, (2010).

[95] J. Carter, M. Soe, L. Oliker, Y. Tsuda, G.Vahala, L. Vahala and A. Macnab. *Magneto-hydrodynamic turbulence simulations on the earth simulator using the lattice Boltzmann*

method. In: *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking and Storage (SC05), Seattle, WA, November 12-18, 2005*. (2005).

[96] S. Williams, J. Carter, L. Oliker, J. Shalf and K. Yelick. *Optimization of a lattice Boltzmann computation on state-of-the-art multicore platforms*. J. Parallel Distrib. Comput. **69(9)**, (2009) 762–777.

[97] T. Zeiser, G. Hager and G. Wellein. *Benchmark analysis and application results for lattice Boltzmann simulations on NEC SX vector and Intel Nehalem systems*. Parallel Processing Letters **19(4)**, (2009) 491–511.

[98] P. Bhatnagar, E. P. Gross and M. K. Krook. *A model for collision processes in gases. I. small amplitude processes in charged and neutral one-component systems*. Phys. Rev. **94(3)**, (1954) 511525.

[99] S. Succi. *The Lattice Boltzmann Equation – For Fluid Dynamics and Beyond* (Clarendon Press), 2001.

[100] D. Wolf-Gladrow. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models*, vol. 1725 of *Lecture Notes in Mathematics* (Springer, Berlin), 2000.

[101] S. Chen and G. Doolen. *Lattice Boltzmann Method for Fluid Flows*. Annu. Rev. Fluid Mech. **30**, (1998) 329–364.

[102] Y. Qian, D. d'Humières and P. Lallemand. *Lattice BGK Models for Navier-Stokes Equation*. Europhys. Lett. **17(6)**, (1992) 479–484.

[103] H. Huber. LRZ, private communication.

[104] M. Wittmann, T. Zeiser, G. Hager and G. Wellein. *Domain decomposition and locality optimization for large-scale lattice Boltzmann simulations*. Computers & Fluids **80**, (2013) 283–289. ISSN 0045-7930. Selected contributions of the 23rd International Conference on Parallel Fluid Dynamics ParCFD2011.

[105] *Intel MPI benchmarks*. http://software.intel.com/en-us/articles/intel-mpi-benchmarks/

Index

80/20 rule, 9, 10

AA pattern, 112
acceleration, 28
adjacency list, 112
Amdahl's Law, 29, 83
applicable peak
  bandwidth, 31
  performance, 30
AVX, 23, 55

backprojection, 13, 91
baseline power, 56, 58, 61, 62, 64, 65, 121
binding, 21
black-box modeling, 10
Boltzmann equation, 110
bottleneck assumption, 32

C-arm CT, 91
cache, 21
  coherence, 22
  line, 21
    eviction, 22
  miss, 21
ccNUMA, 23
  page interleaving, 77
  page placement, 76, 83
chickens, 64
chunk occupancy, 38
clock speed, 21
code
  balance, 30, 39, 41, 55, 70, 73, 75
  optimization, 69
code composition, 84
collision
  integral, 110
  operator, 111
  step, 112
communication
  overhead, 84
computational intensity, 30, 33, 50, 60
compute node, 22
computed tomography, 91
  seeCT, 13
control flow, 81
CRS format, 36
CT, 13, 91

DCT, *see* dynamic concurrency throttling, 110
DGEMM, 55
DVFS, 59, 110
dynamic concurrency throttling, 60, 63
dynamic power, 56, 59, 60, 62, 64

ECM model, 11, 43–51, 86, 87
  for backprojection, 101
  for LBM, 115–118
  for vector triad, 46
ELLPACK, 37
energy
  to solution, 59–63
erratic access, 82
event signatures, 79, 80
exclusive cache, 21
execution units, 17
expensive instructions, 84

false sharing, 22, 82
first-touch, 23
first-touch principle, 77, 83, 104
Flop, 28
fluid lattice site update, *see* FLUP
FLUP, 112
FPGA, 13, 91

Golden Rule, 23, 77
GPGPU, 13, 44, 112
Gustafson's Law, 29

hardware metrics, 69
hardware performance monitoring, *see* HPM
Harpertown, 93, 103, 104, 106
hazards, 85
HPM, 24, 33, 78
HyperTransport, 23

IACA, 101, 103, 115
ILBDC, 110, 111
ILP, 18, 103
inclusive cache, 21, 43
ineffective instructions, 85
instruction
    cache, 20
    overhead, 84
    throughput, 81
instruction-level parallelism, *see* ILP
Intel MPI benchmark suite, 123

Jacobi smoother, 55, 70

lattice site update, *see* LUP
lattice-Boltzmann
    equation, 110
lattice-Boltzmann method, *see* LBM
layer condition, 55, 73
LBM, 13, 70, 109–127
leakage power, 56
light speed, 10
LIKWID, 24, 54, 76, 79
line update kernel, 98
load imbalance, 83
locality domain, 23
LUP, 70

machine model, 11
    cycle-accurate, 11
marker-and-cell, 111, 114
memory
    bandwidth, 22, 33, 35
    interface, 22, 35
    latency, 22
micro-ops, 20, 47
microarchitecture
    anomalies, 82
microbenchmarking, 31, 69
Moore's Law, 21
multi-stream benchmark, 116

non-temporal stores, 50, 74, 76, 112
`numactl`, 77

OLC, 21, 81
OpenMP, 40, 70
    load balancing, 103
    overhead, 29, 42
out-of-order execution, 18
outer-level cache, *see* OLC
overlap assumption, 32
oxen, 64

parallel first touch, 77, 86
particle distribution function, *see* PDF
PDF, 110, 111
performance, 27
    accelerated, 27
    engineering, 12, 33, 67, 87
    model, 9, 69
    patterns, 12, 78–87
    profile, 10
pipeline, 17
    bubbles, 18, 20
    depth, 17
    hazard, 81
    saturation, 81
    throughput, 18
power
    capping, 125
    gating, 56
    model, 11, 59
        for LBM, 118–121
prefetching, 22, 43, 82
profiling, 68
pull scheme, 112
pull-split, 112, 121

QuickPath, 23

RabbitCT, 91
race to idle, 121
    clock ~, 61–64
    code ~, 62, 121
RAPL, 124
raytracer, 54

RCM algorithm, 38
register, 19
relaxation time, 110
roofline model, 30–43, 50, 118
    assumptions, 32
    of energy, 59

Sandy Bridge, 15, 23, 34, 40, 44–46, 49, 53, 56, 57, 72, 92, 102, 113, 127
saturation assumption, 32
saturation point, 45, 49, 50
Schönauer triad, 34
    ECM model, 46
    multicore scaling, 49
SELL-C-σ, 37
sendrecv benchmark, 123
shot-in-the-dark optimizations, 12
SIMD, 13, 19, 33, 60, 85, 111
    intrinsics, 55, 98
    width, 19, 97
single instruction multiple data, *see* SIMD
slow computing, 85
SMT, 20, 24, 50, 93, 103, 105
SoA, 112
sparse lattice, 112
sparse matrix
    storage schemes, 36
    vector multiply, 13, 36, 69, 86
spatial blocking, 74
speedup, 27
SSE, 19, 41
stencil, 70, 77
store miss, 70
STREAM, 45, 72, 93
streaming step, 112
streaming assumption, 32
strided access, 82
strong scaling, 14, 29
structure of arrays, *see* SoA
SuperMUC, 65, 109, 113, 121–123
superscalarity, 18
synchronization overhead, 83

TDP, 21, 56, 120
temporal blocking, 50
thermal design power, *see* TDP
topology, 21
TRT, 13, 111
turbo mode, 21, 25, 54, 75, 76, 116, 119–122
two-relaxation-time, *see* TRT

vector triad, *see* Schönauer triad
vectorization, 19

wall-clock time, 27
weak scaling, 29
Westmere, 23, 41, 101
white-box modeling, 10
work, 27
write-allocate, 21, 35, 44, 47, 70, 72, 74, 76, 112
write-combine buffers, 74

Xeon Phi, 18

Z-plot, 119

Curriculum Vitae

Persönliche Daten

Name	Georg Hager
Geboren am	21. August 1970
in	Hof an der Saale
Staatsangehörigkeit	Deutsch
Adresse	Danteweg 16 90427 Nürnberg
Telefon	0911/3008663
E-Mail	georg.hager@fau.de
Familienstand	verheiratet, zwei Kinder

Ausbildung

1976–1980	Grundschule Hof-Moschendorf
1980–1989	Schiller-Gymnasium Hof
Mai 1989	Allgemeine Hochschulreife (Abitur)
Juni 1989 – August 1990	Grundwehrdienst
WS 1990 – SS 1996	Studium der Physik mit Ziel Diplom an der Universität Bayreuth
Oktober 1993 – März 1994	Auslandsstudium (ERASMUS-Stipendium) an der University of St Andrews, Schottland
13. Mai 1996	Diplom in Physik an der Universität Bayreuth Thema der Diplomarbeit: „Quasiperiodische Lösungen der komplexen eindimensionalen Ginzburg-Landau-Gleichung“ Betreuer: Prof. Dr. Lorenz Kramer
24. Oktober 2005	Promotion an der Ernst-Moritz-Arndt-Universität Greifswald Titel der Dissertation: „A parallelized density matrix renormalization group algorithm and its application to strongly correlated quantum systems“ Betreuer: Prof. Dr. Holger Fehske URN:urn:nbn:de:gbv:9-000024-1

Beruflicher Werdegang

Mai 1996 – Mai 1999	Stipendiat des Graduiertenkollegs „Physik der starken Wechselwirkung“ an der Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

Mai 1999 – März 2000	Wissenschaftliche Hilfskraft am Institut für Theoretische Physik III der FAU
April 2000 – heute	Wissenschaftlicher Mitarbeiter in der HPC-Gruppe des Regionalen Rechenzentrums Erlangen (RRZE) der FAU
2000–2001	Mitarbeiter im KONWIHR-Projekt „cxHPC“ (Center of Excellence for High Performance Computing), Projektleiter: Dr. Gerhard Wellein
2002–2004	Mitarbeiter im KONWIHR-Projekt „HQS@HPC“ (hochkorrelierte Quantensysteme auf Hochleistungsrechnern), Projektleiter: Prof. Dr. Holger Fehske
Oktober 2002 – heute	Lehrbeauftragter an der Technischen Hochschule (früher Fachhochschule) Nürnberg
Mai 2010	Ernennung zum Akademischen Rat und Übernahme in das Beamtenverhältnis auf Probe
Dezember 2011	Übernahme in das Beamtenverhältnis auf Lebenszeit

Eingeworbene Drittmittel

2013–2015	219200 € als PI im Projekt „ESSEX“ (Equipping Sparse Solvers for Exascale) des DFG-Schwerpunktprogrammes 1648 (SPPEXA)
2012	25000 € für das KONWIHR-Projekt „SparseLib“
2009	50000 € für das KONWIHR-Projekt „HQS@HPC-II“

Lehrtätigkeit

Lehre an Hochschulen

WS 2012/13	*Performance-Optimierung und -Modellierung auf modernern Rechnerarchitekturen* Vorlesung und Übung im Institut für Physik der Ernst-Moritz-Arndt-Universität Greifswald
WS 2011/12	*Parallelprogrammierung auf Hochleistungsrechnern* Vorlesung und Übung im Institut für Physik der Ernst-Moritz-Arndt-Universität Greifswald
SS 2000 – heute	*Programming Techniques for Supercomputers* Vorlesung und Übung (zusammen mit Prof. G. Wellein) im Studiengang Computational Engineering an der Technischen Fakultät der FAU
WS 2009 – heute	*Efficient numerical simulation on multi- and manycore processors* Seminar (zusammen mit Mitarbeitern der HPC-Gruppe des RRZE) im Studiengang Computational Engineering an der Technischen Fakultät der FAU
WS 2010 – heute	*Elementary Numerical Mathematics* Leitung und Durchführung der Übungen für die Vorlesung im Studiengang Computational Engineering an der Technischen Fakultät der FAU
WS 2007 – heute	*Parallele Programmierung* Blockvorlesung mit Übungen (zusammen mit Prof. G. Wellein) an der Fakultät für Elektrotechnik, Feinwerktechnik und Informationstechnik (EFI) der TH Nürnberg
SS 2011 – heute	*Parallele Programmierung von Multicore-Systemen* Blockvorlesung mit Übungen im Studiengang Informatik und Wirtschaftsinformatik der Fakultät für Informatik an der TH Nürnberg
März 2011	*Efficient multithreaded programming on modern CPUs and GPUs* Blockvorlesung mit Übungen an der Königlichen Technischen Hochschule (KTH) in Stockholm, Schweden

WS 2002 – SS 2006	*Programmieren 1 & 2 (C/C++)* Vorlesungen mit Übungen im Studiengang Informatik und Wirtschaftsinformatik der Fakultät für Informatik an der Ohm-Hochschule Nürnberg

Betreute Arbeiten

WS 2013	J. Bleisteiner: *Porting and optimizing a lattice-Boltzmann algorithm for the Intel Xeon Phi accelerator.* Masterarbeit im Fach Computational Engineering an der FAU Erlangen-Nürnberg
WS 2012	T. Scharpff: *Analyse und Optimierung von Operationen auf dünn besetzten Matrizen.* Studienarbeit im Fach Informatik an der FAU Erlangen-Nürnberg
WS 2011	K. Sembritzki: *Evaluation of the Coarray Fortran Programming Model on the Example of a Lattice Boltzmann Code.* Masterarbeit im Fach Informatik an der FAU Erlangen-Nürnberg
SS 2010 – WS 2010	J. Daschke, T. Gohla, S. Heidingsfelder: *Erstellung eines Datenbanksystems zur Verwaltung wissenschaftlicher Publikationen.* IT-Masterprojekt an der Ohm-Hochschule Nürnberg
WS 2009	H. Stengel: *Paralleles Programmieren auf hybrider Hardware: Modelle und Anwendungen.* Masterarbeit im Fach Informatik an der Ohm-Hochschule Nürnberg
WS 2008	M. Wittmann: *Potentials of temporal blocking for stencil-based computations on multi-core systems.* Masterarbeit im Fach Informatik an der Ohm-Hochschule Nürnberg
SS 2008 – WS 2008	M. Wittmann, H. Stengel, O. Narr: *RRZE Accounting- und Kontingentreports, Teil II.* IT-Masterprojekt an der Ohm-Hochschule Nürnberg

SS 2007	H. Stengel: *C++-Programmiertechniken f”ur High Performance Computing auf Systemen mit nichteinheitlichem Speicherzugriff unter Verwendung von OpenMP.* Diplomarbeit im Fach Informatik an der Ohm-Hochschule Nürnberg
SS 2006	M. Wittmann, H. Stengel, F. Waldheim, M. Schloyer, S. Witter: *RRZE Accounting- und Kontingentreports.* IT-Studentenprojekt an der Ohm-Hochschule Nürnberg
WS 2005	H. Stengel: *Erstellung einer Benchmarksuite für Anwendungen im Hochleistungsrechnen.* Praktisches Studiensemester am RRZE für die Ohm-Hochschule Nürnberg

Kurse, Workshops, Tutorials

Dezember 2013	*Node-level performance engineering* Zweitägiger Kurs im Rahmen des „PRACE Advanced Training Centre“ (zusammen mit Prof. G. Wellein) am LRZ Garching
November 2013	*The practitioner’s cookbook for good parallel performance on multi- and manycore systems* Ganztägiges Tutorial (zusammen mit Prof. G. Wellein und Dr. J. Treibig) bei der „Supercomputer Conference 2013“ (SC13) in Denver, CO, USA
Oktober 2013	*Node-Level Performance Engineering* Ganztägiges Tutorial beim „aiXcelerate 2013 HPC tuning workshop“ an der RWTH Aachen
September 2013	*Node-Level Performance Engineering* Halbtägiges Tutorial bei der „10th International Conference on Parallel Processing and Applied Mathematics“ (PPAM 2013) in Warschau, Polen
	Node-Level Performance Engineering Ganztägiges Tutorial (zusammen mit Prof. G. Wellein) beim „SPPEXA Doctoral Retreat“ an der TU Darmstadt
Juni 2013	*Performance Engineering on Multicore Platforms* Dreitägiges Tutorial (zusammen mit Dr. J. Treibig) im IBM Toronto Lab, Markham, ON, Kanada

	Node-Level Performance Engineering Ganztägiges Tutorial (zusammen mit Dr. J. Treibig und Prof. G. Wellein) bei der „International Supercomputer Conference 2013“ (ISC13) in Leipzig
April 2013	*Specialist Workshops in Parallel Computing 2013: Advanced Multicore* Zweitägiger Blockkurs (zusammen mit Dr. J. Treibig) an den Universitäten Gent und Leuven, Belgien
März 2013	*Node-level performance engineering* Zweitägiger Kurs (zusammen mit Prof. G. Wellein und M. Kreutzer) beim DLR Köln
Dezember 2012	*Performance engineering on multi-and manycores* Halbtägiges Tutorial bei der „3rd Saudi-Arabian HPC Users Conference“ (SAHPC 2012) an der King Abdullah University of Science and Technology (KAUST), Thuwal, Saudi-Arabien
	Node-level performance engineering Zweitägiger Kurs im Rahmen des „PRACE Advanced Training Centre“ (zusammen mit Prof. G. Wellein) am LRZ Garching
November 2012	*The practitioner's cookbook for good parallel performance on multi- and manycore systems* Ganztägiges Tutorial (zusammen mit Prof. G. Wellein) bei der „Supercomputer Conference 2012“ (SC12) in Salt Lake City, UT, USA
Juni 2012	*Performance-oriented programming on multicore-based Clusters with MPI, OpenMP, and hybrid MPI/OpenMP* Halbtägiges Tutorial (zusammen mit Dr. R. Rabenseifner, Dr. J. Treibig und Dr. G. Jost) bei der „International Supercomputer Conference 2012“ (ISC12) in Hamburg
April 2012	*Specialist Workshops in Parallel Computing: Multithreading and Multiprocessing* Zweitägiger Blockkurs (zusammen mit Dr. J. Treibig) an der Universität Gent, Belgien
April 2012 – heute	*Performance-oriented programming on multicore-based systems, with a focus on the Cray XE6* Ganztägiges Tutorial (einmal pro Semester, zusammen mit Dr. J. Treibig) beim Cray Optimization Workshop, HLRS Stuttgart

2007–2013	*Hybrid MPI and OpenMP parallel programming* Halbtägiges Tutorial (zusammen mit Dr. R. Rabenseifner und Dr. G. Jost) bei allen „Supercomputing“ Konferenzen SC07 bis SC13
Juni 2011	*Performance-oriented programming on multicore-based Clusters with MPI, OpenMP, and hybrid MPI/OpenMP* Ganztägiges Tutorial (zusammen mit Dr. G. Jost, Dr. J. Treibig und Prof. G. Wellein) bei der „International Supercomputing Conference 2011“ (ISC11) in Hamburg
Februar 2011	*Ingredients for good parallel performance on multicore-based systems* Halbtägiges Tutorial beim „16th SIGPLAN Symposium on Principles and Practice of Parallel Programming“ (PPoPP11) in San Antonio, TX, USA
November 2010	*Ingredients for good parallel performance on multicore-based systems* Halbtägiges Tutorial bei der „Supercomputer Conference 2010“ (SC10) in New Orleans, LA, USA
Oktober 2010	*C++ für Programmierer* Fünftägiger Kurs mit Übungen am LRZ München
März 2009	*C++ for C programmers* Viertägiger Kurs mit Übungen bei CD-Adapco, Nürnberg
2004/06/08	*Effiziente Nutzung von Hochleistungsrechnern in der numerischen Strömungsmechanik* Vortrag beim NUMET-Kurzlehrgang des Lehrstuhls für Strömungsmechanik (LSTM) der FAU
September 2006	*High Performance Computing: Sequential Code Optimization by Example* und *High Performance Computing: Selected Topics in Shared Memory Parallelization* Vorträge bei der Wilhelm und Else Heraeus Sommerschule zu Computational Many Particle Physics an der Ernst-Moritz-Arndt-Universität Greifswald

2000 – heute	*Parallel Programming for High Performance Systems* Jährlicher Blockkurs zusammen mit dem LRZ München

Mitarbeit in Programmkomitees

2013	*Workshop on Energy-Efficient Supercomputing (E2SC)* Workshop at SC13, Denver, CO, USA, November 2013
	Workshop on Power-aware Algorithms, Systems, and Architectures (PASA) Workshop at ICPP13, Lyon, Frankreich, Oktober 2013
	International Conference on Parallel Programming and Applied Mathematics Research Paper Committee, Warschau, Polen, September 2013
	Workshop on Unconventional High Performance Computing (UCHPC) Workshop at Euro-Par 2013, Aachen, August 2013
2012	*International Supercomputer Conference 2012 (ISC'12)* Research Paper Committee, Hamburg, Germany, Juni 2012
	Workshop on Large-Scale Parallel Processing 2012 Workshop at IPDPS, Shanghai, China, Mai 2012
2011	*Facing the Multi-Core Challenge II* Workshop for young researchers, KIT Karlsruhe, September 2011
	Workshop on Unconventional High Performance Computing Workshop at Euro-Par 2011, Bordeaux, France, August 2011
	Workshop on High Performance Hardware-Aware Computing Workshop at PPoPP11, San Antonio, TX, USA, Februar 2011

Vorträge

Eingeladene Vorträge

2013	*More Science per Joule: Bottleneck Computing* 10th International Conference on Parallel Processing and Applied Mathematics (PPAM 2013), Warschau, Polen, 9. September 2013
	Performance and Power Engineering on Multicore Systems German Research School for Simulation Sciences, RWTH Aachen, 11. März 2013
2012	*Energy efficiency: A down-to-earth perspective* „Cool Supercomputing" BoF, Supercomputing 2012 (SC12), Salt Lake City, UT, USA, 14. November 2012
	Performance Engineering: From Numbers to Insight Workshop on Productivity and Performance (PROPER) at Euro-Par 2012, Rhodos, Griechenland, 28. August 2012

	Performance Engineering for Multi-/Manycores: Unveiling the Mysteries of Application Performance International Supercomputer Conference 2012 (ISC12), Hamburg, 18. Juni 2012
	Simulating Incompressible Flows With the Lattice-Boltzmann Method: Algorithm, Implementation, Performance Physikalisches Kolloquium der Universität Greifswald, 5. Januar 2012
2011	*Common sense in high performance computing* Leogang HPC Workshop, 2. März 2011
	Monitoring, Accounting und Nutzerverwaltung auf den HPC-Systemen des RRZE ZIH Kolloquium, Technische Universität Dresden, 25. August 2011
	Teaching High Performance Computing to Scientists and Engineers: A Model-Based Approach 7th European Computer Science Summit (ECSS 2011), Milan, Italy, 8. November 2011
	Hybrid-parallel sparse matrix-vector multiplication. SC11 BoF „1000x0=0. Single-node optimisation does matter“, Seattle, WA, USA, 17. November 2011
2010	*MPI/OpenMP hybrid computing (on modern multicore systems)* 39th SPEEDUP Workshop on High Performance Computing, ETH Zürich, 6. September 2010
	Thirteen modern ways to fool the masses with performance results on parallel computers 6th Erlangen International High End Computing Symposium, RRZE, 4. Juni 2010, und 12th Teraflop Workshop, HLRS Stuttgart, 15. März 2010
	Hybrid applications on modern architectures: Things to consider SIAM Conference on Parallel Processing for Scientific Computing (PP10), Seattle, WA, USA, 26. Februar 2010
2009	*Wavefront Parallel Temporal Blocking on Multi-Core Processors with Shared Caches* Los Alamos National Laboratory, Performance Architecture Lab (PAL), 26. August 2009
2007	*Are the Killer Micros Still Attacking?* NEC User Group (NUG) XIX. General Meeting, Cetraro (Italien), 24. Mai 2007

Cluster OpenMP
1st HLRS Parallel Tools Workshop, HLRS Stuttgart, 10. Juli 2007

High Performance Computing at RRZE
Computer Chemistry Center (CCC) Seminar, FAU, 23. April 2007

Fachvorträge

2012	*Performance patterns and hardware metrics on modern multicore processors: Best practices for performance engineering* Workshop on Productivity and Performance (PROPER) at Euro-Par 2012, Rhodos, Griechenland, 28. August 2012
	Simulating incompressible flows with the lattice-Boltzmann method: Algorithm, implementation, performance SIAM Conference on Parallel Processing for Scientific Computing (PP12) Minisymposium MS14, Savannah, GA, USA, 15. Februar 2012
2011	*Prospects for Truly Asynchronous Communication with Pure MPI and Hybrid MPI/OpenMP on Current Supercomputing Platforms* Cray User Group Conference 2011, Fairbanks, AK, USA, 25. Mai 2011
	Parallel sparse matrix-vector multiplication as a test case for hybrid MPI+OpenMP programming Workshop on Large-Scale Parallel Processing (LSPP 2011), Anchorage, AK, USA, 20. Mai 2011
2009	*Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes* 17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2009), Weimar, 20. Februar 2009
2007	*Erste Erfahrungen mit Windows Compute Cluster Server 2003* ZKI Arbeitskreis Supercomputing, Gesellschaft für wissenschaftliche Datenverarbeitung Göttingen (GWDG), 25. Oktober 2007
	Erste Erfahrungen mit dem Sun UltraSPARC T2 Prozessor SunDay, RRZE, 6. November 2007
	Performance Evaluation of Current HPC Architectures Using Low-Level and Application Benchmarks HLRB2/KONWIHR Result and Review Workshop, 3. Dezember 2007, LRZ München

2006	*Why is performance productivity poor on modern architectures?* Dagstuhl Seminar on Petacomputing, Dagstuhl, 16. Februar 2006
	First Experiences with Cluster OpenMP Cluster OpenMP workshop, HLRS Stuttgart, 19. Mai 2006
2005	*Erfahrungen und Benchmarks mit Dual-Core Prozessoren* ZKI Arbeitskreis Supercomputing, Universität Karlsruhe, 22. September 2005
	Betrieb eines heterogenen Clusters ZKI Arbeitskreis Supercomputing, Universität Karlsruhe, 22. September 2005
	Benchmarks on Current Dual Core CPUs (and some comments on OpenMP, C++, Tools etc.) Videokonferenz mit ZIH Dresden am RRZE, 10. Oktober 2005
2004	*Investigation of Stripe Formation in Hubbard Ladders using Parallel DMRG* KONWIHR Result and Review Workshop, Technische Universität München, 2. März 2004
	Application Performance: Altix vs. the Rest SGI User Group Conference, Orlando, FL, USA, 27. Mai 2004
	Intel VTune für Linux Videokonferenz mit HLRS Stuttgart am RRZE, 14. Juli 2004
2003	*Parallelization Strategies for Density Matrix Renormalization Group Algorithms on Shared-Memory Systems* DMRG workshop, RRZE, 7. Mai 2003
	Writing Efficient Programs in Fortran, C and C++: Selected Case Studies Workshop on efficient HPC programming, LRZ München, 21. Juli 2003

Preise und Ehrungen

2011	*Informatics Europe Curriculum Best Practices Award 2011: Parallelism and Concurrency* für den Beitrag „Teaching high performance computing to scientists and engineers: A model-based approach“
2009	*Best Paper Award* bei COMPSAC 2009, the 33rd Annual IEEE International Computer Software and Applications Conference, July 20–24, 2009, Seattle, WA, zusammen mit Prof. G. Wellein, Dr. T. Zeiser, Prof. H. Fehske und M. Wittmann.

Publikationen

Buchveröffentlichung

- Georg Hager and Gerhard Wellein:
  Introduction to High Performance Computing for Scientists and Engineers
  CRC Press, Juli 2010, ISBN 978-1439811924, 356 Seiten.
  DOI:10.1201/EBK1439811924

Artikel in Journalen und Tagungsbeiträge mit Peer Review

2013

- M. Wittmann, G. Hager, T. Zeiser, J. Treibig, and G. Wellein: *Chip-level and multi-node analysis of energy-optimized lattice-Boltzmann CFD simulations.* Submitted. arXiv:1304.7664
- M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop: *A unified sparse matrix data format for modern processors with wide SIMD units.* Submitted. arXiv:1307.6209
- T. Scharpff, K. Iglberger, G. Hager, and U. Rüde: *Model-guided Performance Analysis of the Sparse Matrix-Matrix Multiplication.* Proc. 2013 International Conference on High Performance Computing & Simulation (HPCS 2013), July 1–5, 2013, Helsinki, Finland. DOI:10.1109/HPCSim.2013.6641452
- G. Hager, J. Treibig, J. Habich, and G. Wellein: *Exploring performance and power properties of modern multicore chips via simple machine models.* Accepted for publication in Concurrency and Computation: Practice and Experience. arXiv:1208.2908
- F. Shahzad, M. Wittmann, T. Zeiser, G. Hager, and G. Wellein: *An Evaluation of Different IO Techniques for Checkpoint/Restart.* Accepted for the Workshop on Large-Scale Parallel Processing 2013 (LSPP13).
- J. Treibig, G. Hager, and G. Wellein: *Performance patterns and hardware metrics on modern multicore processors: Best practices for performance engineering.* Proc. 5th Workshop on Productivity and Performance (PROPER 2012) at Euro-Par 2012, August 28, 2012, Rhodes Island, Greece. Lecture Notes in Computer Science 7640, 451-460 (2013), Springer, ISBN 978-3-642-36948-3. DOI:10.1007/978-3-642-36949-0_50
- M. Wittmann, T. Zeiser, G. Hager, and G. Wellein: *Comparison of Different Propagation Steps for Lattice Boltzmann Methods.* Computers & Mathematics with Applications **65**(6), 924–935 (2013). DOI:10.1016/j.camwa.2012.05.002

2012

- K. Sembritzki, G. Hager, B. Krammer, J. Treibig, and G. Wellein: *Evaluation of the Coarray Fortran Programming Model on the Example of a Lattice Boltzmann Code.* Accepted for PGAS '12, The 6th Conference on Partitioned Global Address Space Programming Models, Oct 10–12, 2012, Santa Barbara, CA, USA.

- K. Iglberger, G. Hager, J. Treibig, and U. Rüde: *High Performance Smart Expression Template Math Libraries.* Proc. Workshop on New Algorithms and Programming Models for the Manycore Era (APMM 2012) at HPCS 2012, July 2-6, 2012, Madrid, Spain. DOI:10.1109/HPCSim.2012.6266939

- J. Habich, C. Feichtinger, H. Köstler, G. Hager, and G. Wellein: *Performance engineering for the Lattice Boltzmann method on GPGPUs: Architectural requirements and performance results.* Computers & Fluids **80**, 276–282 (2013). DOI:10.1016/j.compfluid.2012.02.013

- J. Treibig, G. Hager, H. G. Hofmann, J. Hornegger, and G. Wellein: *Pushing the limits for medical image reconstruction on recent standard multicore processors.* International Journal of High Performance Computing Applications **27**(2), 162–177 (2013). DOI:10.1177/1094342012442424

- K. Iglberger, G. Hager, J. Treibig, and U. Rüde: *Expression Templates Revisited: A Performance Analysis of the Current ET Methodology.* SIAM Journal of Scientific Computing **34**(2), C42–C69 (2012). DOI:10.1137/110830125

- M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A. Basermann, and A.R. Bishop: *Sparse matrix-vector multiplication on GPGPU clusters: A new storage format and a scalable implementation.* Proc. LSPP12, the Workshop on Large-Scale Parallel Processing at IPDPS 2012, May 25, 2012, Shanghai, China. DOI:10.1109/IPDPSW.2012.211

2011

- G. Schubert, H. Fehske, G. Hager, and G. Wellein: *Hybrid-parallel sparse matrix-vector multiplication with explicit communication overlap on current multicore-based systems.* Parallel Processing Letters **21**(3), 339-358 (2011). DOI:10.1142/S0129626411000254

- G. Schubert, G. Hager, H. Fehske and G. Wellein: *Parallel sparse matrix-vector multiplication as a test case for hybrid MPI+OpenMP programming.* Proc. LSPP11, the Workshop on Large-Scale Parallel Processing at IPDPS 2011, May 20th, 2011, Anchorage, AK. DOI:10.1109/IPDPS.2011.332

- J. Treibig, G. Wellein and G. Hager: *Efficient multicore-aware parallelization strategies for iterative stencil computations.* Journal of Computational Science **2**, 130–137 (2011). DOI:10.1016/j.jocs.2011.01.010

- C. Feichtinger, J. Habich, H. Köstler, G. Hager, U. Rüde and G.Wellein: *A Flexible Patch-Based Lattice Boltzmann Parallelization Approach for Heterogeneous GPU-CPU Clusters.* Parallel Computing **37**(9), 536–549 (2011). DOI:10.1016/j.parco.2011.03.005

- J. Habich, T. Zeiser, G. Hager and G. Wellein: *Performance analysis and optimization strategies for a D3Q19 Lattice Boltzmann Kernel on nVIDIA GPUs using CUDA.* Advances in Engineering Software **42**(5), 266–272 (2011). DOI:10.1016/j.advengsoft.2010.10.007

2010

- M. Wittmann, G. Hager, J. Treibig and G. Wellein: *Leveraging shared caches for parallel temporal blocking of stencil codes on multicore processors and clusters.* Parallel Processing Letters **20**(4), 359–376 (2010). DOI:10.1142/S0129626410000296

- J. Treibig, G. Hager and G. Wellein: *LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments.* Proc. PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures, San Diego CA, USA, September 13, 2010. DOI:10.1109/ICPPW.2010.38

- J. Treibig, G. Hager and G. Wellein: *Complexities of Performance Prediction for Bandwidth-Limited Loop Kernels on Multi-Core Architectures.* In: S. Wagner et al. (eds.), High Performance Computing in Science and Engineering, Garching/Munich 2009. Springer, ISBN 978-3642138713, 3–12 (2010). DOI:10.1007/978-3-642-13872-0_1, Preprint (Multi-core architectures: Complexities of performance prediction and the impact of cache topology): arXiv:0910.4865

- M. Wittmann, G. Hager and G. Wellein: *Multicore-aware parallel temporal blocking of stencil codes for shared and distributed memory.* Proc. LSPP10, the Workshop on Large-Scale Parallel Processing at IPDPS 2010, April 23rd, 2010, Atlanta, GA, USA. DOI:10.1109/IPDPSW.2010.5470813

2009

- T. Zeiser, G. Hager and G. Wellein: *Benchmark analysis and application results for lattice Boltzmann simulations on NEC SX vector and Intel Nehalem systems.* Parallel Processing Letters **19**(4), 491–511 (2009). DOI:10.1142/S0129626409000389

- J. Treibig and G. Hager: *Introducing a Performance Model for Bandwidth-Limited Loop Kernels.* Proc. Workshop "Memory issues on Multi- and Manycore Platforms" at PPAM 2009, the 8th International Conference on Parallel Processing and Applied Mathematics, Wroclaw, Poland, September 13–16, 2009. DOI:10.1007/978-3-642-14390-8_64

- T. Zeiser, G. Hager and G. Wellein: *The world's fastest CPU and SMP node: Some performance results from the NEC SX-9.* Proc. LSPP 2009, the Workshop on Large-Scale Parallel Processing at IPDPS 2009, May 29, 2009, Rome, Italy. DOI:10.1109/IPDPS.2009.5161089

- G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske: *Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization.* Proceedings of COMPSAC 2009, the 33rd Annual IEEE International Computer Software and Applications Conference, July 20–24, 2009, Seattle, WA. DOI:10.1109/COMPSAC.2009.82

- J. Habich, T. Zeiser, G. Hager, and G. Wellein: *Speeding up a Lattice Boltzmann Kernel on nVIDIA GPUs.* Proc. PARENG09-S01, the First International Conference on Parallel, Distributed and Grid Computing for Engineering, Pecs, Hungary, April 2009. DOI:10.4203/ccp.90.17

- S. Ejima, G. Hager, and H. Fehske: *Quantum phase transition in a 1D transport model with boson affected hopping: Luttinger liquid versus charge-density-wave behavior.* Phys. Rev. Lett. **102**, 106404 (2009). DOI:10.1103/PhysRevLett.102.106404

2008

- N. Schindzielorz, J. Erler, P. Klüpfel, P.-G. Reinhard and G. Hager: *Fission of superheavy nuclei explored with Skyrme forces.* Int. J. Mod. Phys. E **18**(4), 773–781 (2009). DOI:10.1142/S0218301309012860

- H. Fehske, G. Hager and J. Jeckelmann: *Metallicity in the half-filled Holstein-Hubbard model.* Europhys. Lett. **84**, 57001 (2008). DOI:10.1209/0295-5075/84/57001

- G. Hager, T. Zeiser and G. Wellein: *Data access optimizations for highly threaded multi-core CPUs with multiple memory controllers.* Proc. LSPP08, the Workshop on Large-Scale Parallel Processing at IPDPS 2008, Miami, FL, USA, April 18, 2008. DOI:10.1109/IPDPS.2008.4536341

- G. Hager, T. Zeiser and G. Wellein: *Data access characteristics and optimizations for Sun UltraSPARC T2 and T2+ systems.* Parallel Processing Letters **18**(4), 471–490 (2008). DOI:10.1142/S0129626408003521

2007

- G. Hager, A. Weiße, G. Wellein, E. Jeckelmann and H. Fehske: *The spin-Peierls chain revisited.* Proc. of ICM 2006, the 17th International Conference on Magnetism, August 20–25 2006, Kyoto, Japan. J. Magn. Magn. Mater. **310**, 1380–1382 (2007), DOI:10.1016/j.jmmm.2006.10.399. Erratum: J. Magn. Magn. Mater. **316**, 43 (2007), DOI:10.1016/j.jmmm.2007.03.184.

- M. Hohenadler, G. Hager, G. Wellein and H. Fehske: *Carrier-density effects in many-polaron systems.* J. Phys.: Condens. Matter **19**, 255202 (2007). DOI:10.1088/0953-8984/19/25/255202

- T. Zeiser, G. Wellein, A. Nitsure, K. Iglberger, U. Rüde and G. Hager: *Introducing a parallel cache oblivious blocking approach for the lattice Boltzmann method.* Proceedings of ICMMES 2006. Progress in Computational Fluid Dynamics, An Int. J. **8**(1/2/3/4), 179–188 (2008). DOI:10.1504/PCFD.2008.018088

2006

- H. Fehske, G. Hager, G. Wellein and E. Jeckelmann: *Hole-doped Hubbard ladders.* Physica B **378–380**, 319–320 (2006). DOI:10.1016/j.physb.2006.01.136

- A. Weiße, G. Hager, A. R. Bishop and H. Fehske: *Phase diagram of the spin-Peierls chain with local coupling.* Phys. Rev. B **74**, 214426 (2006). DOI:10.1103/PhysRevB.74.214426

2005

- G. Hager, G. Wellein, E. Jeckelmann and H. Fehske: *Stripe formation in doped Hubbard ladders.* Phys. Rev. B **71**, 075108 (2005). DOI:10.1103/PhysRevB.71.075108

- H. Fehske, G. Wellein, G. Hager, A. Weiße, K.W. Becker and A.R. Bishop: *Luttinger liquid versus charge density wave behaviour in the one-dimensional spinless fermion Holstein model.* Physica B **359–361**, 699–701 (2005). DOI:10.1016/j.physb.2005.01.198

- G. Wellein, T. Zeiser, S. Donath and G. Hager: *On the Single Processor Performance of Simple Lattice Boltzmann Kernels.* Proc. ICMMES 2004. Computers & Fluids **35**, 910–919 (2006). DOI:10.1016/j.compfluid.2005.02.008

2004

- G. Hager, E. Jeckelmann, H. Fehske and G. Wellein: *Parallelization Strategies for Density Matrix Renormalization Group Algorithms on Shared-Memory Systems.* J. Comput. Phys. **194**(2), 795–808 (2004). DOI:10.1016/j.jcp.2003.09.018

- H. Fehske, G. Wellein, G. Hager, A. Weiße and A. R. Bishop: *Quantum Lattice Dynamical Effects on Single-Particle Excitations in One-dimensional Mott and Peierls Insulators.* Phys. Rev. B **69**, 165115 (2004). DOI:10.1103/PhysRevB.69.165115

2003

- G. Wellein, G. Hager, A. Basermann and H. Fehske: *Fast sparse matrix-vector multiplication for TFlop/s computers.* In: J.M.L.M. Palma et al. (eds.): High Performance Computing for Computational Science – VECPAR2002, Porto, Portugal, 26–28 June 2002. Berlin: Springer, ISBN 3-540-00852-7, 205–207 (2003). DOI:10.1007/3-540-36569-9_18

Beiträge ohne (vollständiges) Peer Review und technische Berichte

2013

- G. Hager: *Performance engineering: From numbers to insight.* Proc. 5th Workshop on Productivity and Performance (PROPER 2012) at Euro-Par 2012, August 28, 2012, Rhodes Island, Greece. Lecture Notes in Computer Science 7640, 393–394 (2013), Springer, ISBN 978-3-642-36948-3. DOI:10.1007/978-3-642-36949-0_44

- M. Wittmann, G. Hager, G. Wellein, T. Zeiser, and B. Krammer: *MPC and Coarray Fortran: Alternatives to Classic MPI Implementations on the Examples of Scalable Lattice Boltzmann Flow Solvers.* In: W. E. Nagel et al. (eds.), High Performance Computing in Science and Engineering '12, Springer, ISBN 978-3-642-33373-6 (2013) 367–372. DOI:10.1007/978-3-642-33374-3_27

- M. Wittmann, G. Hager, T. Zeiser, and G. Wellein: *Asynchronous MPI for the Masses.* Technical report, arXiv:1302.4280

2011

- G. Hager, G. Schubert, T. Schoenemeyer, and G. Wellein: *Prospects for Truly Asynchronous Communication with Pure MPI and Hybrid MPI/OpenMP on Current Supercomputing Platforms.* Proc. CUG 2011, the Cray Users Group Conference 2011, May 23–26, 2011, Fairbanks, AK.

- J. Treibig, G. Hager, and G. Wellein: *LIKWID performance tools.* Accepted for publication in G. Wittum et al. (eds): Competence in High Performance Computing. Springer (2011). arXiv:1104.4874

2010

- M. Wittmann and G. Hager: *Optimizing ccNUMA locality for task-parallel execution under OpenMP and TBB on multicore-based systems*. Technical report, arXiv:1101.0093

- J. Treibig, G. Hager, M. Meier and G. Wellein: *LIKWID performance tools.* InSiDE **8**(1), 50–53 (2010).

- G. Schubert, G. Hager and H. Fehske: *Performance limitations for sparse matrix-vector multiplications on current multicore environments.* In: S. Wagner et al. (eds.), High Performance Computing in Science and Engineering, Garching/Munich 2009. Springer, ISBN 978-3642138713, 13–26 (2010). DOI:10.1007/978-3-642-13872-0_2

- H. Fehske and G. Hager: *Luttinger, Peierls or Mott? Quantum Phase Transitions in Strongly Correlated 1D Electron-Phonon Systems*. In: F. Hensel et al. (eds.), Metal-to-Nonmetal Transitions. Springer Series in Material Sciences, Vol. 132, (Springer), 1–22 (2010). DOI:10.1007/978-3-642-03953-9_1

2009

- G. Hager, G. Jost, and R. Rabenseifner: *Communication Characteristics and Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-core SMP Nodes.* Proc. CUG 2009, the Cray Users Group Conference 2009, Atlanta, GA, USA, May 4-7, 2009.

- M. Wittmann and G. Hager: *A Proof of Concept for Optimizing Task Parallelism by Locality Queues.* Technical report, arXiv:0902.1884

- R. Rabenseifner, G. Hager, and G. Jost: *Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes.* In: Didier El Baz et al. (eds.), Proceedings of the 17th Euromicro International Conference on Parallel, Distributed, and network-based Processing (PDP 2009), Weimar, Germany, February 18–20, 2009 (Computer Society Press) 427–236. DOI:10.1109/PDP.2009.43

- T. Zeiser, G. Hager, and G. Wellein: *Vector computers in a world of commodity clusters, massively parallel systems and many-core many-threaded CPUs: recent experience based on advanced lattice Boltzmann flow solvers.* In: W. E. Nagel et al. (eds.), High Performance Computing in Science and Engineering 08, Transactions of the High Performance Computing Center, Stuttgart (HLRS) 2008, Springer, ISBN 978-3-540-88301-2, (2009) 333-347. DOI:10.1007/978-3-540-88303-6

2008

- M. Breuer, P. Lammers, T. Zeiser, G. Hager and G. Wellein: *Towards the simulation of the turbulent flow over dimples – Code evaluation and optimization for the NEC SX-8.* In: W. E. Nagel et al. (eds.), High Performance Computing in Science and Engineering 07, Transactions of the High Performance Computing Center, Stuttgart (HLRS) 2007, Springer, ISBN 978-3-540-74739-0 / 978-3-540-74738-3, 303–318 (2008). DOI:10.1007/978-3-540-74739-0_21

2007

- G. Hager and G. Wellein: *Architectures and Performance Characteristics of Modern High Performance Computers.* In: H. Fehske et al. (eds.), Lect. Notes Phys. **739**, 681–730 (2008), ISBN: 978-3-540-74685-0. DOI:10.1007/978-3-540-74686-7_26

- G. Hager and G. Wellein: Optimization Techniques for Modern High Performance Computers. In: H. Fehske et al. (eds.), Lect. Notes Phys. **739**, 731–767 (2008), ISBN: 978-3-540-74685-0. DOI:10.1007/978-3-540-74686-7_27

- G. Hager, H. Stengel, T. Zeiser and G. Wellein: *RZBENCH: Performance evaluation of current HPC architectures using low-level and application benchmarks.* In: S. Wagner et al. (eds.), High Performance Computing in Science and Engineering, Garching/Munich 2007. Transactions of the Third Joint HLRB and KONWIHR Status and Result Workshop, LRZ Garching, Dec 3–4, 2007, Springer, ISBN 978-3-540-69181-5, 485–501 (2009). DOI:10.1007/978-3-540-69182-2_39

- M. Stürmer, G. Wellein, G. Hager, H. Köstler and U. Rüde: *Challenges and potentials of emerging multicore architectures.* In: S. Wagner et al. (eds.), High Performance Computing in Science and Engineering, Garching/Munich 2007. Transactions of the Third Joint HLRB and KONWIHR Status and Result Workshop, LRZ Garching, Dec 3–4, 2007, Springer, ISBN 978-3-540-69181-5, 551–566 (2009). DOI:10.1007/978-3-540-69182-2_43

2006

- G. Wellein, P. Lammers, G. Hager, S. Donath and T. Zeiser: *Towards optimal performance for lattice Boltzmann applications on terascale computers.* In: A. Deane et al. (eds.), Parallel Computational Fluid Dynamics – Theory and Applications. Proceedings of the Parallel CFD 2005 Conference, College Park, MD, USA, May 24–27, 2005. Elsevier, ISBN 0-444-52206-9 (2006) 31–40.

- G. Schubert, A. Alvermann, A. Weiße, G. Hager, G. Wellein and H. Fehske: *Spectral Properties of Strongly Correlated Electron Phonon Systems.* In: G. Münster et al. (eds.), NIC Symposium 2006, John von Neumann Institute for Computing, Jülich, NIC Series, Vol. 32, ISBN 3-00-017351-X, 201-210 (2006). http://www2.fz-juelich.de/nic-series/volume32/schubert.pdf

- A. Nitsure, K. Iglberger, U. Rüde, C. Feichtinger, G. Wellein, G. Hager: *Optimization of Cache Oblivious Lattice Boltzmann Method in 2D and 3D.* In: M. Becker et al. (eds.): ASIM 2006 – 19. Symposium Simulationstechnik, Hannover, 12.–14. September 2006. SCS Publishing House, Frontiers in Simulation **16**, 265–270 (2006).

- P. Lammers, G. Wellein, T. Zeiser, G. Hager and M. Breuer: *Have the vectors the continuing ability to parry the attack of the killer micros?* In: M. Resch et al. (eds.): High Performance Computing on Vector Systems. Proceedings of the High Performance Computing Center Stuttgart, March 2005, Springer, ISBN 3-540-29124-5, 25-39 (2006). DOI:10.1007/3-540-35074-8_2

2005

- G. Hager, T. Zeiser and H. Heller: *Setting up ByGRID – First Steps Towards an e-Science Infrastructure in Bavaria.* In: A. Bode et al. (eds.): High Performance Computing in Science and Engineering, Garching 2005. Transactions of the KONWIHR Result Workshop, October 14–15, 2004, Technical University of Munich, Garching, Springer, ISBN 3-540-26145-1, 97–102 (2005). DOI:10.1007/3-540-28555-5_9

- G. Hager, T. Zeiser, J. Treibig and G. Wellein: *Optimizing performance on modern HPC systems: learning from simple kernel benchmarks.* (In: E. Krause et al. (eds.), Computational Science and High Performance Computing II: The 2nd Russian-German Advanced Research Workshop, Stuttgart, Germany, March 14–16, 2005), Notes on Numerical Fluid Mechanics and Multidisciplinary Design **91**, Springer, ISBN 3-540-31767-8, (2006). DOI:10.1007/3-540-31768-6_23

- S. Donath, T. Zeiser, G. Hager, J. Habich and G. Wellein: *Optimizing Performance of the Lattice Boltzmann Method for Complex Structures on Cache-based Architectures.* In: F. Huelsemann et al. (eds.): Frontiers in Simulation: Simulation Techniques – 18th Symposium in Erlangen, September 2005 (ASIM), SCS Publishing, Fortschritte in der Simulationstechnik, ISBN 3-936150-41-9, 728–735 (2005)

- G. Hager, B. Bergen, P. Lammers and G. Wellein: *Taming the Bandwidth Behemoth – First Experiences on a Large SGI Altix System.* InSiDE **3**(2), 24–25 (2005).

- G. Hager, E. Jeckelmann, H. Fehske and G. Wellein: *Exact Numerical Treatment of Finite Quantum Systems using Leading-Edge Supercomputers.* In: H.G. Bock et al. (eds.): Modelling, Simulation and Optimization of Complex Processes, Springer-Verlag Berlin Heidelberg (2005), ISBN 978-3-540-27170-3, 165–175 (2005). DOI:10.1007/3-540-27170-8_13

2004

- G. Hager, G. Wellein, E. Jeckelmann and H. Fehske: *DMRG Investigation of Stripe Formation in Doped Hubbard Ladders.* In: S. Wagner at el. (eds.): High Performance Computing in Science and Engineering 2004 – Transactions of the Second Joint HLRB and KONWIHR Result and Reviewing Workshop (Second Joint HLRB and KONWIHR Result and Reviewing Workshop Munich, Germany, 2–3 March 2004). Berlin: Springer, ISBN 978-3-540-26657-0, 339–347 (2004). DOI:10.1007/3-540-26657-7_31

- G. Wellein, T. Zeiser, G. Hager and P. Lammers: *Application Performance of Modern Number Crunchers.* CSAR Focus, Ed. 12, Summer-Autumn 2004, 17–19 (2004). http://www.csar.cfs.ac.uk/about/csarfocus/focus12/application_performance.pdf

2003

- H. Fehske, G. Wellein, A. P. Kampf, M. Sekania, G. Hager, A. Weiße, H. Büttner and A. R. Bishop: *One-dimensional electron-phonon systems: Mott- versus Peierls-insulators.* In: S. Wagner et al. (eds.) : High Performance Computing in Science and Engineering 2002 – Transactions of the First Joint HLRB and KONWIHR Result and Reviewing Workshop, Garching, Germany, 10–11 October 2002. Berlin: Springer, ISBN 3-540-00474-2, 339–349 (2003).

- G. Hager, F. Deserno and G. Wellein: *Pseudo-Vectorization and RISC Optimization Techniques for the Hitachi SR8000 architecture.* In: S. Wagner et al. (eds.) : High Performance Computing in Science and Engineering 2002 – Transactions of the First Joint HLRB and KONWIHR Result and Reviewing Workshop, Garching, Germany, 10–11 October 2002. Berlin: Springer, ISBN 3-540-00474-2, 425–442 (2003).

- G. Hager, F. Brechtefeld, P. Lammers and G. Wellein: *Processor Architecture and Application Performance in Modern Supercomputers.* InSiDE **1**(1), 8–13 (2003).

2001

- G. Wellein, G. Hager, A. Basermann and H. Fehske: *Exact Diagonalization of Large Sparse Matrices: A Challenge for Modern Supercomputers.* Proc. CUG 2001, the Cray Users Group Summit 2001, Indian Wells, CA, USA, May 21–23, 2001.

Dr. Georg Hager

Erklärung

Hiermit erkläre ich, dass diese Arbeit bisher von mir weder der Mathematisch-Naturwissenschaftlichen Fakultät der Ernst-Moritz-Arndt-Universität Greifswald noch einer anderen wissenschaftlichen Einrichtung zum Zwecke der Habilitation eingereicht wurde. Ferner erkläre ich, dass ich diese Arbeit selbständig verfasst, keine anderen als die darin angegebenen Hilfsmittel benutzt und insbesondere die wörtlich oder dem Sinne nach anderen Veröffentlichungen entnommenen Stellen kenntlich gemacht habe.

Dr. Georg Hager

Danksagung

Mein Dank gilt in erster Linie Herrn Prof. Holger Fehske, der mich schon als Doktorvater begleitet hatte und mich auch zur Anfertigung dieser Arbeit ermutigte. Er hat maßgeblichen Anteil an meinem wissenschaftlichen Werdegang und schuf die Rahmenbedingungen für eine fruchtbare Zusammenarbeit über viele Jahre.

Den aktuellen und ehemaligen Mitarbeitern der Gruppe für High Performance Computing am Regionalen Rechenzentrum Erlangen, insbesondere Gerhard Wellein, Jan Treibig, Thomas Zeiser, Michael Meier, Markus Wittmann, Moritz Kreutzer, Holger Stengel, Faisal Shahzad, Johannes Habich und Gerald Schubert danke ich für ein diskussionsfreudiges und aktives Umfeld, das in dieser Form sicher außergewöhnlich ist. Speziell die Zusammenarbeit mit Dr. Jan Treibig war entscheidend für die Entwicklung des ECM-Modells und des multicore-Powermodells.

Ich danke außerdem dem Kompetenznetzwerk für wissenschaftliches Hoch- und Höchstleistungsrechnen in Bayern (KONWIHR) für die finanzielle Unterstützung der Projekte HQS@HPC und HSMB. KONWIHR hat über mehr als zehn Jahre durch die Förderung von Projekten zur Code-Parallelisierung und -Optimierung wesentlich dazu beigetragen, das Wissen und die Erfahrung auf diesem Gebiet zu erweitern und in den Rechenzentren zu erhalten.

Weiterhin möchte ich den Initiatoren des DFG-Schwerpunktprogrammes „SPPEXA“ dafür danken, eine Initiative auf den Weg gebracht zu haben, die von unschätzbarem Wert für die Entwicklung hochskalierender numerischer Software sein wird. Dank des aus SPPEXA finanzierten Projekes „ESSEX“ kann die Arbeit an effizienten Lösern für dünn besetzte Probleme auch in den folgenden Jahren weitergehen.

Schließlich danke ich meiner Familie, insbesondere meiner Ehefrau Andrea, für die vorbehaltlose Unterstützung aller meiner beruflichen Aktivitäten. Ohne diesen Rückhalt wäre eine erfolgreiche wisenschaftliche Arbeit nicht möglich.