FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)

A framework that supports in writing performance-optimized stencil-based codes

Markus Stürmer, Ulrich Rüde

Technical Report 10-5

A framework that supports in writing performance-optimized stencil-based codes

M. Stürmer, U. Rüde

Lehrstuhl für Systemsimulation
Friedrich-Alexander University of Erlangen-Nuremberg
91058 Erlangen, Germany

{markus.stuermer, ruede}@informatik.uni-erlangen.de

Abstract

On modern multicore processors, many applications run much slower than one would expect when looking at the vast congregated computational power. After a discussion of factors determining the performance on such CPUs, a framework concept that simplifies writing performance-optimized codes for stencil-based algorithms and a prototypical implementation are presented. Finally, the suitability of this approach is discussed by analyzing performance results for three different applications which have been optimized within that framework.

1 Introduction

Looking back three decades, an Intel 8086 CPU running at 10 MHz was able to perform slightly more than 3 million 16-bit integer additions per second. Today, a single Intel Core2 Quad processor running at 2.8 GHz can do nearly 270 billion such additions per second. Obviously, only a factor of 280 can be explained by the increased clock frequency, but an additional speedup of about the same size is caused by exploiting parallelism on instruction, data, and thread level.

Comparing the run time of thirty-year old binaries, one will see much lower advantage of the new over the old CPU. Most concepts that lead to such a huge peak performance have to be considered when choosing and then implementing an algorithm and also during machine code generation. Some concepts are not even applicable for certain problems at all. And finally, performance of code is not only determined by its calculations, but also strongly by its memory accesses and I/O requirements.

Even if high code performance does not play a major role in all applications, e. g. computational fluid dynamics and many others have a vast demand on computational power. Especially if smaller program kernels account for most of many core hours an application consumes, code optimization and employment of more advanced techniques like cache blocking pay off, despite the long time required for implementing them.

This paper presents a novel framework design that enables experienced programmers to implement thread-parallel and cache-efficient stencil-based codes faster. It is not a black box that tries to create fast code, but more a tool that lets the expert focus on the design of data structures and kernels while program flow and synchronization are taken care of in a configurable way.

The rest of the paper is structured as follows: Section 2 gives an overview of factors that determine performance with an emphasis on multicore systems. Section 3 outlines the concept behind the framework and how it addresses the challenges of multicore programming. Implementation details and performance results of scientific kernels written in a prototype version of the framework in C++ are presented in section 4, followed by a deeper analysis of some results in section 5. Section 6 completes the paper with conclusions and an outlook.

2 Challenges of multicore processors

The following section discusses dominant influences on computational performance on multicore systems and how they can be addressed in software. Of course, their impact varies from problem to problem, and also the complexity of addressing them.

Unfortunately, optimization techniques may interfere with each other. As a consequence, no general rule on how code should be optimized can be given. Instead, the best compromise must be found, where the different measurements complement each other best and constrain each other least.

2.1 Parallelism

A huge potential of modern multicore processors is provided by instruction, data, and thread level parallelism. Unfortunately, only instruction level parallelism (ILP) can be made use of automatically and without recompilation, by means of out-of-order execution in superscalar designs. Wider execution units that operate in a single instruction multiple data (SIMD) way can take advantage of data level parallelism, but this must already be considered during code generation.

Eventually, a program must create and synchronize threads explicitly and in cooperation with the operating system. For code in a high-level language that typically uses scalar and serial description, compilers can employ SIMD only for simple algorithms and create satisfactory multi-threaded programs rarely. For real-life problems programmers must at least assist, e. g. by hints in form of pragmas as in OpenMP. In most cases, optimal results require an adaption or even redesign of data structures and control flow.

While some algorithms can be parallelized easily on all levels, like the well-know STREAM benchmark[1], others may expose parallelism only on certain levels or may be inherently sequential.

[1]http://www.cs.virginia.edu/stream/

2.2 Cache efficiency

Many implementations, like of most stencil-based algorithms, are primarily limited by memory bandwidth, or become memory bound when their computational parts have been optimized. Various strategies exist to reduce main memory transfer, especially by making use of the cache hierarchy. While memory layout optimizations can increase spatial locality, cache blocking techniques can increase temporal locality, too.

Special instructions can be employed to influence allocation and eviction of data in the caches, or to bypass them altogether, e. g. to prevent unnecessary placement of cache lines that are only written.

This *memory bottleneck* already exists for serial software on single-cored machines, but becomes more evident for multicore systems: Each core adds to the demand on memory bandwidth, but the mentioned optimization techniques also become more difficult to apply in parallel.

2.3 Thread affinity and memory locations

Multicore processors, and especially multisocket systems built of such, tend to have a complex memory hierarchy. While this does not affect the logical programming model, the distribution of threads to cores becomes more critical for performance.

Typically, only the cores of a socket, and sometimes even only some cores of a multiprocessor, share a common cache. Shared caches are an advantage if threads operate on small data sets cooperatively or need to synchronize often, but for distinct data sets space and bandwidth must be shared, too. Separate caches and multiple caches can provide better cache and memory bandwidth, but are more susceptible to data sharing effects and synchronization overhead.

For multisocket systems with (usually cache-coherent) non-uniform memory access (cc-NUMA), like servers and workstations based on Intel's Core i7 or AMD's Opteron processors, it is equally important on which memory bus data resides. Optimally, each thread processes mainly operands in the memory local to the processor it is running on. If threads mainly access remote data, they will suffer from lower bandwidth or might even affect other threads due to obstruction of memory buses and interconnects.

3 Framework

The following section describes a framework concept that simplifies the process of writing performance-optimized codes for stencil-based algorithms on regular grids for multicore and shared memory multiprocessor machines. Only two-dimensional problems are considered, as extension to 3D is simple in principle, but further investigation is necessary which approach or approaches are generally best. Algorithms of this class appear in different domains, as becomes obvious in section 4, but the majority of them have similar well-understood properties, so that a framework approach is reasonable and appropriate for them.

3.1 Conception

Blocking techniques change the order operations are performed in a way that increases temporal locality to enhance main memory efficiency. Especially temporal blocking techniques can reduce main memory accesses significantly by performing also multiple operations on data already available in the cache hierarchy.

As a perspicuous example, consider two filters with local convolution kernels that are applied to an image, or alternatively two iterations of an iterative method with a small stencil on a regular grid. Assume further that data of multiple lines, but not the whole domain can be held in the cache hierarchy. In a naive implementation, the grid will be traversed two times and all data will be transfered from and to memory twice, too. In the following, we will call a traversal of the grid, no matter which order is used, a *sweep*. But the second operation can already be applied to a line just after the first operation has been completed for the following one, and while the required operands are still available in the caches. This way, only a single sweep is required. In other situations, partial lines, rectangles, or more complex shapes might need to be used. Cache blocking approaches for sequential codes have been examined thoroughly in the DiME[2] project[DiM08, Kow04, Wei01].

The straight-forward approach of parallelizing temporal blocking techniques is to statically split the domain and assign each part to a thread. But then threads need to synchronization at the boundaries between their portions in a complicated way, so that unnecessary restriction are implied on the quite general concept of stencils.

Instead, no ordering of memory accesses of different threads is guaranteed by the framework during a sweep. As a consequence, data can either only be read by multiple threads, or accessed by a single thread if it is to be modified. This is basically what pthread or OpenMP enforce if beginning and end of a sweep are the only synchronization points.

For a sweep, the rectangular domain is first split into multiple *stripes*, each stripe consisting of a similar number of neighboring lines. Each stripe is then assigned dynamically to a thread, which will then compute all results within that stripe. This can be arranged with least effort by means of atomic counters, where each thread "draws" the number of its next stripe. The consistency model now implies that the results cannot be written over the input values (except for the trivial case of 1×1 stencils). Further, only a single computation can be done per sweep for now, as a second one would require input from another stripe due to its data dependencies.

To perform multiple operations per sweep in a cache-efficient way, the stripes are cut, now in the other dimension, into multiple tiles of the same width. Additionally, each thread gets a small number of buffer structures, each being able to temporary hold the operands and intermediate values for computing a tile.

Idealized, the update of a stripe is now performed in the following way: First, operands of the first tile are copied into a buffer structure, including data that is required to fulfill data dependencies from neighboring stripes. Then, the first computations can take place, but usually the tile cannot be completed on one boundary. Now, the second tile is copied from memory, computation of the first tile can be completed and started for the second one. Then, the results of the first tile can be written to their final destination, the same buffer structure can be reused for input data of the third tile, and so on. The scratch structures are used in the way of a ring buffer and always contain a small section of the stripe in various stages of processing.

This approach can only work effectively if the ring buffer structures remain in the cache hierarchy most of the time, because otherwise main memory access would not de- but increase. Fortunately, the containing cache lines are touched regularly, and on most architectures cache pollution by the input and output values can be avoided by special load and store instruction. But even without that support by the ISA, the approach works surprisingly well in most cases due to increasing size and associativity of caches,

[2]http://www10.informatik.uni-erlangen.de/Research/Projects/DiME/

while padding can help for toxic cases.

Clearly, additional time is spent for pure data movement in this idealized form which led to the conception of the framework. But various optimizations can often more than compensate for this: Intermediate values, e. g. coefficients, can be managed in the the ring buffer and will not account to main memory traffic. Optimized copy routines can exploit special features of the hardware more easily, like non-temporal moves available on the x86-64 architecture that can prevent unnecessary allocation of cache lines and reduce cache coherency traffic. Finally, the usage of predefined container classes and associated, optimized copy routines is only optional, so preprocessing or reordering of data can be combined with movement.

Deliberately, no assumptions regarding the actual data structures and compute kernels were made. As the memory layout is extremely important for performance, a programmer should have as much freedom to choose as possible. Likewise, compute kernels need different optimizations depending on algorithm and platform, so too much flexibility could be lost if SIMD processing or a certain update order within a buffer structure is enforced.

The framework component defining stripes and distributing them takes a key position. Depending on the number of threads, the size of buffer structures, and the range of data dependencies, an appropriate size for stripes and tiles must be determined, while additional restrictions might be necessary to apply. Non-uniform memory designs can be considered if the operating system uses a first-touch policy for memory allocation and provides an API to control affinity of threads. A number of stripes is then associated with each memory location and only distributed to the threads running on nearby cores. If a framework-guided sweep is used to initialize the data structures, an appropriate allocation of memory pages will usually be achieved automatically.

Being able to influence thread affinity is also a prerequisite for the exploitation of shared caches. In this setup, two threads can form a *unit* working cooperatively on a single ring buffer that is held in a common cache, with one thread moving data while the other performs computation solely in-cache. If an optimized implementation is heavily memory bound, this approach can be used to fully maximize main memory throughput.

3.2 Prototype implementation

The framework concept has been implemented in C++ with the well-known pthreads library. Standard constructs like inheritance and template mechanisms are used to write code within the framework and utilize the associated utility classes and functions. The optmized code is combined into a C++ class that defines type and size of the ring buffers statically.

When executing, this class first has to configure the thread setup, i. e. how many threads should be spawned, if threads should form units, and —if a default setup is not appropriate— their affinity to certain cores. Depending on which affinity API back-end is used, a description of the processor topology might be required, too.

At least before the first sweep, data structures need to be created and made accessible through class members. Further, the tiling component must be configured by specifying the size of the domain, the width and maximal height of a buffer element, and how many additional lines are necessary to fulfill data dependencies. As constraint, a number can be specified the lines in a stripe must be a multiple of.

During a sweep, the framework manages synchronization, distribution of stripes and tiles, and the ring buffers, and will return control only after a requested number of sweeps has been completed. The following three member functions within the implementation class are executed to perform the actual work:

storage2buffer This function takes two parameters, a reference to the buffer structure it ought to fill and an object that describes the tile position within the domain.

buffer2storage This function is the opposite to *storage2buffer* and will take the same type of arguments.

compute An object is passed that allow access to the ring buffer, but restricts this to a small, configurable number of newest tiles of the current stripe. Typically, a window of two tiles is required.

As these functions have access to any member of the class, information about the actual data structures or parameters need not to be passed through the framework, and can be changed between sweeps as desired.

4 Applications

The following section presents results from three different applications accelerated using the prototype framework.

All measurements were made on an Apple Mac Pro running SUSE Linux 10.3 in 64 bit mode. The machine has two Intel Xeon E5462 processors clocking at 2.8 GHz, a 1.6 GHz front side bus and 4 GiB of DDR2 fully buffered memory in a four-channel uniform memory access (UMA) topology. Table 1 shows the bandwidth achievable in the STREAM benchmark. The original version with arrays of only 16 MB in size was used, so slightly better bandwidth is possible for larger settings.

Table 1: Memory bandwidth in GB/s measured using the STREAM benchmark. The best results from GNU and Intel C compilers[2] were taken, which one is indicated in parentheses. OpenMP was enabled for the parallel measurements.

	number of threads			
	1	2	4	8
copy	5.6 (i)	8.3 (i)	9.1 (i)	8.0 (g)
scale	5.6 (i)	6.6 (i)	7.3 (i)	7.4 (i)
add	4.9 (i)	6.5 (i)	7.1 (g)	7.0 (g)
triad	5.0 (i)	6.7 (i)	7.3 (i)	7.3 (g)

4.1 Iterative solver for Poisson's equation

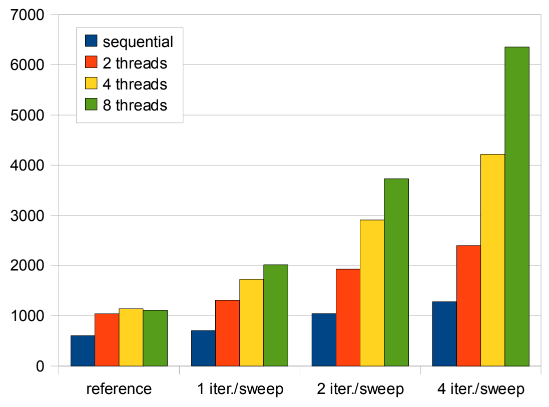
Poisson's equation is a common test problem in numerics. When discretized using finite differences in 2D, the Gauss-Seidel method using red-black ordering is a typical solver for it. From an algorithmic view, it works as follows: The unknowns are split in a checkerboard-like manner into two sets, usually referred to as red and black unknowns. For an iteration, the problem is first solved locally for all red unknowns using a simple five-point stencil operation, and then for the black unknowns as well. As all unknowns of a color set can be updated independent of each other, parallelization of the method is trivial.

This method, however, has a low computational density. Each stencil update consists of five load, six floating point, and a store operation. Assuming that multiple lines of the grid can be held in the cache hierarchy, about 8 B of main memory transfer correspond to a single FLOP in double precision. Thus, roughly one GFLOPS can be expected on the test platform.

Figure 1 shows the performance of a reference implementation in C99 parallelized using OpenMP against an implementation within the framework, which performs multiple whole iterations of the method per sweep. The latter uses SSE copy operations and reorders data during copying to enable vectorization[STR08], which is not applicable for a straight-forward memory layout. The compute kernels, however, have not been SIMD-optimized yet. The effective FLOP rates, i. e. based on the number of operations counted in code and measured run time, are shown.

The performance of the reference implementation is memory bound and corresponds well with the STREAM benchmark, and there's hardly any advantage of using more than a single core per processor. The overhead of separate move and compute kernels is revealed by the sequential performance. Although red and black updates are fused, there's only a small increase for fusing a single iteration into a sweep on a single core. But as eight cores can go twice as fast as the reference implementation, main memory throughput seems to be effectively about halved.

Figure 1: Effective MFLOPS for a straight-forward implementation and an implementation in the prototype framework performing temporal cache-blocking. The grid has a size of 4095×4095 unknowns plus Dirichlet boundary values.



When going from one to four iterations per sweep, a speedup of not more than 1.8 is achieved for one or two threads due to the limited compute power available. This increases to 3.2 if all eight cores are employed, i. e. 80 % of optimum. However, fusing more iterations enlarges data dependencies of a stripe, so that more lines have to be fetched from main memory and more computations need to be done doubly. This effect is reinforcing itself, as the height of stripes will usually have to be decreased, too. Increasing the height of the buffer structures can counteract to some extent, but the implied reduction of tile width might interfere with architectural features like e. g. hardware prefetchers.

[2]gcc 4.3.3 with flags `-O3 -m64 -march=core2 -msse4.1` and icc 11.0 with `-O3 -xSSE4.1 -static-intel -m64`.

Thus, perfect scaling with number of iterations in a sweep is impossible.

The separation into movement kernels accessing main memory and compute kernels operating on caches allows for better investigating the bounds of computation, of main memory transfer, and their interplay.

Table 2 compares the effective performance of the fully functional code against when either compute or copy kernels are disabled. In the serial case, time for movement and computation basically add up. For a single iteration per sweep, about 70 % of the time is spent for data movement, and for four iterations per sweep still about 40 %.

This changes in the most parallel case. Especially for a single iteration per sweep, main memory bandwidth is about a magnitude more limiting than computation. Cores perform data movement most of the time and calculate rarely, enabling constant data flow. Even for more iterations per sweep, data movement remains the major factor.

The numbers also show how this approach of temporal blocking trades additional computations for reduced main memory transfer. Obviously, this works the better the more cores are involved.

Table 2: Milliseconds to perform four iterations actually required and if only movement or computation kernels are active.

iter./sweep	both	computation	movement
		1 thread	
1	569	159	402
2	385	167	210
4	314	188	117
		8 threads	
1	200	22	192
2	108	23	101
4	63	25	55

4.2 Lattice Boltzmann method

Computational fluid dynamics have a large number of applications in science and engineering. Besides classical Navier-Stokes solvers, lattice Boltzmann methods (LBM) have become an interesting alternative. LBM approximate the fluid flow as particle interaction on an equidistant grid of cells.

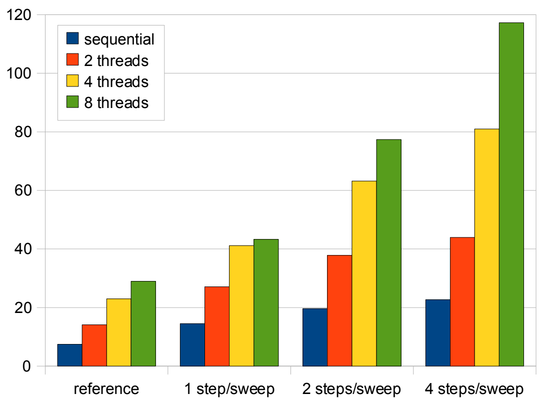
The simple D2Q9 BGK[BGK54] model with bounce-back for obstacle handling was implemented within the framework. As reference, the compute kernel was transfered into a separate C99 program and parallelized using OpenMP.

The first half of a time step consist only of data movement between adjacent cells, but requires many conditional branches to handle obstacles. The second part computes new values for a cell and is completely local. LBM performance is usually not expressed in terms of MFLOPS, but as lattice site updates per second (LUPS).

Figure 2 shows the performance of both codes for a different number of threads and, for the framework version, time steps per sweep. The kernel is a scalar straight-forward implementation of the method, but has not been particularly optimized.

The framework code can get a much better performance even for a single time step per sweep only by using optimized copy kernels that make use of so-called non-temporal memory operations. By implementing a relaxed coherency protocol, they can reduce bus traffic and prevent cache pollution and unnecessary allocation of cache lines that are only written to. A better data reuse on the first cache level, as operating on tiles implies spatial cache blocking, could also play a role.

Figure 2: MLUPS of the LBM kernel, run separately or within the framework for a 2048×2048 domain. Although supported, no obstacle cells were set up inside the domain.



A feature not yet examined is the possibility to let two threads operate cooperatively on a shared cache[WT08]. Obviously, this only makes sense when many cores are utilized, as one or two movement threads could not saturate the memory bus. Table 3 shows the performance for eight threads, either used separately or as four units. For only a few time steps per sweep, forming units is beneficial. The four movement threads can constantly perform memory accesses, as long as the four compute threads can process data fast enough. If many time steps are performed per sweep, computation becomes more important than memory bandwidth, and it is better if all threads take part in it.

Table 3: Performance with eight threads running separately or as four units in MLUPS.

	timesteps per sweep					
	1	2	3	4	5	6
separate	43	77	101	117	134	138
units	52	102	113	108	107	102

4.3 Multigrid solver for image smoothing

As the tiling component of the framework can be reconfigured between sweeps, algorithms working on multiple levels are also possible.

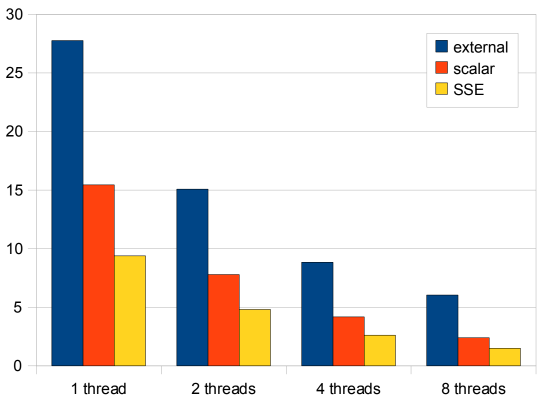
Gilboa et al.[GSZ04] propose to smooth images using a non-linear diffusion model, which requires discretization and solution of a complex partial differential equation. This can be done efficiently using a full approximation scheme (FAS). For an introduction into multigrid methods see e. g. [BHM00]. The solver has been implemented and optimized twice, externally using OpenMP and within the framework.

Common components are used for the multigrid method[Kös08]: As smoother, two ω-Jacobi iterations are applied, the variable coefficients of its five-point stencil need to be also updated before computing the residual. The restriction is performed by averaging the respective four fine-grid values, and constant prolongation is used, both common for a cell-based discretization.

Implemented separately, all components of the multigrid solver are memory bound. To improve cache reuse, as many operations as possible have been fused into a single sweep using the framework, so that each grid level is visited at most twice per V-cycle. When going from a finer to a coarser level, coefficients need to be pre-computed first. Then the right hand side is set up, except for the finest grid, and two iterations of the Jacobi method with successive coefficient updates must be performed, before the residual can be computed and restricted to the next coarser grid together with the current approximation. When going from coarse to a fine, the current approximation needs to be corrected, followed by two Jacobi iterations requiring a coefficient update each.

Figure 3 presents wall-clock times for the external implementation and within the framework, with scalar kernels and kernels optimized for Intel's streaming SIMD extension (SSE).

Figure 3: Run time for five V-cycles on an image of 4096 $\times$ 4096 pixels for three different implementations.



The SSE implementation is nearly faster by a factor of three than the external implementation when running on a single core. Moreover, it scales better with the number of cores, increasing its advance to a factor of more than four when all eight cores are utilized.

The floating point performance computed from run time and operations required for the multigrid would be about 28 GFLOPS, but hardware performance counters measure more than 36 GFLOPS. The replacement of floating point divisions, which cannot be performed pipelined on Core2, by multiplications with reciprocal estimates followed by an additonal Newton-Raphson iteration leads to an increase of operations of about 12 %, although reducing run time. Even more operations are done doubly due to the parallel cache blocking technique.

Still, main memory bandwidth seems not to be saturated, as the scalar implementation has constantly 61 to 63 % of the SSE version's performance, no matter how many cores are used. A deeper analysis of main memory bandwidth is done in the next section.

5 Further analysis

To get an impression how efficient the presented cache blocking approach actually is, the main memory bandwidth of the complex diffusion solver described in section 4.3 was measured based on memory bus requests using the likwid[3] tool and compared to a reasonable estimate of a lower bound: As the finer grids require much more memory than caches present, the bytes required to load or store the affected arrays at each level during the coarsening and

[3]the Google code project page of *Like I knew what I am doing* can be found at `http://code.google.com/p/likwid/`

during the prolongation phase are counted. For the finest levels, we assume that data can be held in the caches and needs to be read and written to main memory at most once.

Table 4 shows that in the serial case only 10 % more memory transfer than the conservative estimate was observed, and about 25 % in the parallel case. The difference is partially caused by data that needs to be loaded doubly, but also thread synchronization and instruction code contribute.

Table 4: Measured memory transfer and estimated lower bound for one V-cycle of the SSE-optimized complex diffusion mutligrid solver in MB.

		threads in parallel		
estimate	serial	2	4	8
1162	1283	1448	1433	1460

Experiments have shown that write misses on the lowest cache level do not necessarily lead to prior allocation of the whole cache line for the Core2 architecture, but that non-temporal stores, as they are used in the SSE kernels, can effectively prevent it. As a consequence, the scalar kernels —whose values are not shown here— transfer only about 10 to 20 % more data instead of about 50 %.

6 Conclusions and future work

Main memory bandwidth was identified as a limiting factor for many stencil-based algorithms. Although its influence can be reduced by temporal blocking techniques, their implementation is tedious, especially in parallel. The feasibility of a framework supporting that task was demonstrated, but still experience and quite some time are required.

Currently, the framework is undergoing a major revision. Besides minor improvements, support for STI's Cell Broadband Architecture (CBEA) has been added. This heterogeneous multicore processors feature not only general purpose, but also specialized accelerator cores. Those cannot directly access main memory, but use DMA transfers to copy data into an own, fast, but small memory.

As a consequence, compute and movement kernels cannot be implemented as member functions of a single, central class. Instead, the framework provides an interface to transport information between the main program running on a general purpose core, which controls execution and manages data structures in the main memory, and code running on the accelerators, which need to operate on that data. To enable compilation on CBEA and standard platforms without modifications, a whole toolbox of container classes and an unified interface for data movement and communication is required.

First results on the CBEA, including an implementation of the complex diffusion solver, are promising, but out of this paper's scope.

7 Acknowledgements and remarks

Many thanks to Dr.-Ing. Harald Köstler for persuading and convincing me to implement the multigrid method for solving the complex diffusion problem within the framework, and him and Mr Daniel Ritter for giving valuable suggestions for this paper.

This report is heavily based on a submission to the Konwihr[4] and Intel® multi-core challenge 2008/2009, which was awarded a third prize.

References

[BGK54] P.L. Bhatnagar, E.P. Gross, and M. Krook. A Model for Collision Processes in Gases. I. Small Amplitude Processes in Charged and Neutral One-Component Systems. *Phys. Rev.*, 94(3):511–525, 1954.

[BHM00] W. L. Briggs, V. E. Henson, and S. F. McCormick. *A Multigrid Tutorial.* SIAM, 2 edition, 2000.

[DiM08] Abschlussbericht des Projekts Ru 422/7-5 (DiME-2). Lehrstuhl für Informatik 10 (Systemsimulation), Friedrich-Alexander-Universität Erlangen-Nürnberg; Lehrstuhl für Rechnertechnik und Rechnerorganisation / Parallelrechnerarchitektur, Technische Universität München; Lehrstuhl für Rechnerarchitektur und Parallelverarbeitung, Universität Karlsruhe, 2008. http://www10.informatik.uni-erlangen.de/Research/Projects/DiME/pubs/DiME_final.pdf.

[4]the homepage of the *Kompetenznetzwerk für Technisch-Wissenschaftliches Hoch- und Höchstleistungsrechnen in Bayern* can be found at `http://www.konwihr.uni-erlangen.de/`

[GSZ04] G. Gilboa, N. Sochen, and Y.Y. Zeevi. Image Enhancement and Denoising by Complex Diffusion Processes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1020–1036, 2004.

[Kös08] Harald Köstler. *A Multigrid Framework for Variational Approaches in Medical Image Processing and Computer Vision*. PhD thesis, 2008.

[Kow04] M. Kowarschik. *Data Locality Optimizations for Iterative Numerical Algorithms and Cellular Automata on Hierarchical Memory Architectures*. Jun 2004. Advances in Simulation 13.

[STR08] M. Stürmer, J. Treibig, and U. Rüde. Optimising a 3D multigrid algorithm for the IA-64 architecture. *Int. J. Computational Science and Engineering*, 4(1):29–35, 2008.

[Wei01] C. Weiß. *Data Locality Optimizations for Multigrid Methods on Structured Grids*. PhD thesis, Lehrstuhl für Rechnertechnik und Rechnerorganisation, Institut für Informatik, Technische Universität München, Munich, Germany, December 2001.

[WT08] J. Weidendörfer and C. Trinitis. Offloading application controlled data prefetching in numerical codes for multi-core processors. *Int. J. Computational Science and Engineering*, 4(1):22–28, 2008.