Parallel Processing Letters

DOMAIN-SPECIFIC OPTIMIZATION OF TWO JACOBI SMOOTHER KERNELS AND THEIR EVALUATION IN THE ECM PERFORMANCE MODEL

STEFAN KRONAWITTER[†] and HOLGER STENGEL[‡] and
GEORG HAGER[‡] and CHRISTIAN LENGAUER[†]

[†] *Department of Informatics and Mathematics, University of Passau*
Innstraße 33, 94032 Passau, Germany

[‡] *Erlangen Regional Computing Center (RRZE)*
Martensstraße 1, 91058 Erlangen, Germany

Received April 2014
Revised July 2014

Communicated by Guest Editors

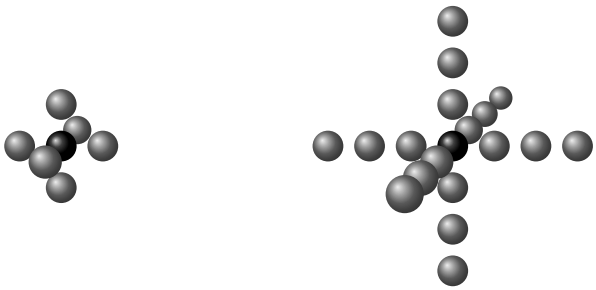
ABSTRACT

Our aim is to apply program transformations to stencil codes in order to yield the highest possible performance. We recognize memory bandwidth as a major limitation in stencil code performance. We conducted a study in which we applied optimizing transformations to two Jacobi smoother kernels: one 3D 1st-order 7-point stencil and one 3D 3rd-order 19-point stencil. To obtain high performance, the optimizations have to be customized for the execution platform at hand. We illustrate this by experiments on two consumer and two server architectures. We also verified the need for complex optimizations with the help of the Execution-Cache-Memory performance model. A code generator with knowledge about stencil codes and execution platforms should be able to apply our transformations automatically. We are working towards such a generator in project ExaStencils.

Keywords: Jacobi smoothers, high-performance computing, program transformations, stencil codes, ECM performance model

1. Introduction

Multigrid methods [17] are widely used in scientific applications, especially in physics and chemistry simulations. Much of the time consumed by a multigrid algorithm is spent in its smoother. This paper is not about multigrid, but the center of our study are two Jacobi smoothers used in multigrid methods [2]: one for a 3D 1st-order 7-point smoother and one for a 3D 3rd-order 19-point smoother. Both smoothers use constant coefficients and a single output and two input grids. The three grids are located in distinct memory areas.



(a) 1st-order stencil (b) 3rd-order stencil

Fig. 1. Footprint of two 3D Jacobi stencils.

```
for (int x = 1; x < dimX-1; ++x)
 for (int y = 1; y < dimY-1; ++y)
  for (int z = 1; z < dimZ-1; ++z)
    out[x][y][z] = a * in[x][y][z]
        + b1 * (in[x][y][z-1] + in[x][y][z+1] + in[x][y-1][z]
              + in[x][y+1][z] + in[x-1][y][z] + in[x+1][y][z])
        -  c * rhs[x][y][z];
```

Fig. 2. Kernel code of a 3D 1st-order Jacobi smoother.

```
for (int x = 3; x < dimX-3; ++x)
 for (int y = 3; y < dimY-3; ++y)
  for (int z = 3; z < dimZ-3; ++z)
    out[x][y][z] = a * in[x][y][z]
        + b1 * (in[x][y][z-1] + in[x][y-1][z] + in[x-1][y][z]
              + in[x][y][z+1] + in[x][y+1][z] + in[x+1][y][z])
        + b2 * (in[x][y][z-2] + in[x][y-2][z] + in[x-2][y][z]
              + in[x][y][z+2] + in[x][y+2][z] + in[x+2][y][z])
        + b3 * (in[x][y][z-3] + in[x][y-3][z] + in[x-3][y][z]
              + in[x][y][z+3] + in[x][y+3][z] + in[x+3][y][z])
        -  c * rhs[x][y][z];
```

Fig. 3. Kernel code of a 3D 3rd-order Jacobi smoother.

In the 3D 1st-order smoother kernel, the update of a single element requires the old input value along with the nearest neighbors in each direction, and a single corresponding point of the second input array. Figure 1(a) depicts the 3D footprint of this stencil. The source code of the corresponding kernel is shown in Figure 2.

In the 3D 3rd-order 19-point smoother kernel, the update of a single element requires not only the nearest neighbors but all elements in a range of three points in each direction. Figure 1(b) depicts the 3D footprint of this stencil. The source code of the corresponding 3D kernel appears in Figure 3.

The direct implementations of the codes in Figures 2 and 3 have poor performance (see below for details). Suitable optimizations can improve the performance considerably, yet, they are not applied automatically by present-day compilers. To tackle this problem we present transformations, some of which exploit domain knowledge. Our intention is to let them be applied automatically by a domain-specific optimizing compiler. We are working towards such a compiler in project ExaStencils[a], which is part of the Priority Research Initiative SPPEXA[b], funded by the German Research Foundation (DFG).

A serious limiting factor for lower-order stencils turns out to be the memory bandwidth, since the computation of a single result value requires loading almost as many values from memory as floating-point operations are performed and the overall number of instructions that need to be executed by the processor is low. For higher-order kernels the CPU needs to process a larger amount of instructions for a single lattice update, as many more data elements must be retrieved from the memory hierachy. Therefore, the memory bandwidth is not the only limiting factor and simpler optimizations may achieve similar performance improvements. We employ different blocking techniques to reduce the bandwidth requirements of our codes. The evaluation of our optimizations show that modern x86 CPUs require a more complex blocking technique than IBM's BlueGene/Q.

In summary, we make the following contributions:

- a set of six optimizing transformations that can be applied automatically to stencil codes,
- an empirical evaluation of the optimizations applied to two codes on four architectures,
- a more detailed, model-based, evaluation of the performance for an x86 server processor,
- insights into the suitability of the various optimizations for the individual architectures,
- insights into the basic limitations of stencil code performance.

Our work extends a preliminary study [12], with enhanced optimizations and an additional performance evaluation on a new server processor using our Execution-Cache-Memory (ECM) model [8, 15].

The rest of the paper is organized as follows. Section 2 describes six different optimizations, which can be used to optimize stencil codes and to achieve high performance. Section 3 presents an empirical performance analysis of the presented optimizations and Section 4 uses our ECM model to evaluate the optimization techniques.

[a] `http://www.exastencils.org`
[b] `http://www.sppexa.de`

2. Optimizations

This section describes six optimization techniques, which are sufficient to achieve high performance on all test platforms. Their performance is discussed in the next section.

We choose three basic optimizations, which can be applied safely to improve the performance of the generated target code. These optimizations are not domain-specific. They improve the performance on any architecture and do not exploit knowledge about stencil codes. In particular, they do not alter the order of the kernel iterations and do thus not change the data access characteristics. They are described in Subsection 2.1. Subsection 2.2 describes three further, domain-specific optimizations. They do affect the order of the kernel iterations which may reduce the pressure on the bus from processor to main memory. The effect is that different architectures favour different optimizations.

2.1. *Basic transformations*

One of the simplest optimizations is to remove complex address computations from the innermost loop of the kernel. Consider the source code in Figure 2. If each array is stored linearly in memory, a 3D array access `in[x][y][z]` results in the polynomial index expression

```
*(in+(x*dimY+y)*dimZ+z).
```

For each iteration of the two loops on `x` and `y`, the values of `x`, `y`, `dimY` and `dimZ` remain constant. Thus,

```
in_p = in+(x*dimY+y)*dimZ
```

can be precomputed before entering the innermost loop. Then, the 3D access `in[x][y][z]` can be replaced by `in_p[z]`, which is computed much more easily. This technique of precomputing the constant values in the index polynomial for access of elements of a multidimensional array has been standard fare in compiler classes for decades [1]. However, it is not always applied by a compiler, e.g., by gcc 4.8, since different other, previous transformations can stand in the way of this optimization. We observed that gcc is not able to perform this transformation after the code has been vectorized manually via vector intrinsics. Consequently, we applied it explicitly to the innermost loop, which is in every language laid out contiguously in memory – even in Java. Figure 4 illustrates this optimization for a single 3D array, regardless of whether it is linearized in memory or accessed using a 3D pointer `double ***in`.

Another optimization beneficial for higher-order stencils would be to reorder the computations such that an entire cache line is used at the same time. This prevents a repeated load of the same data into L1 cache [6]. Consider again the stencil of Figure 1(b). Let us assume that the data is stored in row-major order in memory (as, e.g., in C) and that the required neighbors in the same row can be loaded from

```
for (int x = 0; x < dimX; ++x)
  for (int y = 0; y < dimY; ++y) {
    double *in_p = &in[x][y][0];
    for (int z = 0; z < dimZ; ++z)
      use(in_p[z]);
  }
```

Fig. 4. Simplified address calculation for the innermost loop.

contiguous addresses, while the elements from the same column have a larger stride. If this stride is a power of 2 and the set associativity of the processor's L1 cache is n, only n different blocks are available to hold the cache lines for the neighboring elements. Therefore, a cache line could be evicted, although subsequent elements in it are needed in the next iteration: unnecessary cache misses result. To prevent them, contiguous input data for multiple output elements can be stored in CPU registers at the same time, while the computations are reordered to process the loaded data early.

This register blocking can also be viewed as a preparatory step for vectorizing the kernel, our third basic optimization. Almost all modern processors have SIMD units, which can be used to perform the same computation in parallel on multiple data elements. Current x86 processors support the Advanced Vector Intrinsics (AVX) or, at least, the Streaming SIMD Extensions (SSE). AVX provides 256-bit registers, which can be used to load, store and process eight single-precision or four double-precision floating-point values in parallel. In contrast, SSE provides only 128-bit registers, which can also be used for both single-precision and double-precision values. Another common architecture in high-performance computing is the BlueGene/Q. Each of the 16 cores of a BlueGene/Q chip has a 256-bit vector unit but, in contrast to Intel's AVX, IBM's Quad Processing Extension (QPX) performs all operations with double precision, which restricts the possible vector length to 4.

Today, most compilers provide automatic vectorizers but, in many situations, an explicit vectorization can yield higher performance. The corresponding compiler intrinsics can be used to declare and work with appropriate vector types. Not only must one select the suitable intrinsics carefully, one must also make sure that all input data is aligned correctly if the architecture does not support unaligned load and store operations for vector types, as is the case with BlueGene/Q. For multi-dimensional arrays, the programmer must also choose the width of the innermost dimension carefully: if it is not a multiple of the vector length, it must be padded to ensure that all lines are aligned properly. When using normal store instructions, the hardware has to load a cache line from main memory before it can be modified, which increases the bandwidth requirements. This so-called write-allocate is avoided by the use of nontemporal stores on x86 processors, wherever meaningful, when the test code is vectorized.

2.2. *Domain-specific transformations*

Domain-specific optimizations of stencil codes modify the iteration domain of the kernel. Since every data element is accessed repeatedly during the computation, one simple optimization is to reorder the iterations such that each grid point is processed fully during one stay in the processor's cache. An approach commonly used is to divide the output grid into axis-aligned blocks and load all input data required to compute a single block in the highest-level cache. Then, only data points near the border of the blocks need to be loaded repeatedly.

Let us call a 2D subset of a 3D grid block a plane. The simple 3D blocking scheme just described can be improved by exploiting the fact that, for the 1st-order kernel, only three and, for the 3rd-order kernel, seven neighboring planes must reside in cache at the same time. Thus, the outermost dimension does not need to be blocked explicitly. In addition, it is beneficial to choose a large enough block size for the innermost loop. This ensures that, e.g., the processor's prefetcher can hide the memory latency. Each time the computation switches to a new block, the streaming prefetcher requires a few iterations to adapt to the new data streams until the highest possible performance can be reached. Nguyen et al. call this technique 2.5D blocking [14].

Another possibility to increase the performance is to combine multiple applications of the kernel code [18]. Since the results of all kernel invocations except the last one are temporary and are not needed after the final result is computed, it is sufficient to store only few planes in a temporary buffer. E.g., if two applications of a 1st-order Jacobi smoother are being combined, the buffer for the temporary results must be large enough to hold no more than three planes, as depicted in Figure 5. In a warm-up phase, the first two planes of the initial smoother application are computed and stored in the buffer. In the main phase, a third plane is computed, which is then stored in the next buffer slot, overwriting the oldest entry. Using all three temporary planes, one result plane can be computed and stored in the output grid. This is repeated until the last plane of the input has been processed.

If the cache is large enough to hold the complete buffer, the amount of data

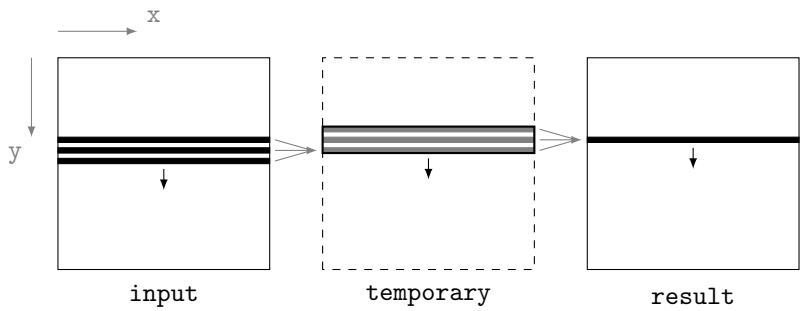


Fig. 5. 2D representation of temporal blocking for a 1st-order Jacobi smoother.

that must be transferred across the bus from the processor to main memory can be reduced. For large grids, even three planes may require too much memory since the cache size is typically only a few MB. But the amount of data that must reside in cache can be reduced by combining both temporal and spatial blocking techniques [16, 20]. For this so-called 3.5D blocking [14], the output grid is tiled, using 2.5D blocking, and the resulting spatial blocks are blocked further temporally. On the one hand, this reduces the amount of data to be transferred across the bus significantly, if the buffer is not evicted accidentally from cache. On the other hand, a combined temporal and spatial blocking requires a small part of the temporary grid to be recomputed repeatedly. This leads to an increase in overhead if the block size is too small or the number of smoother applications which are combined is too large. A suitable number N of smoother applications to be combined for a maximum throughput is given by the inequality

$$N \geq \left\lceil \frac{\gamma}{\Gamma} \right\rceil \tag{1}$$

introduced by Nguyen et al. [14]. γ denotes the bandwidth-to-compute ratio (code balance) of the stencil kernel. Analogously, Γ describes the peak-bandwidth-to-peak-compute ratio of the execution platform (machine balance).

Let us consider the 3D 1st-order smoother code of Figure 2. An update of a single grid point requires three multiplications, six additions, one subtraction, eight reads from both input arrays and one store to the output, resulting in nineteen operations in total. Since input elements that have been loaded previously can be reused, only two elements must be fetched from main memory, one from `in` and one `rhs`, which results in transferring sixteen bytes for double precision. Analogously, a single value is written back to main memory, which results in another eight bytes to be transferred. The write-allocate for the store of the result increases the pressure to the bus and in total, this leads to a bandwidth-to-compute ratio of $\gamma = 32$ bytes / 19 op.

3. Performance analysis

This section presents the results of our performance measurements of the optimizations described in the previous section on four different systems.

We have tested our optimizations on four different systems. Two are consumer x86 machines, while the others are server processors. The first test environment has a consumer Intel IvyBridge quadcore processor with a base operating frequency of 2.4 GHz, a 6 MB L3 cache and 16 GB of DDR3-1600 main memory. This processor supports Intel's Turbo Boost technology, which can increase the clock rate by up to 1 GHz. For a better comparison of the results for a different number of cores used, we disabled Turbo Boost for the performance tests. The second system has an AMD Thuban hexacore processor, clocked at 2.7 GHz, also a 6 MB L3 cache, but only 8 GB of DDR2-800 memory. Both processors have an on-chip dual-channel memory controller. In contrast to these two consumer chips, the third test system is a dual-socket ten-core IvyBridge EP machine with the clock frequency fixed to

3.0 GHz. Each of the two processors has a 25 MB wide L3 cache and is connected to 32 GB DDR3-1866 RAM with four memory channels. Our last test system consists of several 16-core BlueGene/Q processors. The deepest cache of this architecture is the L2 cache, which is 32 MB wide. Each processor also contains two built-in dual-channel memory controllers with 8 GB of DDR3-1333 main memory. For both IvyBridge EP and BlueGene/Q, we use only one processor since we are interested in socket-level performance.

Both Intel processors, as well as the AMD Thuban, benefit from quite efficient out-of-order execution units. That is, the processor is able to reorder the instruction stream decoded from the binary for best use of the available execution units. In contrast, BlueGene/Q executes all instructions in-order but, to compensate for this limitation, a single physical processor core can execute up to four threads in parallel in hardware. Thus, if a single thread is not able to load the processor core to capacity, parallel threads can take up the slack. For this reason, we also measured the performance of 32 and 64 threads running on a single 16-core chip. Some of Intel's IvyBridge processors are also able to execute two threads concurrently in hardware but, the aforementioned out-of-order execution unit is sufficient to achieve best performance for the applications considered here. Because of the additional overhead, a management of multiple threads could actually decrease performance at worst-case. Consequently, we disabled this so-called hyper-threading technology on the Intel processors.

All three x86 processors can perform a floating-point ADD instruction and an independent MULT instruction in parallel; BlueGene/Q has instead a fused multiply-add instruction. However, most stencil codes contain far fewer multiplications than additions and cannot exploit these special instructions. Thus, the peak compute performance of stencil codes can reach at most half the theoretical peak compute performance of these architectures. We measured both peak bandwidth and peak stencil compute performance using micro benchmarks and their results are shown in Table 1. The peak bandwidth is measured in GB/s and the peak compute performance in Gop/s. A single operation op is either a floating-point operation or a load/store instruction.

Table 1. Peak bandwidth (GB/s), peak double-precision compute performance (Gop/s) and bandwidth-to-compute ratio (B/op) of the four platforms.

Platform	Peak BW	Peak Gop/s	Byte/op
IvyBridge	22	38	0.58
Thuban	13	31	0.42
IvyBridge EP	47	120	0.39
BlueGene/Q	38	102	0.37

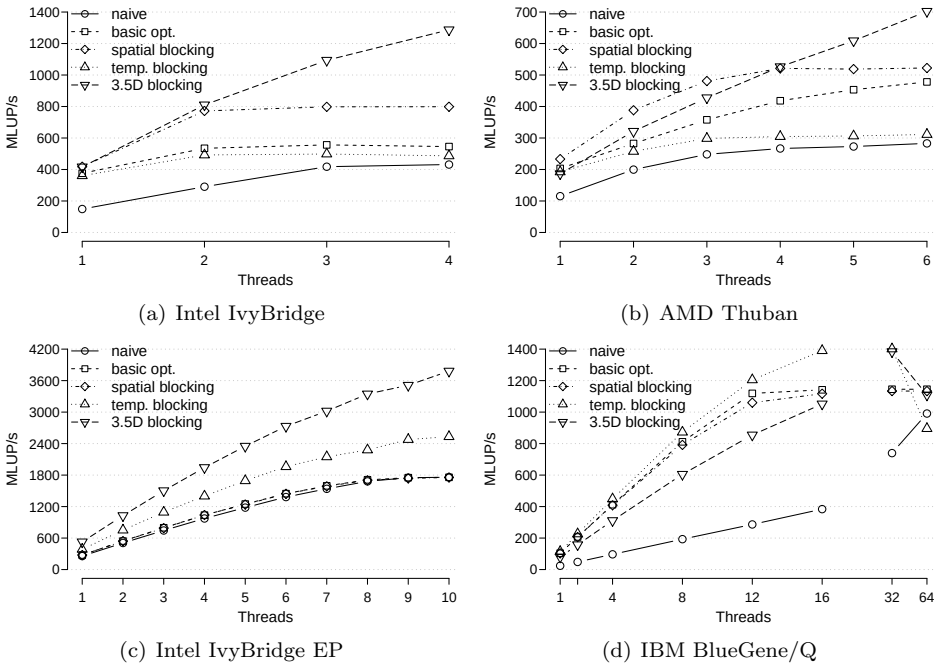
On the consumer x86 architectures, we used gcc 4.8 to generate the executables, which is the most common compiler for consumer systems. We checked that the Intel compiler icc 13 did not generate faster binaries for our optimized test codes. On the two server systems we used the compiler provided by the vendor of the processors

instead. Therefore the source code was translated by Intel's icc 13 on IvyBridge EP. On the BlueGene/Q, we used bgxlC, IBM's XL C++ compiler version 12, which is optimized for this architecture. Because of a compiler bug in the compiler's C frontend, we had to use the C++ compiler, even though our test code is written in standard C99.

3.1. *3D 1st-order smoother*

Our first kernel is a 3D 1st-order Jacobi smoother. Figure 6 illustrates the benefits of our transformations on all test platforms. Performance is measured in million lattice updates per second (MLUP/s). As a baseline, we measured a *naive* implementation, which is basically the same code as shown in Figure 2, except that it was parallelized using OpenMP.

Experiment *basic opt.* includes all optimizations described in Section 2.1. We did not measure the results for any proper subset of these three optimizations, since they depend on each other. E.g., address precalculation is performed by all compilers unless the code is vectorized manually, and the vectorization strategy is based on register blocking. In the *spatial blocking* code, the middle y-loop is being blocked in order to ensure that three subsequent planes of the input code fit into the processors last level cache. The performance of a purely temporal blocking version is labeled with *temp. blocking*, and *3.5D blocking* refers to the combined temporal



(a) Intel IvyBridge (b) AMD Thuban

(c) Intel IvyBridge EP (d) IBM BlueGene/Q

Fig. 6. Performance results for the 1st-order smoother on four platforms.

and spatial blocking described in Section 2.2.

Our basic transformations enable the compiler to generate code that saturates the available memory bandwidth of all test systems. As both consumer processors have a relatively small cache, three planes of the input grid are too large to ensure that data is not evicted before it is required again. A spatial blocking of the middle loop avoids this problem. On IvyBridge EP and BlueGene/Q the *spatial blocking* code shows no improvement over *basic opt.*, as their last level cache is large enough to be able to hold multiple planes of the input grid.

In contrast, the *naive* implementation is compute-bound on BlueGene/Q. This means that it does not use the full memory bandwidth; thus, it benefits from the use of additional available cores. On both consumer processors, the performance of *naive* does not increase linearly and is also not able to reach that of the *basic opt.* version. Due to the usage of the Intel compiler on IvyBridge EP, we were able to add additional simple opimization pragmas to the *naive* version, which enables the compiler to create code as fast as the hand-tuned *basic opt.* code.

For the *3.5D blocking* technique, we determined the value of N in Eq.(1) to be 5 for all except Intel IvyBridge. Consequently, we chose to implement a version that merges five Jacobi iterations. Figures 6(a), 6(b) and 6(c) show a linear speedup for all x86 architectures and a resulting performance higher than the memory bandwidth-bound *basic opt.* and *spatial blocking* versions. This technique transforms a bandwidth-bound to a compute-bound kernel. The same holds for the IBM BlueGene/Q processor with 32 threads, as shown in Figure 6(d), but the difference to the *basic opt.* version is not as high as for the other processors. The performance with less than 16 threads is also worse than for the two other optimized versions, which can be explained by the higher complexity of the *3.5D blocking* code.

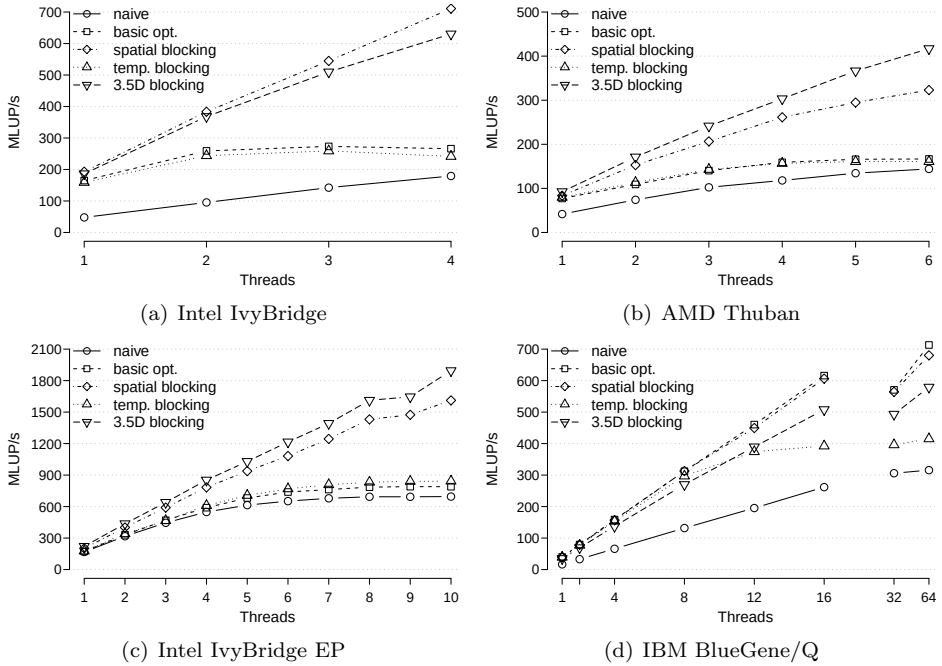
The purely temporal blocking experiment showed the most surprising behaviour. On the two consumer architectures, there is no improvement at all, even though we combined only two smoother applications, rather than five, in order to save on temporary buffer space needed to hold the data handed from one smoother call to the next. The reason is that the amount of memory required for three planes of the 512^3 elements large input does not fit into the cache of either processor. Therefore, a simple temporal blocking shows no improvement. On the other hand, the caches of the two server processors are large enough to hold all temporary buffers as well as larger parts of the input: even for three smoother applications combined, this code performs well as shown in Figure 6(d). Combining five smoother applications, as in case of the *3.5D blocking* version, requires temporary buffers to be large enough to store multiple planes of four intermediate results but, in order to reduce the danger that parts of the buffer are evicted from cache, an additional spatial blocking is required. This leads to smaller blocks of data processed consecutively, which results in an increase of the number of blocks – or, rather, the number of times that computation starts with a cold L1 cache. Due to the comparatively high latency of BlueGene/Q's L2 cache, the purely temporal blocking scheme outperforms *3.5D blocking*. In contrast to the *temp. blocking* scheme, the latter also requires the

recomputation of temporary results at the border of each block, which causes additional overhead. With more than one thread per core, the hardware scheduler can hide the high cache latency. Therefore, the combined temporal and spatial blocking code catches up with the purely *temp. blocking* version.

3.2. *3D 3rd-order smoother*

The benefits of our optimizations of the 3rd-order smoother are depicted in Figure 7. Again, we started with a *naive* baseline implementation as shown in Figure 3, which was again parallelized using OpenMP. This implementation performs similarly to the 1st-order experiment. The only difference is that the Intel compiler was not able to generate a binary as efficient as that for the *basic opt.* on IvyBridge EP. After applying all basic transformations, the code becomes memory-bound on the x86 systems but, in contrast to the 1st-order smoother, the 3rd-order *basic opt.* version is more compute-intensive: the simple structure of the *basic opt.* code enables this version to outperform all others on BlueGene/Q.

The *spatial blocking* version also performs well on all systems. With only 6 MB L3 cache on IvyBridge and Thuban, all previous versions require a repeated load of the input data in cache, which increases the pressure on the memory bus. Therefore, performance can improve significantly when the input grid needs to be loaded only once from main memory.



(a) Intel IvyBridge

(b) AMD Thuban

(c) Intel IvyBridge EP

(d) IBM BlueGene/Q

Fig. 7. Performance results for the 3rd-order smoother on four platforms.

Combining temporal and spatial blocking, the *3.5D blocking* version, first requires to calculate the bandwidth-to-compute ratio γ of the 3rd-order smoother in order to determine the value of N in Eq.(1). Consider again the code in Figure 3. Updating a single grid element requires 24 floating-point operations and 21 memory instructions; thus, a total of 45 operations are needed whereas, in the best case, only 32 bytes have to be transferred across the bus. With a bandwidth-to-compute ratio of $\gamma = 32$ bytes / 45 op $= 0.71$ bytes/op and

$$N \geq \left\lceil \frac{\gamma}{\Gamma} \right\rceil = 2$$

for all platforms, we combined two smoother applications in the *3.5D blocking* technique. In contrast to the 1st-order smoother, the combination of temporal and spatial blocking is not crucial to maximize the throughput of the code as the larger number of data accesses increases the pressure on the processor and the L1 cache. On IvyBridge, the performance is even a bit lower than of the *spatial blocking* code.

A purely temporal blocking approach requires even larger temporary buffers than for the 1st-order kernel, since more neighboring planes have to be stored for the computation of a single output plane. Consequently, on both the Intel IvyBridge and the AMD Thuban, the *temp. blocking* scheme shows no benefit over the *basic opt.* scheme: both are memory-bound. On Intel IvyBridge EP and IBM BlueGene/Q, theoretically, the cache is large enough to store the entire buffer of the temporal blocking code but, practically, relevant cache lines are accidentally evicted from the highest-level cache and have to be reloaded for the next access. Therefore, this code performs worse than the *3.5D blocking* approach, even though both versions combine two smoother applications.

As a general rule, if the sequential, spatially blocked code scales well across the cores on a socket, the potential for temporal blocking is limited, because the memory access time does not dominate in this case. The ECM performance model (see next section) can describe this effect in a quantitative way.

4. Performance Model

In this section, we introduce the ECM performance model and apply it to the 3D 7-point stencil on the Intel IvyBridge EP processor.

In order to develop a purposeful optimization strategy, the limiting resources of the baseline implementation as well as the potential benefit of a planned optimization have to be considered. These insights can be obtained from setting up a performance model, such as the well-known roofline model [19]. The roofline model considers the achievable data transfer bandwidth and the arithmetic throughput to estimate the best possible performance ("light speed").

As we are also interested in the single-core performance and the parallel scaling behaviour on the chip, we apply a refinement of the roofline model recently introduced, the ECM performance model [8, 15]. The ECM model aims to predict the performance of single-threaded loop execution by accounting for run time contribu-

tions from pure code execution (assuming all required data is located in L1 cache) and data transfers throughout the memory hierarchy. A crucial assumption of the model in its basic form is that all data transfers between memory hierarchy levels (including loads and stores in the L1 cache) are mutually non-overlapping and add up to the total run time unless the pure code execution T_{exec} (arithmetic, in-register operations, etc.) takes longer:

$$T = T_0 + \sum_{l=1}^{n} T_l \ , \quad \text{with} \quad T_0 = \max\left(T_{\text{LD/ST}}, T_{\text{exec}}\right) .$$

Here T_l is the data transfer time between memory hierarchy levels l and $l+1$, and $T_{\text{LD/ST}}$ is the time for all loads and stores in the L1 cache. An estimate for T_0 can be obtained by hand in simple cases, or determined with profiling tools like Intel IACA[c]. Data transfer bandwidths between the levels of the memory hierarchy are either documented or have to be measured with suitable micro-benchmarks such as STREAM [13]. The model assumes a streaming scenario with efficient prefetching, hence latency effects are neglected. The cache line size is usually used as the basic unit of work in the model, since this is the minimum amount of data transferred between memory hierarchy levels. For parallel performance predictions, ideal scaling is assumed until the bandwidth of the relevant bottleneck (e.g., a shared cache or memory interface) is fully saturated.

The analysis in this section was performed on the Intel IvyBridge EP system introduced in Section 3. In the following we apply the ECM model to the 1st-order Jacobi smoother shown in Figure 2. Therefore, we determine the time required for one cache line update (eight double-precision elements) for each level of the hierarchy. IACA reports that the in-core execution is dominated by load instructions. The architecture is capable of loading 32 bytes (one full-width AVX load) from L1 cache to a register per cycle. Eight double-precision elements have to be loaded per LUP (central element, its six neighbors, and the `rhs`), resulting in 16 cycles to perform eight LUPs.

The number of load and store streams in the different levels of the cache hierarchy and, thus, the number of transferred cache lines, depend on the number of arrays and on the spatial and temporal access locality. Data reuse during a full grid sweep with an n-order stencil solver is governed by a layer condition, as described earlier: if $2n+1$ layers of the source grid fit into the usable cache, only a single element (the next in the outermost dimension, i.e., in the $x+n$ layer) has to be loaded from the subsequent memory level per update [4, 9]. All other required elements still reside in cache from previous updates. In practice, one cannot assume that the full cache size C_l is available for the source grid; as a rule of thumb we assume that half the

[c]Intel Architecture Code Analyzer
`http://software.intel.com/en-us/articles/intel-architecture-code-analyzer/`

cache can be used:

$$8 \text{ bytes} \cdot (2n+1) \cdot \texttt{dimY} \cdot \texttt{dimZ} < \frac{C_l}{2} \ .$$

If the layer condition is not satisfied at a certain cache level l, the number of load streams to the next level increases, giving rise to a growth of T_{l+1}. To avoid this the layer size may be adjusted with spatial blocking in one or both of the inner dimensions in order to re-establish the layer condition.

The growing overhead of spatial blocking with decreasing block size can outweigh its benefit for the small, lower-level caches. Therefore, in our Jacobi 3D code, spatial blocking is used to ensure the layer condition for the `in` array in L3 cache. Together with the load to `rhs`, the store to `out` and the associated write-allocate transfer, four double-precision elements (32 bytes) have to be transferred between memory and L3 cache per update. On a cache line basis this is equivalent to 16.3 cycles, given a memory bandwidth of 47 GB/s and 3.0 GHz clock frequency. As the layer condition is not satisfied for higher-level caches, two additional load streams to `in` have to be considered (one for the k and one for the $k-1$ layer), resulting in overall six streams. The transfer of six cache lines takes twelve cycles, because the architecture is capable of inter-cache transfers at 32 bytes per cycle.

See Figure 8 for an overview of the model. The aggregated duration of eight updates is 56.3 cycles, which translates to 426 MLUP/s at 3.0 GHz. The pre-

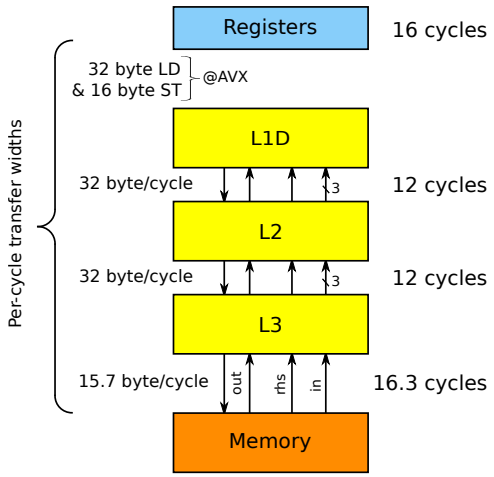


Fig. 8. ECM model for the 3D Jacobi 7-point stencil with spatial blocking on an IvyBridge EP core. An arrow is a 64-byte cache line transfer. Spatial blocking is used to satisfy the layer condition in L3 cache, resulting in only one load on the `in` array per update. As the overhead of spatial blocking would outweigh the benefit for the lower-level caches, two more load streams have to be accepted from L3 and L2 cache. Storing requires an additional load (write-allocate) on this architecture. Memory bandwidth is measured using the STREAM benchmark (47 GB/s @ 3.0 GHz), other bandwidths are given by the machine model. From this, the required time for one cache line update can be calculated for each level of the hierarchy (right column). The 16 cycles for L1 result from loading the central element, its six neighbors, and the `rhs`.

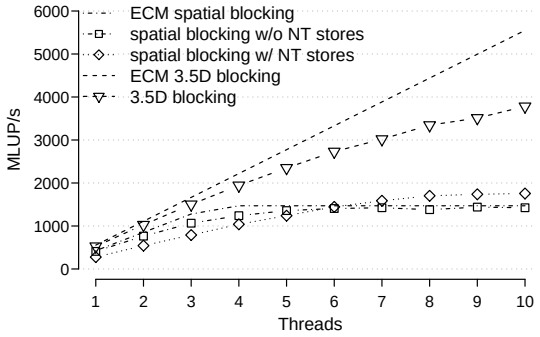


Fig. 9. Single-chip performance scaling of the 3D 7-point Jacobi smoother on 512^3 double-precision grid points.

dicted single-core performance is in good agreement, with a measured value of 409 MLUP/s. As the memory interface is the single resource shared among cores, the roofline model predicts a limit for the full socket of (47 GB/s)/(32 B/LUP) = 1.47 GLUP/s. Assuming ideal scaling, this saturation performance should be reached at four cores. As Figure 9 illustrates, this socket scaling behavior is qualitatively predicted by the ECM model.

Note that the single-socket saturation performance can be increased by a theoretical factor of 4/3 via the use of nontemporal stores. This leads to a maximum socket performance of (47 GB/s)/(24 B/LUP) = 1.96 GLUP/s. In practice, the achievable main memory bandwidth is lower with nontemporal stores, leading to a measured performance of only 1.76 GLUP/s. This value will be the baseline for later comparisons with further optimizations. The ECM model in its current state is not suitable for modeling the scaling behavior when nontemporal stores are used.

Another prominent strategy to reduce memory pressure is temporal blocking as described in Section 2.2. With 3.5D blocking for the L3 cache, the implementation considered here, five Jacobi sweeps are executed on a data set once loaded in L3 cache, as explained in Section 3.1. Neglecting the overhead induced by blocking, this reduces memory traffic by a factor of five (compared to the spatially blocked baseline code without nontemporal stores). With this optimistic assumption, the 16.3 cycles required for memory access reduce to 3.3 cycles, resulting in a best-case single-core performance prediction of 555 MLUP/s, which is a 30% improvement over the baseline. The measurement of 508 MLUP/s is within a 10% margin of the prediction. A 30% speedup may seem marginal, but the true potential of temporal blocking lies in the lifting of the main scalability bottleneck: since the pressure on the memory interface is strongly reduced, performance should scale perfectly up to 13 cores. However, the data in Figure 9 does not show ideal scaling. Possible reasons could be load imbalance and synchronisation overhead as well as excess work due to halo updates. This discrepancy will be subject to further investigation.

5. Related Work

Due to the low amount of computation performed per data point and the usually large grid size, the main optimization goal of stencil codes is to reduce the bandwidth requirements and increase the reuse. Therefore, spatial blocking can be used to increase performance [4, 6, 7]. For the automatic discovery of suitable tiling sizes, different auto-tuning techniques offer themselves [3, 5, 10]. However, a blocking of only the spatial dimensions requires loading each data element at least once from the main memory to the processors cache, which is not sufficient to achieve best performance for lower-order stencil kernels.

If the stencil code is reiterated, it is necessary to include the corresponding time dimension in the blocking scheme to increase data locality. A combination of temporal and spatial blocking introduces data dependences that must be addressed accordingly. Section 2.2 suggests a way of dealing with this phenomenon: reiterate some computations on different processor cores [11, 14]. A similar technique, which does not require the recomputation of values, is called time skewing [21]. But the combination of temporal and spatial blocking without a need to recompute values requires either additional memory or more complex address calculations. Time skewing comes with different blocking techniques, which may result in differently sized trapezoidal blocks [7]. In contrast, the 3.5D blocking scheme described previously generates always equal-sized blocks.

6. Conclusions

We have conducted a study of domain-specific program transformations to bring Jacobi smoother stencil codes to highest performance. In summary, we have learned the following.

Memory bandwidth can be a serious performance brake, especially for low-order Jacobi stencils. In order to not let it take hold, the optimization of the codes has to be customized for the execution platform.

For the 1st-order kernel the best partitioning scheme is *3.5D blocking* for all systems. On BlueGene/Q and AMD Thuban, all cores of the processor must be loaded to capacity; otherwise, other transformations can achieve a better performance. A performance analyises of the *spatial blocking* and the *3.5D blocking* version using the ECM performance model revealed the need for a combination of a temporal and a spatial blocking in order to reduce the bandwidth requirements and also to ensure satisfaction of the layer condition.

The 3rd-order Jacobi smoother requires to load more than twice as many data elements from the memory hierarchy, which increases the amount of work per processor involved in a single lattice update. This reduces the single-core performance to roughly a factor of two. The performance improves if the layer condition is satisfied, which can easily be assured by applying a simple *spatial blocking*. Besides having a simpler code structure, this version performs even better than the *3.5D blocking* scheme on Intel IvyBridge and BlueGene/Q.

An additional observation is that the auto-vectorizer of the Intel compiler performs very well for stencil codes but, with gcc or the IBM compiler, vectorization has to be done mostly by hand.

All optimizations presented in Section 2 can be applied automatically by a stencil code generator. If the generator has knowledge of the target architecture, it can generate highly optimized code for a specific problem. Using the cache and grid size, along with the available vector instructions, the smoother code can be tiled and vectorized to saturate the memory bandwidth and to achieve a similar performance as the *spatial blocking* version presented in Figures 6 and 7. Building such a generator is the charter of project ExaStencils.

Acknowledgements

This work is part of project ExaStencils in the DFG Priority Research Initiative SPPEXA, grant no. LE 912/15-1. We are grateful for useful discussions with Gerhard Wellein (RRZE).

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2nd edition, 2007. Section 6.4.3.

[2] M. Bolten. *Multigrid Methods for Structured Grids and their Application in Particle Simulation*. PhD thesis, Bergische Universität Wuppertal, 2008.

[3] K. Datta. *Auto-tuning Stencil Codes for Cache-Based Multicore Platforms*. PhD thesis, EECS Department, University of California, Berkeley, Dec. 2009.

[4] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Rev.*, 51(1):129–159, Feb. 2009.

[5] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proc. Int. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12. IEEE Press, 2008.

[6] H. Dursun, K. Nomura, W. Wang, M. Kunaseth, L. Peng, R. Seymour, R. K. Kalia, A. Nakano, and P. Vashishta. In-core optimization of high-order stencil computations. In *Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 533–538. CSREA Press, July 2009.

[7] M. Frigo and V. Strumpen. Cache oblivious stencil computations. In *Proc. 19th Int. Conf. on Supercomputing (ICS)*, pages 361–366. ACM, June 2005.

[8] G. Hager, J. Treibig, J. Habich, and G. Wellein. Exploring performance and power properties of modern multi-core chips via simple machine models. *Concurrency and Computation: Practice and Experience*, 2013. Early view, DOI: 10.1002/cpe.3180.

[9] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. Computational Science Series. CRC Press, 2010.

[10] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *Proc. 24th Int. Parallel & Distributed Processing Symp. (IPDPS)*, Apr. 2010. 12 pp.

[11] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *Proc.*

2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI), pages 235–244. ACM, 2007.

[12] S. Kronawitter and C. Lengauer. Optimization of two Jacobi smoother kernels by domain-specific program transformation. In A. Größlinger and H. Köstler, editors, *Proc. 1st Int. Workshop on High-Performance Stencil Computations*, pages 75–80. www.epubli.de, Jan. 2014.

[13] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, Dec. 1995.

[14] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In *Proc. Int. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–13. IEEE, Nov. 2010.

[15] J. Treibig and G. Hager. Introducing a performance model for bandwidth-limited loop kernels. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics (PPAM)*, LNCS 6067, pages 615–624. Springer, Sept. 2010.

[16] J. Treibig, G. Wellein, and G. Hager. Efficient multicore-aware parallelization strategies for iterative stencil computations. *J. Computational Science*, 2(2):130–137, May 2011. Special issue on Simulation Software for Supercomputers.

[17] U. Trottenberg, C. W. Osterlee, and A. Schüller. *Multigrid*. Academic Press, 2000.

[18] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske. Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. *Proc. IEEE 33rd Ann. Computer Software and Applications Conf. (COMPSAC)*, 1:579–586, July 2009.

[19] S. W. Williams, A. Waterman, and D. A. Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Technical Report UCB/EECS-2008-134, EECS Department, University of California, Berkeley, Oct. 2008.

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

[21] D. Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *Proc. 14th Int. Parallel & Distributed Processing Symp. (IPDPS)*, pages 171–180. IEEE Computer Society, May 2000.