Journal of Computational Science 2 (2011) 130–137

Contents lists available at ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

Efficient multicore-aware parallelization strategies for iterative stencil computations

Jan Treibig, Gerhard Wellein, Georg Hager*

Erlangen Regional Computing Center (RRZE), University of Erlangen-Nuremberg, Martensstr. 1, 91058 Erlangen, Germany

ARTICLE INFO

Article history:
Received 6 April 2010
Received in revised form
10 November 2010
Accepted 17 January 2011
Available online 16 February 2011

Keywords:
Stencil computations
Spatial blocking
Temporal blocking
Wavefront parallelization
Multicore
Simultaneous multi-threading

ABSTRACT

Stencil computations consume a major part of runtime in many scientific simulation codes. As prototypes for this class of algorithms we consider the iterative Jacobi and Gauss-Seidel smoothers and aim at highly efficient parallel implementations for cache-based multicore architectures. Temporal cache blocking is a known advanced optimization technique, which can reduce the pressure on the memory bus significantly. We apply and refine this optimization for a recently presented temporal blocking strategy designed to explicitly utilize multicore characteristics. Especially for the case of Gauss-Seidel smoothers we show that simultaneous multi-threading (SMT) can yield substantial performance improvements for our optimized algorithm on some architectures.

1. Introduction and related work

Stencil computations can be found at the core of many scientific and technical applications based on regular lattices. For the important class of partial differential equation (PDE) solvers they are a key performance factor. This does not only hold for serial applications but is also true for massively parallel large scale multigrid PDE solvers (see, e.g., [1]), where the time-consuming smoothing steps are frequently composed of stencil computations such as red-black Gauss-Seidel or Jacobi schemes.

It has been shown recently [2] that for state-of-the-art multicore architectures a near to optimal stencil implementation requires elaborate tuning, even if more complex temporal blocking techniques are ignored. Conventional temporal blocking performs multiple updates on a small block of the computational domain before proceeding to the next block. Apart from having a machine dependent tuning parameter, this kind of temporal blocking in three spatial dimensions has been found to not deliver performance improvements on conventional x86-based microarchitectures [3] because it generates rather short loops resulting in substantial performance penalties from pipeline start-up effects and, even worse, from excess memory traffic caused by the hardware prefetchers. This is not necessarily true, however, for more specialized designs like, e.g., the Cell processor [4,5]. In [6] it was shown that adapted variants of these temporal blocking techniques, operating on whole lines instead of rectangular small blocks, show very good results on modern architectures also in 3D. Cache oblivious algorithms as proposed by Frigo et al. [7] are hardware independent but come at the cost of irregular block access patterns, which cause many data TLB misses. This was shown in [8] for a 3D lattice Boltzmann (LB) application kernel. For a general overview about optimizations specific to stencil algorithms we refer to [4].

Recently a proof of concept for a wavefront-based shared memory parallelization scheme was first presented [9]. It allows implicit temporal blocking for stencil computations in multicore environments with shared caches. The basic idea is to run multiple wavefronts through the computational domain at the same time but appropriately shifted in space (depending on the stencil used). Each wavefront represents an update step of the lattice and is executed by a single thread. Binding all threads that run successive wavefronts for the same computational domain to a single multicore chip with a shared cache restricts data access to main memory to a load operation for the initial wavefront and a store operation for the final wavefront; all other (intermediate) data accesses can be satisfied from the shared cache. The scheme is tailored to multicore architectures with shared caches – which will be the major design principle for most standard architectures in the years to come – and does not exhibit the drawback of very short loop lengths.

* Corresponding author. Tel.: +49 9131 85 28973; fax: +49 9131 302941.
E-mail addresses: jan.treibig@rrze.uni-erlangen.de (J. Treibig), gerhard.wellein@rrze.uni-erlangen.de (G. Wellein), georg.hager@rrze.uni-erlangen.de (G. Hager).

1877-7503/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.jocs.2011.01.010

Table 1
Test machine specifications. The cacheline size is 64 bytes for all processors and cache levels. Note that the Nehalem EX system was only equipped with half the required memory boards, reducing main memory bandwidth by 50%. The STREAM results were obtained with the triad variant.

Microarchitecture	Intel Core 2	Intel Nehalem EP	Intel Westmere	Intel Nehalem EX	AMD Istanbul
Model	Xeon X5482	Xeon X5550	Xeon X5670	Xeon X7560	Opteron 2435
Clock [GHz]	3.2	2.66	2.93	2.26	2.6
Cores per socket	4	4	6	8	6
SMT threads per core	N/A	2	2	2	N/A
L1 Cache	32 kB	32 kB	32 kB	32 kB	64 kB
Associativity	8	8	8	8	2
L2 Cache	2 × 6 MB (shared)	4 × 256 KB	6 × 256 KB	8 × 256 KB	6 × 512 KB
Associativity	24	8	8	8	16
L3 Cache (shared)	–	8 MB	12 MB	24 MB	6 MB
Associativity	–	16	16	24	48
Bandwidths [GB/s]:					
Theoretical socket BW	12.8	32.0	32.0	17.1	17.1
STREAM 1 thread	4.6	11.9	11.0	5.3	7.2
STREAM socket NT/noNT	4.8/5.6	18.5/23.7	21.0/23.6	9.1/13.6	9.8/11.4

This paper introduces a more generic and flexible implementation of the wavefront technique with an improved spatial blocking scheme and highly efficient synchronization primitives. In addition to the application on Jacobi, already shown in [9], we extend the method to the lexicographic Gauss-Seidel smoother, whose data dependencies require substantial modifications in thread scheduling [10]. We present results for socket and node on a wide range of current multicore architectures. Moreover we show how our technique can leverage SMT threads on Intel processors, increasing the utilization of the floating point units.

2. Experimental testbed

A wide selection of modern x86-based multi-core processors (cf. Table 1) has been chosen to try different variants of the wavefront parallelization strategy and to demonstrate its performance potentials. All of these chips feature a large outer level cache, which is shared by two (Intel Harpertown), four (Intel Nehalem EP), six (Intel Westmere EP, AMD Istanbul) or eight cores (Intel Nehalem EX). The maximum number of cores sharing an outer level L2/L3 cache will be denoted as "L2/L3 group" in the following. The Harpertown processor (implementing the Core 2 architecture) is usually considered as a quad-core chip since four cores are put in a single package (see Fig. 1(a)). However, it is built up from two independent L2 groups without a shared L3 cache and thus we will consider it as two independent dual-core processors, i.e., two L2 groups. The Nehalem EP is Intel's first quad-core chip featuring a shared L3 cache for all four cores (L3 group). In addition a complete redesign of the memory subsystem allowed for a substantial increase in main memory bandwidth at the cost of introducing a ccNUMA architecture for multi-socket servers. The follow-on processor (Westmere) reflects a shrink in transistor size, which allows to increase both the number of cores as well as the L3 cache size by 50% (Fig. 1(b)). Intel also reintroduced simultaneous multithreading (SMT) with the Nehalem architecture, a hardware optimization to improve utilization of execution units. Each core supports two SMT threads. AMD's competitor to Intel Nehalem is the Istanbul processor design, which comes as a six core L3 group and is based on a ccNUMA architecture on the multi-socket node level as well. The 8-core Intel Nehalem EX processor is not mainly targeted at HPC clusters but at large mission-critical servers. Since it already implements a substantially improved cache architecture and can easily be manipulated on the main memory bandwidth side we have included it in this report to simulate future architectural developments. A comprehensive summary of the most important processor features is presented in Table 1. As the two iterative schemes considered in this paper are known to be memory bandwidth intensive we have reported in Table 1 the maximum attainable main memory bandwidth as measured by the STREAM triad benchmark [11] with and without non-temporal stores. In the latter case, the full bus traffic including the write allocate transfer for the store stream is reported. For a more detailed analysis of the memory and cache hierarchy see [12]. Note that the Intel Nehalem EX test system was equipped with only half of the possible number of memory cards, reducing the bandwidth accordingly. The ability of pinning a selected team of threads to a single cache group and determining the cache group topology of a multi-core processor is vital for the parallelization approach described in this report. In this context we use the open-source tool likwid [13], which has been developed in our group.

3. Iterative schemes and baseline performance

Iterative schemes based on regular stencil computations in three spatial dimensions are used in many numerical applications, e.g., linear solvers or multi-grid methods. As prototypes for this class of algorithm we have chosen the well-known Jacobi method for a Poisson problem and the Gauss-Seidel method for a Laplace problem. For reasonably large data sets those methods are known to be data-intensive and the attainable main memory bandwidth imposes an upper limit for performance, which can be modeled rather accurately [2,9]. In this section we will first briefly introduce both schemes, determine an upper performance limit (via main memory bandwidth as given in Table 1), and introduce in case of Gauss-Seidel a simple code optimization to achieve the expected performance number. Those optimal measurements will be the baseline to compare with for the multi-core aware parallelization approach.

The Jacobi scheme in three spatial dimensions can basically be formulated as follows (we use C notation):

```
for(iter=0; iter<iterEnd;iter++) {
 for(k=0; k<Nk; k++) {
 for(j=0; j<Nj; j++) {
  for(i=0; i<Ni; i++) {
  dst[k][j][i]=a * src[k][j][i]+b * (
            src[k][j][i-1]+src[k][j][i+1]+
            src[k][j-1][i]+src[k][j+1][i]+
            src[k-1][j][i]+src[k+1][j][i]);
}}}}
```

Fig. 2 shows the basic update scheme of this kernel. The domain is decomposed into lines (y dimension) and planes (z dimension). The computational kernel updates one line at a time, and the seven point stencil in 3D is mapped to a memory access pattern with five streams, only one of which has to be loaded from memory if three planes fit in the outermost cache level (see Fig. 2 right). The store on the dst array generates an additional data stream unless non-temporal stores are used. We implemented an optimized assembly version of the innermost loop (*line update kernel*), with standard

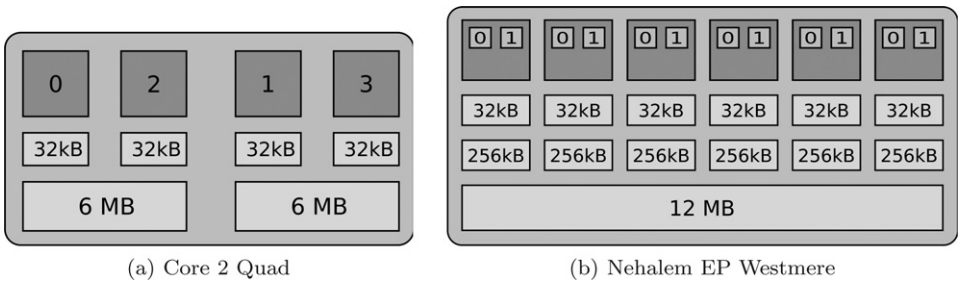

Fig. 1. Cache and thread topology of Intel Core 2 Quad and Nehalem EP Westmere processors.

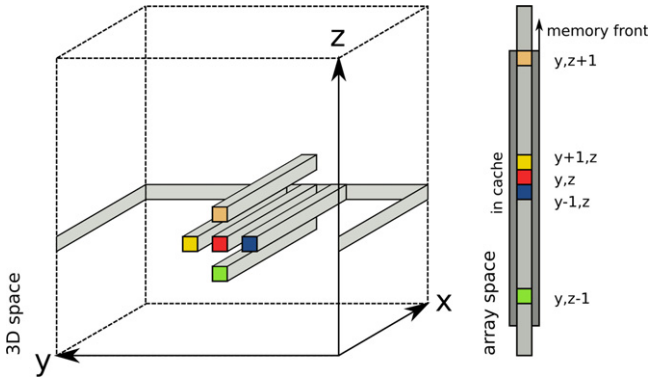

Fig. 2. Stencil structure and corresponding mapping in memory space.

optimizations like, e.g., explicit SSE vectorization, register blocking, and improved instruction scheduling. This subroutine can be used for all parallel variants, since they only modify the processing order of the outer loop nests. Non-temporal stores were employed for in-memory situations only. This ensures results comparable to the optimized variants in [2]. Fig. 3(a) shows the serial baseline performance of the Jacobi kernel against a version in which the line update kernel was written in plain C, for the L2/L3 cache group and main memory. Besides the usual GFlop/s metric we also report performance using the more meaningful lattice site updates per second (LUP/s). The C code was compiled with Intel compiler icc 11.0 and standard optimization flags (-O3 -xW -align -fno-alias). The same executable was used for all machines. Note that the C code could probably reach better performance by applying code transformations or including pragmas/intrinsics, and that the compiler refuses to apply non-temporal stores at all. Still this comparison shows the effectiveness of the optimization techniques independent of their implementation level. As expected the highly clocked but bandwidth-starved Harpertown processor shows the largest drop between in-cache and main memory performance. The in-cache performance for the Nehalem variants is directly correlated with their clock speed. On Nehalem EP and Westmere the small drop between in-cache and main memory performance shows that the serial Jacobi method is not primarily memory bandwidth limited on this machine. The Istanbul, despite its theoretically similar capabilities, shows low performance, and there is no significant difference between hand-optimized and C versions, or in-cache and memory performance. The combination of exclusive caches and large cache latency overhead implies that a major part of the runtime has to be spent transferring data within the cache hierarchy (cf. [12]). Therefore also the applied optimizations do not show a larger effect, and the actual processing of cachelines is only a small part of total runtime. On the other hand, all Intel processors show a high cache efficiency for these bandwidth-demanding data access patterns.

Turning to multi-threaded execution on a single socket one can assume that main memory bandwidth is saturated. Following above discussion, the minimum data transfer between main memory and the cache hierarchy for a single cell update is one load and one store. In this case a simple model for the maximum performance can be set up (for double precision):

$$P_0 = \frac{M_S}{16\,\text{bytes}}\,[\text{LUP/s}] \tag{1}$$

The computer's attainable main memory bandwidth (M_S) can be measured, e.g., with the threaded STREAM triad benchmark. This simple approach is known to provide a good upper performance limit for memory bandwidth limited situations.

We estimate the potential performance gain for temporal blocking by benchmarking the saturated L2/L3 cache group performance with a dataset that fits in the outer cache level. The larger the difference between in-cache and main memory performance, the higher the expected performance improvement by temporal blocking. For

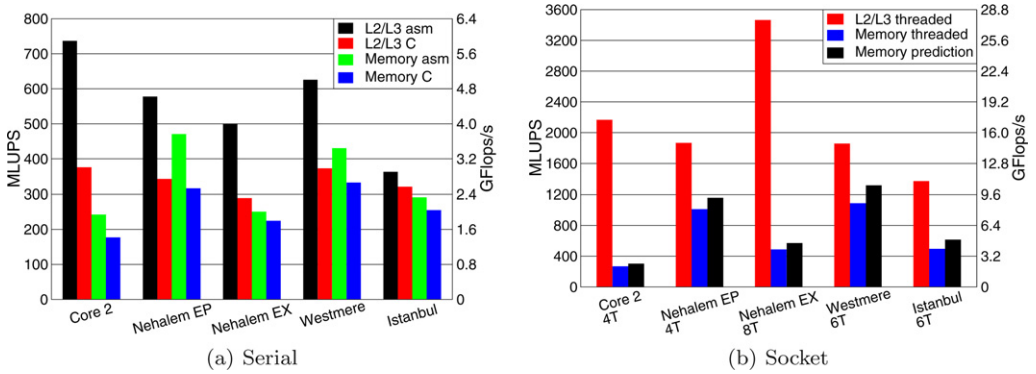

Fig. 3. Jacobi baseline performance in GFlop/s and MLUP/s (million lattice site updates per second) for C language (C) and assembly (asm) kernels, comparing memory and outer-level cache performance. Domain sizes were chosen as $100 \times 50 \times 50$ (4 MB data set fitting in the outermost cache level) and $400 \times 200 \times 200$ (256 MB data set to be loaded from main memory), respectively. For all machines all physical cores of a socket were utilized.

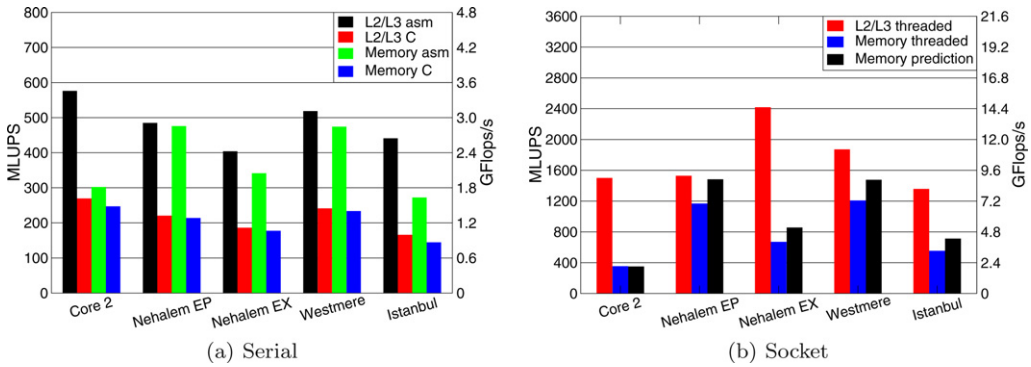

Fig. 4. Gauss-Seidel baseline performance. Note that the C implementation does not employ the dependency optimization described in the text. Same data sets as in Fig. 3.

a more elaborate prediction based on a diagnostic performance model, cf. [14]. Fig. 3(b) shows the saturated L2/L3 cache group and saturated main memory performance, together with the limit predicted by (1). Nehalem EP and Westmere show a more balanced performance between main memory and cache. Hence they are expected to benefit less from our optimizations. The measurements reveal that while Westmere clocks higher, the uncore (L3 cache and memory controller) has the same clock speed as Nehalem EP and therefore reaches similar in-cache performance, despite its two additional cores. Nehalem EX introduces a novel segmented L3 cache, which shows a near to perfect bandwidth scalability with the number of cores. The in-memory results are well in line with the performance limits predicted based on the STREAM triad sustained bandwidth as listed in Table 1.

The Gauss-Seidel method as opposed to Jacobi performs an in-place update. It can be formulated as follows (using C notation):

```
for(iter=0; iter<iterEnd;iter++) {
 for(k=0; k<Nk; k++) {
 for(j=0; j<Nj; j++) {
  for(i=0; i<Ni; i++) {
  src[k][j][i]=    b * (src[k][j][i-1] + src[k][j][i+1] +
                   src[k][j-1][i]+src[k][j+1][i] +
                   src[k-1][j][i]+src[k+1][j][i]);
}}}}
```

Gauss-Seidel looks similar to Jacobi at first glance in terms of stencil structure and data transfer volumes. A difference apparent at once is that SIMD vectorization is ruled out because of the recursive structure on the central row. The in-place update prevents optimal pipelining, thus Gauss-Seidel performance is inferior to Jacobi despite comparable data transfer volumes and less computations (Figs. 3 and 4). Additionally there is no substantial drop between in-cache and memory performance. This is especially remarkable on the Core 2 processor and clearly indicates that pipelining problems limit the achievable performance. To overcome these problems, our optimized kernel interleaves two updates in order to break up register dependencies and partially hide the recursion. The optimized version implements the following adjusted update scheme:

```
for(iter=0; iter<iterEnd;iter++) {
 for(k=0; k<Nk; k++) {
 for(j=0; j<Nj; j++) {
 tmp1=          src[k][j][2] +
                src[k][j-1][1] + src[k][j+1][1] +
                src[k-1][j][1] + src[k+1][j][1];
 for(i=1; i<Ni-1; i++)
 tmp2=          src[k][j][i+1] +
                src[k][j-1][i] + src[k][j+1][i] +
                src[k-1][j][i] + src[k+1][j][i];
 src[k][j][i]=b * (src[k][j][i-1]+tmp1);
 tmp1=tmp2;
}}}}
```

Fig. 4(a) shows the serial Gauss-Seidel results. A large part of the speedup between the optimized assembly implementation as compared to the C version can be attributed to this Gauss-Seidel specific code reordering. The small drop from in-cache to memory performance indicates that the sequential Gauss-Seidel on Nehalem EP and Westmere is not memory bandwidth limited anymore. Only on the bandwidth-starved Harpertown there is still a significant drop between in-cache and main memory domain. Istanbul shows a much more competitive performance for the optimized code. The data transfers are still inefficient, but the optimizations can show their effect and the impact of the inefficient caches is smaller because now the L1 runtime part is much larger. Westmere benefits from its two additional cores compared to Nehalem EP indicating that the L3 bandwidth is not fully saturated by four threads.

A further problem connected to the recursive nature of Gauss-Seidel is that a straightforward parallelization based on domain decomposition cannot be employed. A common solution is to use the Red-Black Gauss-Seidel method instead, which can be easily parallelized. We choose another option to parallelize the standard lexicographic Gauss-Seidel method based on a pipeline parallel approach (see Fig. 5(a)). It retains the same algorithm as a sequential Gauss-Seidel sweep. Each thread operates on a sub-block. Plane updates of threads are shifted in time to retain the correct update order. The threaded socket results for Gauss-Seidel are illustrated in Fig. 4(b). Compared to the Jacobi solver, the parallel Gauss-Seidel algorithm is still limited by the loop-carried dependencies in the kernel, which leads to a smaller performance difference between in-cache and memory situations for all but the Westmere and Istanbul processors. Westmere can still hit the bandwidth limitations due to its larger core count, while Istanbul is known to be restrained by the large overheads for cache line transfers [12], making the inefficient pipelining less dominant.

Since non-temporal stores cannot be applied, we use STREAM triad measurements with standard stores (Table 1) in the performance model (1) for Gauss-Seidel.

4. Temporal blocking through multi-core aware wavefront parallelization

Our wavefront parallelization technique implements implicit temporal blocking by utilizing the property of modern multicore architectures that multiple cores share the outermost cache level. The grid is decomposed into blocks. A block is updated by a “thread group,” consisting of a number of threads. Each thread in a thread group performs one sweep on the block, successively updating the “planes” in z direction. Planes updated by consecutive threads are guaranteed to be still located in the shared cache. This update mechanism is illustrated in Fig. 6. Because multiple updates are performed while holding the data in cache and the intermediate update steps need not be stored, the second grid of the out-of-place

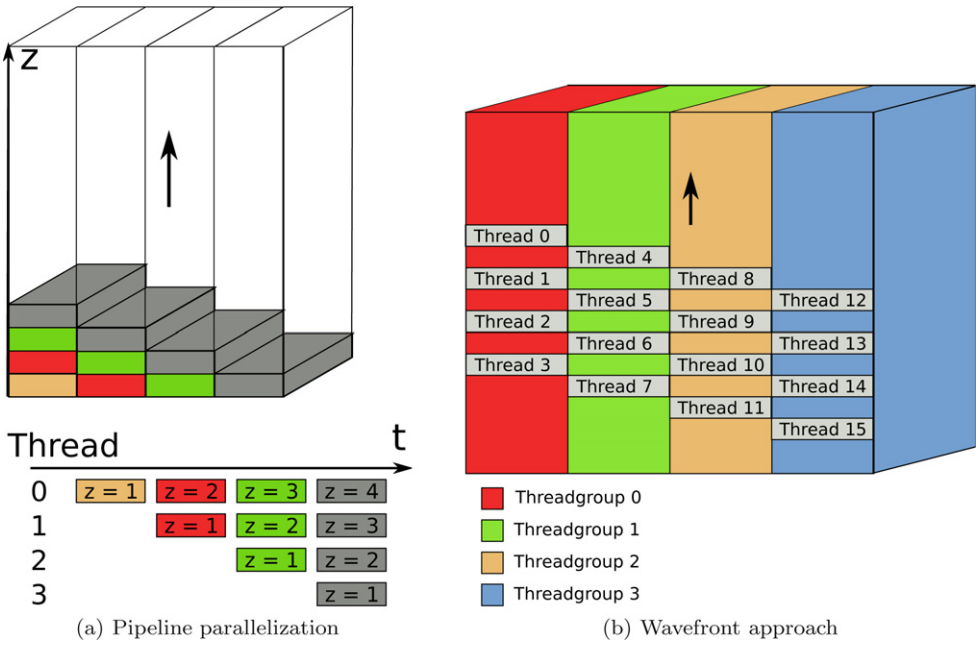

Fig. 5. Parallelization of the Gauss-Seidel algorithm by pipelied parallel execution (a) and the wavefront approach (b).

Jacobi method is not required. Odd-numbered updates are instead written to a temporary array, which is large enough to hold the intermediate steps of all updates until the last update is written back to the src array. In step (a) the first thread of a group performs the first plane update, loading from src and writing to a small temporary array. With a distance of two to ensure the correct update order, the second thread performs the second update from the temporary array back to the src array in step (b). In our example with four threads another two updates are performed until the fourth update is written to the src array. The temporary array is used in a round robin manner, and hence shifted through the z dimension of the spatial grid. It must be large enough to hold the needed dst planes of all threads (eight in our example). The threads have to be synchronized after each plane update, and block sizes must be chosen so that the temporary data can be kept in the outermost cache level. Since the number of thread groups, the number of threads per thread group, and the block size can be freely configured, this scheme can map to the underlying hardware in a very flexible way. A drawback of the current implementation is that the maximum number of blocked updates is determined by the number of threads

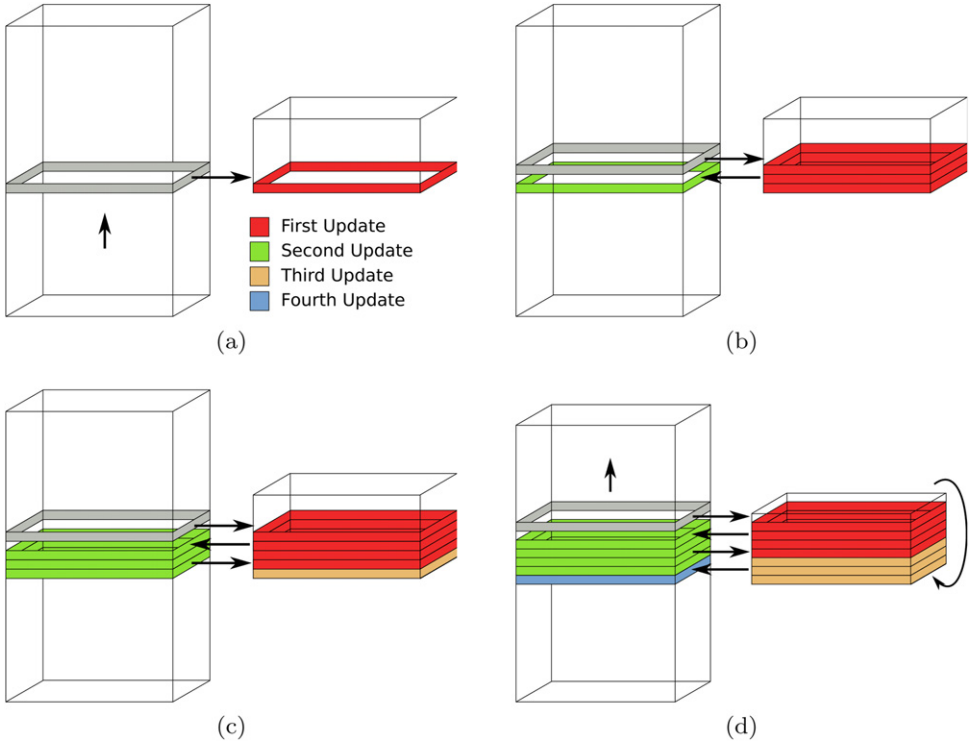

Fig. 6. Temporal wavefront blocking (1 thread group with four threads).

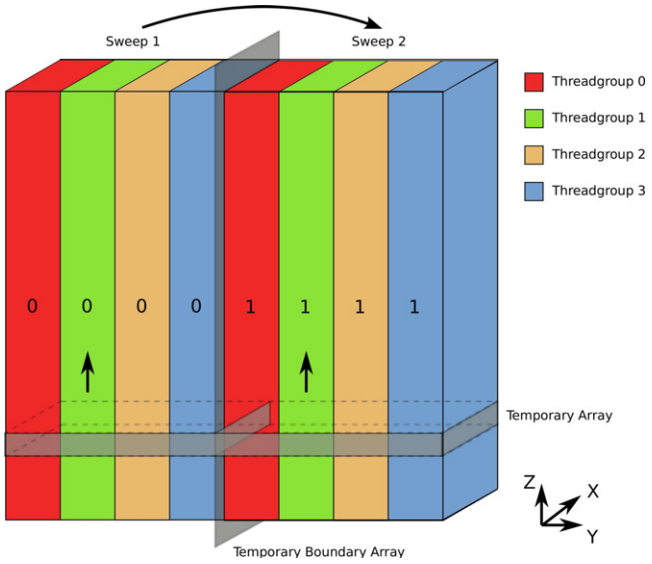

Fig. 7. Organization of thread groups with spatial blocking.

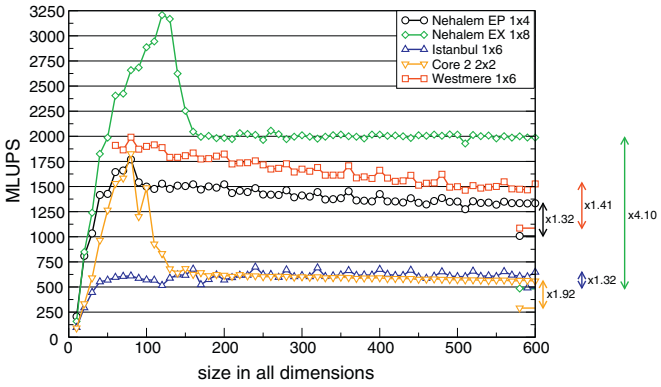

Fig. 8. Wavefront temporal blocking results for the Jacobi smoother.

available. This method requires fine-grained parallelism, making it crucial to employ an efficient synchronization mechanism. Our threaded implementation is based on the POSIX thread API. The pthread barrier turned out to be unsuitable due to a very large overhead. For small thread counts, as applicable on a single socket, an implementation of a spin waiting loop was used for the barrier. Since this does not perform well with SMT threads, a tree barrier was implemented which provided less overhead whenever more than one logical thread per core was used.

The spatial blocking scheme is illustrated in Fig. 7. Every thread group performs a parallel wavefront update as explained above. The domain is decomposed into B blocks along the y dimension (eight in Fig. 7). Each thread group works on one or more blocks. All thread groups update each block in a synchronized fashion, and one z sweep is performed on the first N blocks, where N is the number of thread groups. The boundary between consecutive sweeps must be set up so that the next sweep can proceed with correct boundary conditions on the interface to the previous sweep. If t is the number of threads in a group, a boundary array must thus hold t planes in z–x direction. Hence no additional computations are necessary for the boundary treatment.

Fig. 8 shows the results for Jacobi with temporal wavefront blocking on one socket. The baselines drawn on the right are the threaded results without temporal blocking for a problem size of $200 \times 200 \times 200$. On Core 2 there is a very large gap between in-cache and in-memory memory baselines, indicating a considerable potential for temporal blocking. A speedup of two could be achieved by wavefront temporal blocking, but to leverage the potential on this architecture a bigger blocking factor (i.e., more cores per thread group) would be required in order to decouple from main memory bandwidth. A completely different picture can be seen on the Nehalem EP. Here the gap between in-cache and memory performance is rather small. The threaded memory performance utilizing non-temporal stores is already 1008 MLUPS. While a speedup of 25–50% seems fair, the scaling of memory bandwidth with the number of cores results in a high baseline and limits the benefit of our optimization. The result on the Westmere processor is similar, still there is a speedup benefit from the higher blocking factor of six. The Intel Nehalem EX in the configuration used here shows the highest benefit. It allows a blocking factor of eight together with a very high bandwidth L3 cache. On the other hand, this (artificial) configuration is strongly bandwidth-starved in terms of available bandwidth per core. This combination results in speedups of four independent of the problem size. Istanbul has a good initial position promising a significant speedup, there is a large gap between saturated in-cache and main memory performance, and the number of cores per socket allows a blocking factor of six. However, it only achieves speedups comparable to Nehalem EP.

Fig. 9(a) shows the intranode scaling from one to two sockets. As expected, the threaded version scales almost perfectly on all ccNUMA architectures. Still also the Harpertown Core 2 machine performs well for two sockets. The wavefront versions show a reasonable scaling on the Intel processors. Because of the temporal fields there is some data traffic between sockets involved, preventing a better scaling. This overhead is larger on Nehalem EP and Westmere than on the Nehalem EX machine. On Istanbul there is nearly no benefit for the wavefront version on the node level against the standard threaded version. While the wavefront optimization shows significant benefit also on the node level, it is usually best to employ hybrid (MPI+OpenMP) parallelization with one process

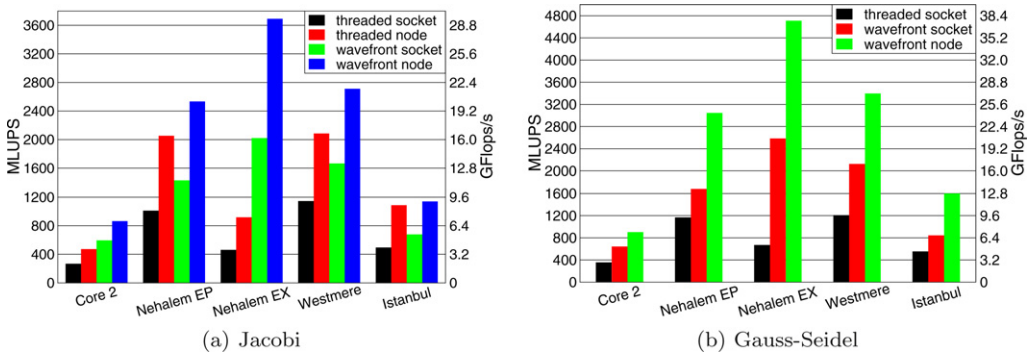

Fig. 9. Scaling of threaded and wavefront blocked versions from one to two sockets for Jacobi (a) and Gauss-Seidel (b).

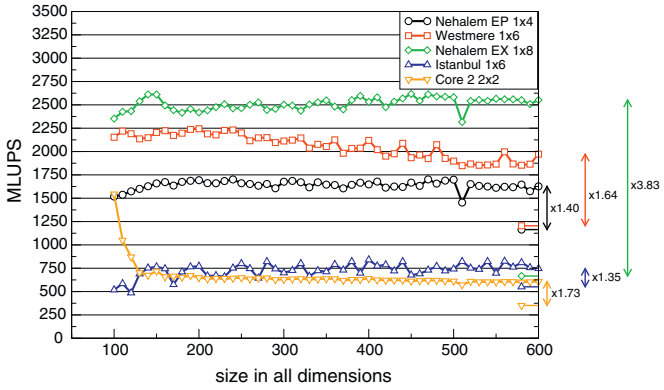

Fig. 10. Wavefront temporal blocking results for the Gauss-Seidel smoother.

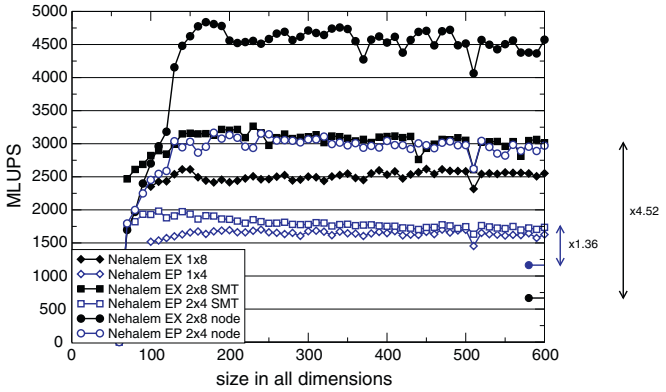

Fig. 11. Wavefront temporal blocking results for the Gauss-Seidel smoother with SMT. (Thread groups: NehalemEP with 1×4 (socket) and 2×4 (socket SMT, node), Nehalem EX with 1×8 (socket) and 2×8 (socket SMT, node).

per socket and apply wavefront blocking only within the socket. This has been demonstrated in Ref. [14], where it was also shown under which conditions it is possible to maintain an intra-process performance improvement in a hybrid MPI+OpenMP setting via multilayer halo exchange.

Our scheme for temporal wavefront parallelization can be adapted for the in-place Gauss-Seidel method if used in combination with the pipelined parallel approach. Since all updates operate on one array, additional temporary arrays are not required. To ensure the correct update ordering the updates have to be shifted for lexicographic Gauss-Seidel between thread groups as illustrated in Fig. 5(b). This is a natural extension of the threaded pipelined parallelization introduced in Section 3. The results for temporal wavefront blocking with Gauss-Seidel are shown in Fig. 10. Again the baseline of a threaded Gauss-Seidel with pipeline parallelization for a size of $200 \times 200 \times 200$ is indicated on the right y axis. On the Core 2 the combination of low (and non-scaling) main memory performance together with only a blocking factor of two leads to a speedup of nearly two for temporal blocking. Nehalem EP shows improvements of 30–40%. Westmere profits from its two additional cores and hence from the deeper blocking factor, reaching a speedup of over 50%. The potential of our optimization technique is again seen on the example of Nehalem EX. Despite its hardware configuration yielding the lowest bandwidth compared to the other Nehalem processors, it reaches the best overall performance and can fully benefit from its eight cores and strong L3 cache subsystem, showing an impressive speedup factor of 3.8. The Istanbul architecture again shows disappointing results, comparable to the Nehalem EP. Its exclusive cache hierarchy seems to be unsuited for these bandwidth-demanding in-cache loops, but the exact reason for the small gains of our optimization were not investigated in more detail.

In Fig. 9(b) the node scaling for Gauss-Seidel is shown. The Intel processors behave simliar to Jacobi with reasonable scaling from one to two sockets. Overhead on Nehalem EP and Westmere is again larger than on Nehalem EX.

As noted in Section 3, Gauss-Seidel cannot be fully pipelined due to its recursive structure. While our optimized assembly kernel reduces this penalty, it is still noticeable, causing the floating point units to be underutilized. For exactly this situation (a shared resource being not fully used) the Nehalem processors implement simultaneous multithreading (SMT) with two hardware threads sharing a physical core. Further possible benefits of using SMT threads for our temporal wavefront optimization are manifold, deeper blocking factors being just one effect. Since main memory bandwidth on the Nehalem (EP and Westmere) processors scales with the number of threads, the utilization of the memory bus can be increased using multiple thread groups. And finally, two SMT threads also share the L1 and L2 caches, potentially reducing necessary cacheline transfers in our pipelined wavefront setting. One drawback of using SMT on current multicore architectures apart from an expensive thread synchronization is the additional pressure on the L3 bandwidth which is now shared by twice the number of threads. Using SMT threads needs no changes in the implementation, but proper affinity of threads to resources must be observed. Without SMT, each thread in the group is bound to a physical core. With SMT, two thread groups are used: Each thread of the first group is pinned to the first SMT thread of each core, while each thread of the second group uses the second SMT thread on the same set of physical cores. Fig. 11 shows the results, with square symbols denoting SMT data, for which one thread was scheduled per SMT thread (two per physical core). The configuration with two thread groups shows the best performance. While on the Nehalem EP the gain is relatively small, Nehalem EX shows the expected improvement from SMT. These results confirm that today's multicore architectures require high-bandwidth shared caches as well as high-bandwidth memory interfaces for fully exploiting their computational capabilities. In this respect, the segmented cache on the Nehalem EX proves to be superior to the simpler design on Nehalem EP.

5. Conclusion

We have presented a novel way to implement temporal blocking, specifically designed to leverage the shared outer-level cache on today's multicore architectures. The optimizations were evaluated on a wide range of current multicore processors to show their potential. Besides the well-known Jacobi method we have presented a highly efficient implementation of the recursive Gauss-Seidel method. For the threaded and temporal wavefront implementation of Gauss-Seidel we employed a pipeline parallel approach, retaining the update ordering of the serial algorithm. The optimization reaches considerable speedups on all architectures. Results on our (artificially) bandwidth-starved Nehalem EX system confirm that a large ratio between in-cache and memory bandwidths improves the gain for our temporal blocking approach. The cache subsystem of the AMD Istanbul turned out to be incapable of benefitting from our optimizations to the same extent as comparable Intel architectures. Finally we have shown that employing SMT threads for temporal blocking of the Gauss-Seidel solver yields a substantial performance improvement, with overall speedups of up to five on Intel Nehalem EX. Our measurements show that for multicore designs incorporating a shared last level cache, a large, scalable cache bandwidth is instrumental for the overall performance of highly optimized, temporally blocked stencil smoothers.

Acknowledgments

We are indebted to Intel Germany for providing test systems and early access hardware for benchmarking. This work was supported by the Competence Network for Scientific High Performance Computing in Bavaria (KONWIHR) via project OMI4PAPPS.

References

[1] B. Bergen, F. Hülsemann, U. Rüde, Is 1.7×1010 unknowns the largest finite element system that can be solved today? in: ACM/IEEE (Ed.), Proceedings of the ACM/IEEE SC 2005 Conference Supercomputing Conference '05, Seattle, November 12–18, 2005, 2005.
[2] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, K. Yelick, Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures, in: ACM/IEEE (Ed.): Proceedings of the ACM/IEEE SC 2008 Conference Supercomputing Conference'08, Austin, TX, November 15–21, 2008, 2008.
[3] M. Kowarschik, Data Locality Optimizations for Iterative Numerical Algorithms and Cellular Automata on Hierarchical Memory Architectures. Ph.D. Thesis, July 2004, SCS Publishing House, Germany, 2004, ISBN 3-936150-39-7.
[4] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, K. Yelick:, Optimization and performance modeling of stencil computations on modern microprocessors SIAM Rev. 51 (1) (2009) 129–159.
[5] M. Christen, O. Schenk, P. Messmer, E. Neufeld, H. Burkhart:, Parallel data-locality aware stencil computations on modern micro-architectures, in: Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS), May 25–29, Rome, Italy, 2009.
[6] J. Treibig, Efficiency Improvements of Iterative Numerical Algorithms on Modern Architectures. Ph.D. Thesis, July 2009, URN: urn:nbn:de:bvb:29-opus-14036.
[7] M. Frigo, C.E. Leiserson, H. Prokop, S. Ramachandran:, Cache-oblivious algorithms, in: 40th Annual Symposium on Foundations of Computer Science, FOCS 99, October 17–18, New York, NY, 1999.
[8] T. Zeiser, G. Wellein, A. Nitsure, K. Iglberger, U. Rüde, G. Hager:, Introducing a parallel cache oblivious blocking approach for the lattice Boltzmann method, Prog. CFD 8 (1–4) (2008) 179–188.
[9] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, H. Fehske, Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization, in: Proc. COMPSAC 2009, 2009, doi:10.1109/COMPSAC.1.2009.82.
[10] G. Hager, F. Deserno, G. Wellein, Pseudo-vectorization and RISC optimization techniques for the Hitachi SR 8000 architecture, in: S. Wagner, et al. (Eds.), High Performance Computing in Science and Engineering Munich 2002, Springer, 2003, pp. 425–442.
[11] J.D. McCalpin, STREAM: Sustainable Memory Bandwidth in High Performance Computers. http://www.cs.virginia.edu/stream.
[12] J. Treibig, G. Hager:, Introducing a performance model for bandwidth limited loop kernels, in: Workshop on Memory Issues on Multi- and Manycore Platforms, PPAM, 2009.
[13] J. Treibig, G. Hager, G. Wellein, LIKWID: a lightweight performance-oriented tool suite for x86 multicore environments, PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures, San Diego CA, September 13, 2010. arXiv:1004.4431, doi:10.1109/ICPPW.2010.38, in press.
[14] M. Wittmann, G. Hager, J. Treibig, 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.

Jan Treibig holds a Ph.D. in Computer Science from the University of Erlangen-Nuremberg. From 2006 to 2008 he was a software developer and quality engineer in the embedded automotive software industry. Since 2008 he is a research scientist in the HPC Services group at Erlangen Regional Computing Center (RRZE). His main research interests are low-level and architecture-specific optimization, performance modeling, and tooling for performance-oriented software developers.

Gerhard Wellein holds a Ph.D. in Solid State Physics from the University of Bayreuth and is a regular Professor at the Department for Computer Science at University of Erlangen. He heads the HPC group at Erlangen Regional Computing Center (RRZE) and has more than ten years of experience in teaching HPC techniques to students and scientists from Computational Science and Engineering. His research interests include solving large sparse eigenvalue problems, novel parallelization approaches, performance modeling and architecture-specific optimization.

Georg Hager holds a Ph.D. in Computational Physics from the University of Greifswald. He has been working with high performance systems since 1995, and is now a senior research scientist in the HPC group at Erlangen Regional Computing Center (RRZE). Recent research includes architecture-specific optimization for current microprocessors, performance modeling on processor and system levels, and the efficient use of hybrid parallel systems. His daily work encompasses all aspects of user support in high performance computing like lectures, tutorials, training, code parallelization, profiling and optimization, and the assessment of novel computer architectures and tools.