Optimising a 3D multigrid algorithm for the IA-64 architecture

Markus Stürmer*, Jan Treibig and Ulrich Rüde

Department of Computer Science 10 (System Simulation),
University of Erlangen-Nuremberg,
Cauerstraße 6, 91058 Erlangen, Germany
E-mail: markus.stuermer@informatik.uni-erlangen.de
E-mail: jan.treibig@informatik.uni-erlangen.de
E-mail: ulrich.ruede@informatik.uni-erlangen.de
*Corresponding author

Abstract: Multigrid methods are amongst the most efficient algorithms to numerically solve partial differential equations. However, standard implementations usually cannot exploit the potential of modern processors. The IA-64 architecture transferes most complexity to the software side to provide a highly superscalar design with large caches, leading to unique control over the actual execution. Exemplified on a simple multigrid solver equation in 3D and the Itanium 2 processor, we present how known performance optimisation techniques can be successfully combined. While implementation details are specific, the optimisation concept should be applicable for a wide range of numerical algorithm and CPUs.

Keywords: performance optimisation; multigrid; cache blocking; streaming execution; IA-64.

Reference to this paper should be made as follows: Stürmer, M., Treibig, J. and Rüde, U. (2008) 'Optimising a 3D multigrid algorithm for the IA-64 architecture', *Int. J. Computational Science and Engineering*, Vol. 4, No. 1, pp.29–35.

Biographical notes: Markus Stürmer received his Diplom in Computer Science in 2006 and is currently Research Assistant at the System Simulation Group of the University of Erlangen-Nuremberg and writing his PhD thesis. His research interests are performance optimisations of scientific applications with a focus on multi- and future manycore architectures.

Jan Treibig received his Diplom-Ingenieur in Chemical Engineering at the University of Erlangen-Nuremberg in 2002 and is writing his PhD thesis. His research interests are low-level optimisation of bandwidth-limited, iterative algorithms, and the influence of ISA and hardware characteristics of modern computer architectures and compilers on their performance.

Ulrich Rüde is Professor in the Department of Computer Science at the University of Erlangen-Nuremberg and is the Head of the Chair for System Simulation. His fields of interest include high performance computing for applications in science and technology, computational science and engineering, modelling and simulation, numerical analysis, and adaptive multilevel algorithms. He is currently the Editor-in-Chief of the *SIAM Journal on Scientific Computing*.

1 Introduction

Solving large linear systems of equations, as they emerge in the field of numerical simulation from the discretisation of partial differential equations, is a highly demanding task. Even when using multigrid methods, which offer approximately optimal complexity and are amongst the most effective solvers for partial differential equations, algorithmic optimisation is suggestive to exploit the available computational power. To demonstrate and study optimisation techniques, a simple multigrid solver for Poisson's equation in 3D with Dirichlet boundary conditions as outlined in Briggs et al. (2000) was chosen as model problem. The cubic problem domain Ω is discretised by a regular grid with $2^n - 1$ unknowns in every dimension. A simple V-cycle with trilinear interpolation, full weighting restriction and a Red Black Gauss Seidel smoother with a seven point stencil is used; further a Full Multigrid method is built upon it.

Previous work identified the memory subsystem as the primary bottleneck for scientific computations. Main memory bandwidth obviously limits the achievable overall performance. For iterative algorithms it is possible to reuse data, which is already in cache, multiple times. By that, the pressure on the memory bus is lowered significantly. This technique, better known under the term cache blocking, is an established way to achieve better performance on cache-based architectures; on cache blocking techniques for iterative solvers see e.g., Douglas et al. (2000), Kowarschik et al. (2001) or

Kowarschik (2004). Unfortunately, modern architectures introduce other hardware optimisations apart from a memory hierarchy, which often contradict to the common blocking techniques. These other optimisations involve, e.g., SIMD instructions and data prefetching in hardware or software. These new techniques can be summarised under the term streaming execution. To effectively use these new processors, it is not sufficient to concentrate on the memory bottleneck while neglecting other aspects of the architecture. We present an adapted optimisation strategy which lowers the memory bandwidth requirements while providing a suitable data access pattern for improved data prefetching, which is crucial to get a high sustained memory bandwidth on modern architectures. While these requirements are shared by many mainstream architectures, we picked the IA-64 architecture as target platform. The IA-64 architecture in opposite to common architectures uses a simplified core logic and lets most of the optimisation be done in software. The saved transistors on the die can instead be used for other resources as register files, more arithmetic logical units and larger caches. It provides massively superscalar design with many execution units and an impressive fast and large cache hierarchy. Much of the complexity has been transferred to the compiler or assembler programmer that must create code much more diligently. Only by considering latencies when scheduling instructions and using the special concepts of the architecture good performance can be achieved. While this makes code generation more difficult, it also gives much more control over execution and enables better evaluation of the applied optimisation techniques. For this transparency and massive hardware capabilities the Itanium 2 processor, current implementation of this architecture, was chosen as test platform. We will show, that our improved streaming aware blocking approach together with an optimally vectorised arithmetic core achieves very high performance on our target architecture. The paper starts with an overview of the implemented algorithm. After that the applied optimisation techniques are described followed by detailed performance results. We conclude with a summary of our results.

2 Description of the adopted multigrid algorithm

The following section will outline our model problem and the different components of the multigrid algorithm that is used to solve it. A more extensive introduction to multigrid can be found, e.g., in Briggs et al. (2000), for a more detailed overview of multigrid methods we refer to Brandt (1977), Hackbusch (1985) or Trottenberg et al. (2001).

We want to numerically solve Poisson's equation in 3D with Dirichlet boundary conditions

$$\Delta\phi = f \text{ in } \Omega \text{ and } \phi = \Gamma \text{ on } \delta\Omega.$$

We assume that $\Omega = [0,1]^3$ and discretise ϕ and f on a grid with uniform mesh size h. The Poisson operator is discretised using finite differences resulting in the well-known seven point stencil

$$\begin{aligned}\Delta^\star = \frac{1}{h^2}[&-6\phi(x,y,z) + \phi(x+h,y,z) + \phi(x-h,y,z)\\ &+ \phi(x,y+h,z) + \phi(x,y-h,z)\\ &+ \phi(x,y,z+h) + \phi(x,y,z-h)].\end{aligned}$$

This results in a linear system of equations

$$Ax = b.$$

While direct solvers are often very efficient for small problems, larger problems are usually solved with iterative solvers by subsequently improving an initial guess $\hat{x}_0$.

Two observations are necessary to understand the implemented multigrid method: First, most iterative solvers, like Jacobi or Gauss Seidel methods, reduce oscillatory error components in few iterations but take many to actually converge to the final solution. Second, the usually unknown algebraic error in the ith iteration $e_i = \hat{x}_i - x$ and the residual $r_i = b - A\hat{x}_i$ satisfy the residual equation $Ae_i = r_i$. By that equation, the error can be calculated and used to correct the approximation.

This inspires the two-grid correction scheme:

- use few sweeps of an iterative method as smoother for the error in the approximation
- calculate the residual and generate a representation on a coarser grid
- obtain the error by solving the residual equation on the coarse grid, which is less costly
- prolongate the error to the finer grid and use it to correct the approximation
- apply some more iterations of the smoother, as the approximation has been corrected by an interpolated error.

In multigrid the coarse grid problem is solved recursively, until the grid size permits efficient solving using a direct solver or an iterative method. We restrict ourselves to a simple variant and start on the finest grid with $(2^n - 1)$ unknowns in every dimension. After applying ν_1 iterations of a Red Black Gauss Seidel (RBGS) method, which is known to smooth the error very well, we calculate the residual and coarsen it to a grid with $(2^{n-1} - 1)$ unknowns in every dimension using a weighted $3 \times 3 \times 3$ point stencil (full-weighting restriction). We continue until the coarsest grid has a single unknown and therefore can be solved exactly using a single Gauss Seidel step. The solutions are prolongated using trilinear interpolation and ν_2 RBGS steps are applied after correction on every grid level. While still being an iterative method, each multigrid $V(\nu_1, \nu_2)$-cycle can reduce the error by more than a magnitude, depending on the type of smoothing and grid transfer used. More complex multigrid cycles are possible, but we will restrict ourselves to V-cycles only.

The idea of Full Multigrid (FMG) is to construct a good initial approximation recursively: Coarser representations

of the right hand side are generated and only on the coarsest grid level a true guess, typically the zero solution, is made. This coarse approximation is improved using one (or more) multigrid cycles and then interpolated to the next finer level. If the convergence rate of the multigrid cycles is good enough and the grid transfer is of sufficiently high order, an FMG has asymptotically optimal complexity of $O(n)$; i.e., the approximation error in solving the linear system is about the error made in discretising the underlying partial differential equation.

3 Optimisations

The following section presents a short summary of the applied optimisation techniques. The first step is to optimise the memory layout. While it was primarily developed to support the Red Black Gauss Seidel method, the other components of the algorithm benefit, too. The chapter continues with optimisations that reduce the number of floating point calculations and memory transfers compared to a straightforward implementation. Then the spatial and temporal blocking techniques are presented and it is shown how software prefetching can be used to improve their efficiency. The last section is about using freed memory bandwidth by parallelisation using a second processor sharing the same memory bus.

Some code level optimisations will not be discussed, as they are very special to the IA-64 architecture and not applicable elsewhere; the most important amongst them is a method to reduce the usually large overhead of short loops compilers currently cannot apply automatically.

3.1 Memory layout

The memory layout implemented is based on Stürmer (2005). The unknowns and the surrounding boundary values are split into two arrays depending on their 'colour' in the RBGS method, and the lines are filled up to equal length. The right hand side b is stored the same way to simplify the memory address calculation.

This yields four arrays in total, containing only red or black points each, which we will call *half arrays* for convenience from now on. As primary advantage red and black values are not mixed in the same cache line anymore and can be read or modified independently. When blocking methods are applied, main memory throughput is not saved this way, but transfer between the different cache levels can still be reduced.

Most modern CPUs provide some kind of SIMD unit that has a higher memory bandwidth to the caches and higher arithmetical throughput than the scalar units, but also puts higher restrictions on order and alignment of operands in memory. Further padding should be used at problem sizes where cache thrashing leads to loss of performance. If the *half arrays* are aligned and padded properly, this memory layout supports optimal SIMD processing throughout the multigrid algorithm. The first unknown of every *half line* must be aligned to the SIMD vector size and the lines padded to equal length for it.

While the Itanium 2 does not provide double precision SIMD operations, it still benefits as faster load operations for naturally aligned 16 Byte pairs of floating points values can be used (so called Parallel Loads).

3.2 Optimisation of the algorithm

For the given multigrid algorithm, a lot of computations and memory can be saved without changing the result, as shown in Stürmer et al. (2006).

The Gauss Seidel method sets the new value of a point to locally satisfy the underlying equation. Especially the RBGS variant has useful properties that can be exploited: Every red (or black) sweep only depends on black (or red) values and the pointwise residual on every red (or black) point evaluates to zero.

If at least one pre-smoothing step is applied, the residual at every black point can be assumed zero (within computational accuracy) afterwards and only 14 of 27 points of the restriction stencil must be considered.

Further, the first red post-smoothing step depends only on black values, hence only those need to be interpolated from the coarser grid and corrected. Moreover, the *half array* usually containing the red unknowns can temporarily hold the pointwise residuals between pre-smoothing and restriction.

Provided that at least one pre- and one post-smoothing RBGS step is done, the following optimised, memory throughput and instruction reduced V-cycle can be used: After pre-smoothing, the residual for the red points is calculated and written into the memory locations of the corresponding unknowns. The restriction only has to consider data from this *half array*. Later, only the black unknowns are corrected, after the needed values have been interpolated in place. The first red update of the subsequent post-smoothing step will overwrite the no longer needed residuals with new approximations of the red unknowns.

To coarsen the right hand side and the boundary conditions for the Full Multigrid we use direct injection, assuming those are exact values of an underlying PDE problem. The trilinear interpolation is also used to transfer the solutions to finer grids, which will be sufficient for most applications.

3.3 Spatial blocking

All multigrid components can use spatial blocking techniques to reuse operands on cache and even register level, the latter especially on the IA-64 architecture due to its large register file.

When the residuals are restricted to the coarse right hand side, the fine grid points are traversed and their contribution is added to the surrounding coarse grid points. This is faster than computing the fine grid point by point by applying the restriction stencil, as the coarse grid has only eighth the size of the fine grid. This is still

true for our optimised program where we only need the red residuals and therefore only half as much fine grid data.

On the IA-64 architecture we traverse four neighbouring fine grid lines at once and distribute the values to four course grid lines. Every fine grid line has to be visited only once and every coarse grid line four times. It should be noted that physically data from four fine *half lines* is distributed to eight coarse *half lines*.

For smaller grids the coarse grid residual data will reside in the cache hierarchy for all four updates. To achieve this for larger planes, too, another blocking level must be introduced and only some lines of every plane are processed at once.

Correction and interpolation is done the other way round, i.e., the values of four coarse grid lines are used to update four fine *half lines* of black unknowns.

Relaxation and residuals are also calculated in four neighbouring lines, so values from 12 lines are used concurrently. This also allows to exploit common expressions, so the number of fused-multiply-add operations needed for a relaxation is reduced from seven to six and a half.

3.4 Temporal blocking

The highest potential for improvement lies in using a temporal blocking method for the RBGS method. The smoother is the most expensive component of the multigrid solver, as well in terms of memory transfer as arithmetical operations (Weiß, 2001). By fusing complete or multiple iterations in a single sweep, the main memory data transfer can be reduced significantly.

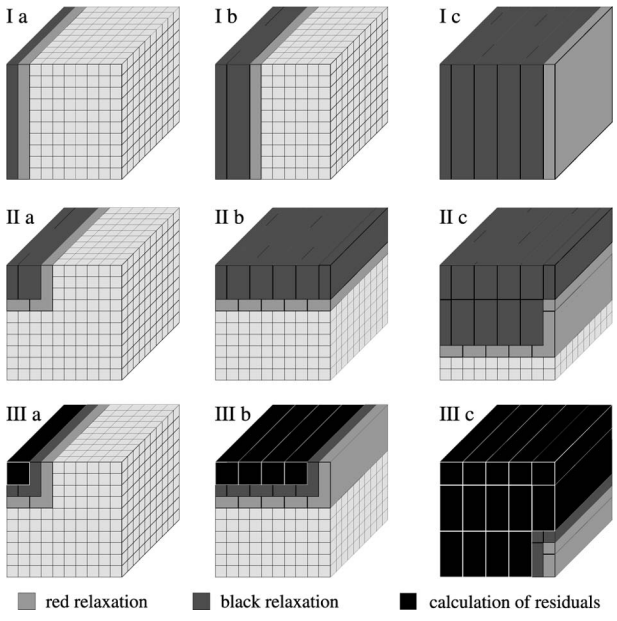
For a simple blocking method, the calculation of i combined iterations starts with a special handling at the first planes (Figure 1 I a): Therein the first red update is calculated in the first $2 \times i$ planes, followed by a black update in the first $2 \times i - 1$ planes and so on, until all i iterations have been completed in the first plane.

The blocked calculation itself begins with the first red update in the $(2 \times i + 1)$th and $(2 \times i + 2)$th plane and continues with the first black update in plane $2 \times i$ and $2 \times i + 1$ in a cascade like manner until the ith black update in the second and third plane (I b). This cascade is moved two planes further and so on until the other side is reached (I c). A special border handling – analogous to the beginning – completes the computation. As long as all data needed for two cascades can be held in caches, every operand has to be loaded into the cache hierarchy and, if modified, written back to main memory exactly once.

For greater plane sizes the introduction of another blocking level is required: The grid is split in x-z-plane into *super-blocks* containing only a section of every plane, so that data can be held in caches long enough to use the above described blocking scheme effectively again (Figure 1 II). These *super-blocks* must overlap, however, to fulfill data dependencies, and some operands must be transferred from and to main memory at least twice. The more iterations are combined in one sweep, the more lines this overlapping region will cover and the fewer lines a *super-block* can contain either. Larger caches or shorter lines, on the other hand, allow higher *super-blocks*.

Often the calculation of the residual and its restriction are fused, but it seems more appropriate here to calculate the red residuals in a blocked manner like an $(i + 1)$th red relaxation step (Figure 1 III). As the residual is written over the already modified red unknowns, calculation of residuals – or of a residual norm, if desired – is nearly free in terms of main memory transfer and primarily limited by arithmetical in-cache performance. Only for larger planes the height of necessary *super-blocks* must be reduced which induces some penalty.

Figure 1 Spatial and temporal blocking of smoother and calculation of the residual. *(I) simple plane blocking of a single RBGS iteration* (a) initial boundary handling (b) first block (c) blocking complete, only boundary handling missing.
(II) super-blocking of a single RBGS iteration (a) first super-block boundary handling and its first sub-block (b) first super-block complete (c) second super-block without final boundary handling.
(III) super-blocking of one RBGS iteration fused with calculation of residual (a) first super-block boundary handling and its first sub-block (b) first super-block except its final boundary handling (c) everything finished except final boundary handling



3.5 Prefetching

The long latencies of typically a few hundred CPU cycles for accessing memory, which is not residing in cache, can have an enormous impact on program performance. Most modern processors therefore provide dedicated hardware trying to detect data streams and load data into the cache hierarchy early by guessing what might be needed in the near future.

The IA-64 architecture, however, completely relies on the compiler or assembly programmer to generate

appropriate prefetch instructions for the various data streams. While this puts high demands on the code generation, even several complex data streams can be handled concurrently that way. Especially if the access patterns are known in advance, like it is the case in many numerical algorithms, software prefetching can anticipate changes or jumps in the data streams.

If multiple lines are processed concurrently, as with the proposed spatial blocking technique, data streams jump at the end of lines. As line length and prefetch distance are constant, it is possible to make the prefetch stream jump and keep ahead of the data stream easily.

For the blocking RBGS smoother, as described above, most memory transfer will occur during the first red update in a new memory area and determine performance. The following black update will only read new black values from the right hand side, if split arrays are used. Iterations fused additionally will depend on already available data, only; their performance depends on cache and computational performance of the CPU. Optimally, memory transfer should be distributed equally over the computation.

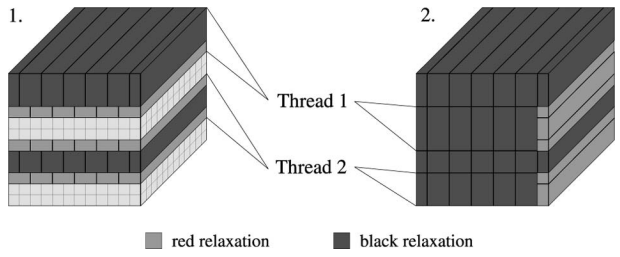
This is accomplished by software prefetching. Prefetching data for updating the next planes is distributed equally over the first red and the following fused updates of previous planes – computation itself is done on prefetched data only. The more iterations are performed per sweep, the fewer data has to be prefetched during each update.

Prefetch instructions on the IA-64 architecture provide fine control over data locality. Operands from future planes are brought from main memory into the lowest cache hierarchy while values needed soon are fetched into higher levels. We'll refer to this method as extended prefetching.

3.6 Parallelisation

While all multigrid components could be parallelised easily, at least for machines with unified memory access, all components are able to saturate the memory bus with a single CPU. An exception is the smoother with multiple iterations fused.

Figure 2 Parallelisation of the blocked smoother. (1) each thread doing its first super-block and (2) as the 'lower' thread has finished its first super-block in time, the 'upper' thread can directly overlap its last with the other's first super-block



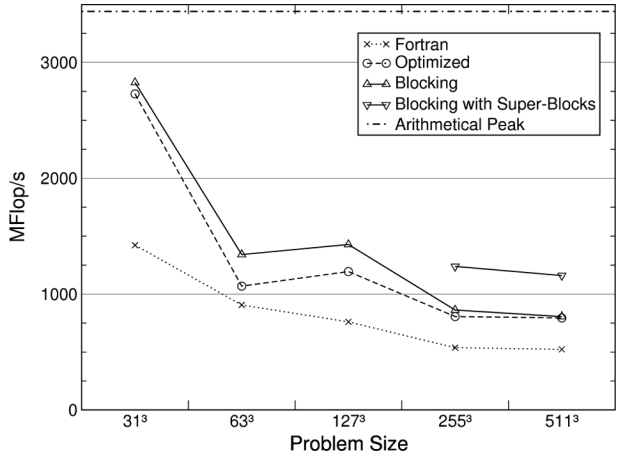
Its parallelisation, possible with or without subsequent calculation of the residual, utilises the available blocking method: Every thread is statically assigned a contiguous portion of super-blocks (Figure 2). Only at the overlap of those areas, the threads have to negotiate using semaphores to respect data dependencies.

4 Results

The optimisations described in the previous chapter were implemented in assembly language and tested on a HP zx6000 workstation with two Itanium 2 processors running at 1.4 GHz and having 1.5 MB of level 3 cache each. The machine has 10 GB of double data rate SD-RAM with a theoretical throughput of 6.4 GB per second. The main memory bandwidth is shared by the two processors.

Figure 3 shows the performance of the optimised RBGS code and the efficiency of the blocking methods. As reference the performance of a straightforward implementation with an usual memory layout in Fortran is included. Only by updating four lines together and using the split memory layout, the performance is substantially improved. Plane blocking is only advantageous for smaller problems, but will degenerate to performance of a non-blocking smoother if the planes do not fit into the caches anymore. However most of the performance can be preserved for large grids by introducing another blocking level.

Figure 3 Performance of the RBGS smoother for single iterations



Another good reference value is the theoretical peak performance of the algorithm. For every RBGS update, seven fused-multiply-add operations – counting as 8 Flops – are necessary. As the Itanium 2 can perform two such operations per clock cycle, every update takes at least 3.5 cycles, effectively 4 if the loop has not been unrolled by the compiler or programmer. For a 1.4 GHz CPU, a maximum of 2.8 GFlop/s without or 3.2 GFlop/s with loop unrolling can be reached. As the spatial blocking method allows reuse of common expressions, only 6.5 operations are necessary per relaxation in our optimised code, taking at least 3.25 cycles if properly unrolled. If still counting 8 Flops per update, 3.44 GFlop/s are the maximum for the optimised implementation.

Especially for multiple iterations per sweep, program performance is compromised by memory bursts. Figure 4 shows that the efficiency of blocking methods can be improved significantly when prefetching is not only used to cover the load latencies of the current data streams, but also to smooth the main memory transfer. At least on our test platform, this extended prefetching is a more powerful optimisation technique than fusing more iterations.

Figure 4 Performance of the RBGS smoother with multiple iterations and extended prefetching

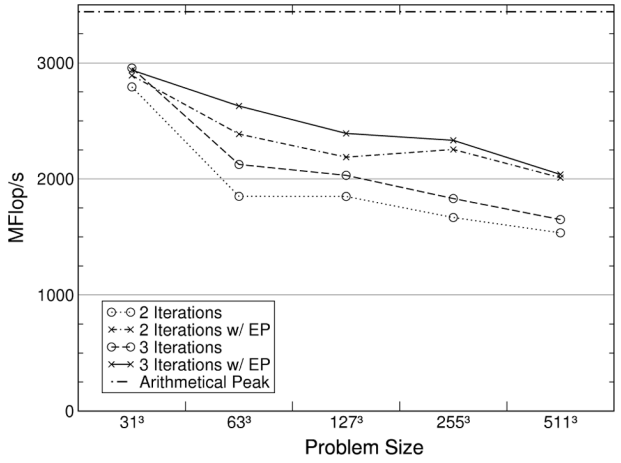
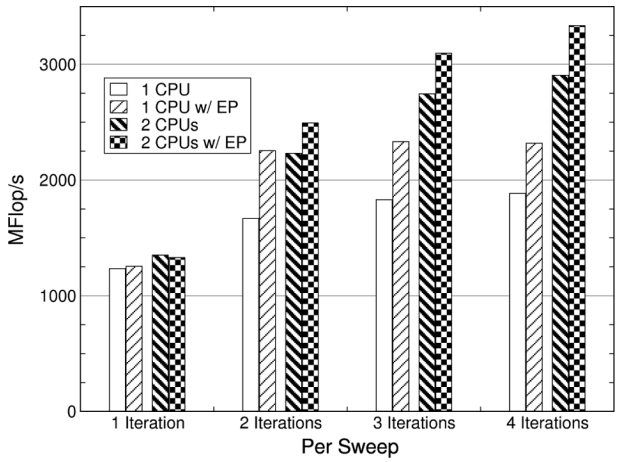


Figure 5 shows the impact of parallelisation, exemplified at grid size 255^3. For one iteration per sweep, a single CPU can nearly saturate the memory bus even without extended prefetching – the optimised memory layout already distributes memory transfer over red and black updates.

Figure 5 Performance of serial and parallel RBGS smoother at problem size 255^3



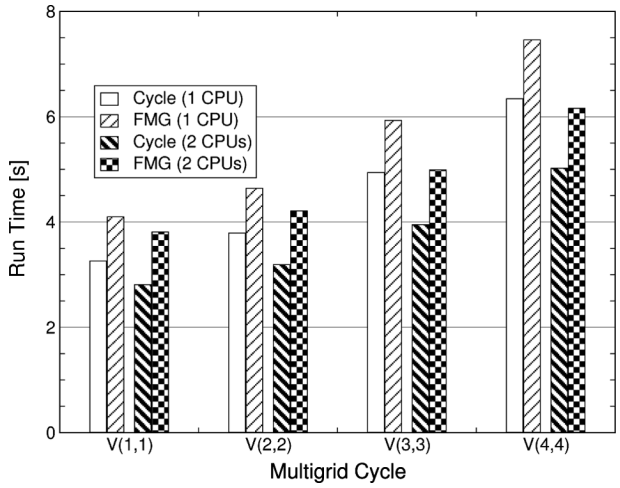
Even a sweep with two iterations is memory bound in theory for this combination of CPU and memory. But as a straightforward implementation using temporal blocking leads to memory bursts, its performance does not scale as expected. Extended prefetching can alleviate this problem very well and is more efficient than using the second CPU. However, it is not able to fully saturate the memory bus and the combination of parallelisation and extended prefetching (EP) can improve performance further.

With three or more iterations per sweep, the smoother is CPU bound and profits more from parallelisation, though extended prefetching still gives additional performance. It conforms that our optimised solver has such low bandwidth requirements for multiple blocked iterations, that a second CPU sharing the same memory bus is still able to improve performance.

Table 1 Runtime and asymptotic convergence rate for different V-cycles on one and two CPUs; extrapolated convergence rate per second and runtime until error reduced by a factor of 10^{-9} for comparison

		single CPU			*two CPUs*		
Cycle	c_{cycle}	t_{cycle}	c_{1s}	$t_{10^{-9}}$	t_{cycle}	c_{1s}	$t_{10^{-9}}$
V(1,1)	0.24	0.39	0.026	5.68	0.35	0.016	5.03
V(1,2)	0.15	0.41	0.011	4.55	0.36	0.005	3.93
V(1,3)	0.12	0.47	0.011	4.58	0.39	0.004	3.78
V(1,4)	0.10	0.55	0.014	4.84	0.47	0.007	4.20
V(2,1)	0.15	0.41	0.011	4.55	0.36	0.005	3.94
V(2,2)	0.12	0.43	**0.007**	**4.21**	0.37	0.003	3.57
V(2,3)	0.10	0.49	0.009	4.35	0.40	**0.003**	**3.52**
V(2,4)	0.08	0.61	0.016	5.02	0.50	0.007	4.16
V(3,1)	0.12	0.47	0.011	4.60	0.39	0.004	3.82
V(3,2)	0.10	0.49	0.009	4.35	0.40	0.003	3.56
V(3,3)	0.08	0.55	0.011	4.56	0.43	0.003	3.56
V(3,4)	0.07	0.67	0.019	5.24	0.54	0.007	4.23
V(4,1)	0.10	0.55	0.014	4.84	0.48	0.007	4.21
V(4,2)	0.08	0.61	0.016	5.03	0.50	0.007	4.19
V(4,3)	0.07	0.67	0.019	5.24	0.54	0.008	4.25
V(4,4)	0.06	0.68	0.018	5.15	0.51	0.004	3.82

Figure 6 Run time of V-cycle and associated FMG on one and two CPUs



When using multigrid as an iterative method, one is usually not interested in MFlop rates, but in reducing the error as fast as possible. We present asymptotic convergence rates of different V-cycles and their runtime on our test platform in Table 1. Convergence rates are determined using a power method with 100 iterations on a 255^3 grid. To make the measurements comparable, the convergence

rate per second and the time to reduce the error by a factor of 10^{-9} were interpolated. For this problem size and machine configuration for example, the $V(2,2)$-cycle is the best choice for one, the $V(2,3)$-cycle for two CPUs.

Figure 6 compares the runtime of an V-cycle for a problem size of 511^3 and the time necessary for the associated Full Multigrid. The V-cycle time includes calculation of both a maximal and Euclidean norm afterwards, which was not done in the FMG. As the additional work for an FMG is mostly done on coarser grids, it takes less than a third longer than the V-cycle.

5 Conclusion

We implemented a highly optimised geometric multigrid method solving Poisson's equation in 3D, tuned for the Itanium 2 processor. For our 3D Red Black Gauss Seidel smoother we achieved speedups around factor 4-5 against a standard high-level language implementation. Most of this factor even could be preserved for the whole multigrid algorithm.

The Itanium 2 has proven to be an adequate platform for scientific computing, especially for loop based iterative algorithms. Moreover its transparent behaviour makes it possible to implement and study different performance optimisation techniques or combinations of them in an unique way.

We have shown that established performance optimisation techniques – cache blocking, memory layout optimisations and software prefetching – can be combined to improve performance further, if their mixture is tailored to the algorithm and the requirements of the executing platform. The core optimisation strategy is therefore valuable for a large class of algorithms on every cache-based architecture supporting a streaming execution pattern, as it is the case for all modern desktop processors. Which techniques are more important, and therefore determine the basic design aspects, and their implementation details will be different from platform to platform; a good choice will only be possible with a thorough knowledge of the target architecture.

References

Brandt, A. (1977) 'Multi-level adaptive solutions to boundary-value problems', *Mathematics of Computation*, Vol. 31, No. 138, pp.333–390.

Briggs, W., Henson, V. and McCormick, S. (2000) *A Multigrid Tutorial*, 2nd ed. Society for Industrial and Applied Mathematics, Philadelphia.

Douglas, C., Hu, J., Kowarschik, M., Rüde, U. and Weiß, C. (2000) 'Cache optimization for structured and unstructured grid multigrid', *Electronic Transactions on Numerical Analysis (ETNA)*, Vol. 10, pp.21–40.

Hackbusch, W. (1985) *Multi-Grid Methods and Applications*, Springer-Verlag, Berlin.

Kowarschik, M. (2004) *Data Locality Optimisations for Iterative Numerical Algorithms and Cellular Automata on Hierarchical Memory Architectures*, SCS Publishing House, Advances in Simulation 13.

Kowarschik, M., Weiß, C. and Rüde, U. (2001) 'DiMEPACK – a cache–optimized multigrid library', in Arabnia, H. (Ed.): *Proc. Int. Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2001)*, Vol. I, Las Vegas, NV, USA, CSREA, CSREA Press, pp.425–430.

Stürmer, M. (2005) *Optimierung des Red-Black-Gauss-Seidel-Verfahrens auf ausgewählten x86-Prozessoren*, Lehrstuhl für Informatik 10 (Systemsimulation), Institut für Informatik, University of Erlangen-Nuremberg, Studienarbeit, Germany.

Stürmer, M., Treibig, J. and Rüde, U. (2006) 'Optimizing a 3D multigrid algorithm for the IA-64 architecture', in Becker, M. and Szczerbicka, H. (Eds.): *Simulations-Technique – 19th Symposium in Hannover, September 2006*, Vol. 16 of *Frontiers in Simulation*, ASIM, SCS Publishing House, pp.271–276.

Trottenberg, U., Oosterlee, C. and Schüller, A. (2001) *Multigrid*, Academic Press, San Diego.

Weiß, C. (2001) *Data Locality Optimisations for Multigrid Methods on Structured Grids*, PhD Thesis, Lehrstuhl für Rechnertechnik und Rechnerorganisation, Institut für Informatik, Technische Universität München, Munich, Germany.