Optimization of two Jacobi Smoother Kernels by Domain-Specific Program Transformation

Stefan Kronawitter
University of Passau
Innstraße 33
94032 Passau, Germany
stefan.kronawitter@uni-passau.de

Christian Lengauer
University of Passau
Innstraße 33
94032 Passau, Germany
christian.lengauer@uni-passau.de

ABSTRACT

Our aim is to apply program transformations to stencil codes, in order to yield highest possible performance. We observe memory bandwidth as a major limitation in stencil code performance. We conducted a small study in which we applied optimizing transformations to two Jacobi smoother kernels: one 3D 1st-grade 7-point stencil and one 3D 3rd-grade 19-point stencil. To obtain highest performance, the optimizations have to be customized for the execution platform at hand. We illustrate this by experiments on two x86 architectures and one BlueGene/Q architecture. A compiler with specific knowledge about stencil codes and execution platforms should be able to apply our transformations automatically. We are working towards such a compiler in the DFG-funded project ExaStencils.

1. INTRODUCTION

Multigrid methods [6] are widely used in scientific applications, especially in physics or chemistry simulations. Much of the time consumed by a multigrid algorithm is due to the smoother used by it. We study two concrete Jacobi smoothers [2]: one for a 3D 1st-grade 7-point smoother and one for a 3D 3rd-grade 19-point smoother. Both smoothers refer to a single output and two input grids. The three grids are located in distinct memory areas.

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

In the 3D 3rd-grade 19-point smoother kernel, the update of a single element requires not only the nearest neighbours but all elements in a range of three points in each direction. Figure 2 depicts the 2D footprint of this stencil. The source code of the corresponding 3D kernel appears in Figure 4.

The direct implementations of the codes in Figures 3–4 have very poor performance. 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[1], which is part of the DFG-funded priority research initiative SPPEXA[2].

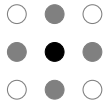


Figure 1: Footprint of a 2D 1st-grade stencil.

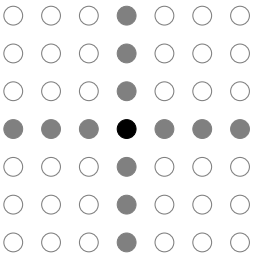


Figure 2: Footprint of a 2D 3rd-grade stencil.

A serious limiting factor 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. We employ different blocking techniques to reduce the bandwidth requirements of our codes. The evaluation of our optimizations show that architectures with small, but very fast caches, like modern x86 consumer CPUs, require a more complex blocking technique than architectures with a large, but comparatively slow highest-level cache, like, e.g., 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 experimental evaluation of the optimizations applied to two codes on three architectures,
- insights into the suitability of the various optimizations for the individual architectures,
- insights into the basic limitations of stencil code performance.

HiStencils 2014
First International Workshop on High-Performance Stencil Computations
January 21, 2014, Vienna, Austria
In conjunction with HiPEAC 2014.

http://www.exastencils.org/histencils/2014/

[1] http://www.exastencils.org
[2] http://www.sppexa.de

```
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];
```

Figure 3: Kernel code of a 3D 1st-grade 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][z+1] + in[x][y-1][z] + in[x][y+1][z] + in[x-1][y][z] + in[x+1][y][z])
              + b2 * (in[x][y][z-2] + in[x][y][z+2] + in[x][y-2][z] + in[x][y+2][z] + in[x-2][y][z] + in[x+2][y][z])
              + b3 * (in[x][y][z-3] + in[x][y][z+3] + in[x][y-3][z] + in[x][y+3][z] + in[x-3][y][z] + in[x+3][y][z])
              -  c * rhs[x][y][z];
```

Figure 4: Kernel code of a 3D 3rd-grade Jacobi smoother.

2. OPTIMIZATIONS

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. 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 and 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 3. 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, 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 was 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 5 illustrates this optimization for a single 3D array, regardless of whether it is linearized in memory or accessed using a 3D pointer `double ***in`.

```
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]);
  }
```

Figure 5: Simplified address calculation for the innermost loop.

Another optimization beneficial for higher-grade stencils would be to reorder the computations such that an entire cache line is read at the same time. This prevents a repeated load of the same data into L1 cache [3]. Consider again the stencil of Figure 2. Let us assume that the data is stored in row-major order in memory (as, e.g., in C) and that the required neighbours in the same row can be loaded from contiguous addresses, while the elements from the same column have a higher stride. Consequently, when loading elements from the same column but from a different row, each access requires the corresponding cache line to be loaded into L1 cache, even if only a single double precision value is copied to a register. Therefore, the cache line could be evicted before subsequent elements 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 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 almost every situation, 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 multidimensional 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.

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 completely 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-grade kernel, only three and, for the 3rd-grade kernel, seven neighbouring planes must reside in cache at the same time. Thus, the outermost dimension must not be blocked explicitly. Nguyen et al. call this technique 2.5D blocking [4].

Another possibility to increase the performance is to combine multiple applications of the kernel code. 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-grade Jacobi smoother are being combined, the buffer for the temporary results must be large enough to hold three planes. In a warm-up phase, the first two planes of the initial smoother application have to be 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 that must be transferred over 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 is typically only a few MB wide. But the amount of data that must reside in cache can be reduced by combining both blocking techniques described previously. For this so-called 3.5D blocking [4], the output grid is tiled, using 2.5D blocking, and the resulting spatial blocks are further blocked temporally. On the one hand, this reduces the amount of data to be transferred over 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. [4]. γ denotes the bandwidth-to-compute ratio of the stencil kernel. Analogously, Γ describes the peak-bandwidth-to-peak-compute ratio of the execution platform.

Let us consider the 3D 1st-grade smoother code of Figure 3. 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. In general the hardware has to load a cache line from main memory before it can be modified, so the old value of the output array has to be fetched from main memory, too. This so-called write-allocate increases the pressure to the bus and in total, this makes a bandwidth-to-compute ratio of $\gamma = 32$ bytes / 19 op.

Another, similar technique, which does not require the recomputation of values, is called time skewing [7]. But, the combination temporal and spatial blocking without a need to recompute values requires either additional memory or more complex address calcuations. Time skewing comes with different blocking techniques, which may result in differently sized trapezoidal blocks [5]. In contrast, the 3.5D blocking scheme described previously generates always equally sized blocks.

3. PERFORMANCE ANALYSIS

We have tested our optimizations on three different systems. Two are consumer x86 machines, while the third is a BlueGene/Q architecture. The first test environment has an Intel Ivy Bridge quadcore processor, which is clocked at 3.1 GHz when all four cores are running, a 6 MB L3 cache and 16 GB of DDR3-1600 main memory. 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. In contrast to these two consumer chips, our third test system consists of several 16-core BlueGene/Q processors, of which we use only one since we are interested in node-level performance. The deepest cache of this architecture is the L2 cache, which is 32 MB wide. Each processor also contains two built-in memory controllers with 8 GB of DDR3-1333 main memory.

Both Intel Ivy Bridge and 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 threads in-order but, compensating 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 threads running on a

Platform	Peak BW	Peak Gop/s	Byte/op
Ivy Bridge	22	49	0.45
Thuban	13	31	0.42
BlueGene/Q	38	102	0.37

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

single 16 core chip. Some of Intel's Ivy Bridge processors are also able to execute two threads concurrently in hardware. But, as all threads execute the same, small kernel code and there are no other code blocks, which might make use of different parts of the processor, the aforementioned out-of-order execution unit is sufficient to achive best performance. Because of the additional overhead, a management of multiple threads would actually decrease performance. Consequently, we disabled this so-called hyper-threading technology of Ivy Bridge.

Both x86 processors can perform a floating-point addition and an independent multiplication in parallel; BlueGene/Q has instead a fused multiply-add instruction. But 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 one half of 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.

On the x86 architectures, we used gcc 4.7 to generate the executables. We ascertained that the Intel compiler icc 13 did not generate faster binaries for our test codes. 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-grade smoother

The first kernel we have examined is a 3D 1st-grade Jacobi smoother. Figure 6 illustrates the benefits of our transformations an all three test platforms. As a baseline, we measured a *naive* implementation of this kernel, which is basically the same code as shown in Figure 3, except that it was parallelized using the following OpenMP pragma:

```
#pragma omp parallel for collapse(3) schedule(static)
```

which directs the compiler to distribute the iterations of the three-fold loop nest equally across all threads.

Experiment *basic opt.* includes all optimizations described in Section 2.1, whereas *temp. blocking* measured a purely temporal blocking and *3.5D blocking* refers to the combined temporal and spatial blocking described in Section 2.2. An purely spatial blocking experiment is not included, since it yields no improvement over the *basic opt.* experiment.

Our basic transformations enable the compiler to generate quite efficient code which saturates the available memory bandwidth of all test systems while using at most half of the physical processor cores. In contrast, the *naive* implementation is compute-bound on all architectures. This means that it does not use the full memory bandwidth; thus, it benefits from using additional available cores. In our experiments, every core added increased the performance.

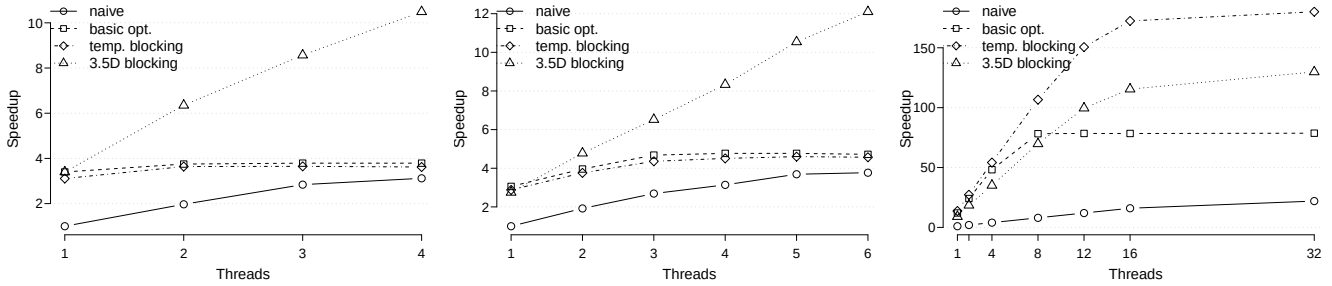
For the *3.5D blocking* technique, we determined the value of N in Inequality 1 as 5 for AMD Thuban and IBM BlueGene/Q and 4 for Intel Ivy Bridge. Consequently, we chose to implement a version that merges five Jacobi iterations. Figures 6(a) and 6(b) show a linear speedup for both x86 architectures and a resulting performance much higher than the memory bandwidth-bound *basic opt.* version. This technique transforms the bandwidth-bound to a compute-bound kernel. The same holds for the IBM BlueGene/Q processor when using all 16 physical cores, as shown in Figure 6(c), but the difference to the *basic opt.* version is not as high as for the consumer processors.

The purely temporal blocking experiment showed the most surprising behaviour. On the two x86 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 x86 processor. Therefore, a simple temporal blocking shows no improvement. On the other hand, BlueGene/Q's cache is large enough to hold all temporary buffers as well as larger parts of the input: even for three smoother applications combined, this code performs very well as shown in Figure 6(c). Combining five smoother applications, as in case of the *3.5D blocking* version, requires temporary buffers to be large enough to store 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.

3.2 3D 3rd-grade smoother

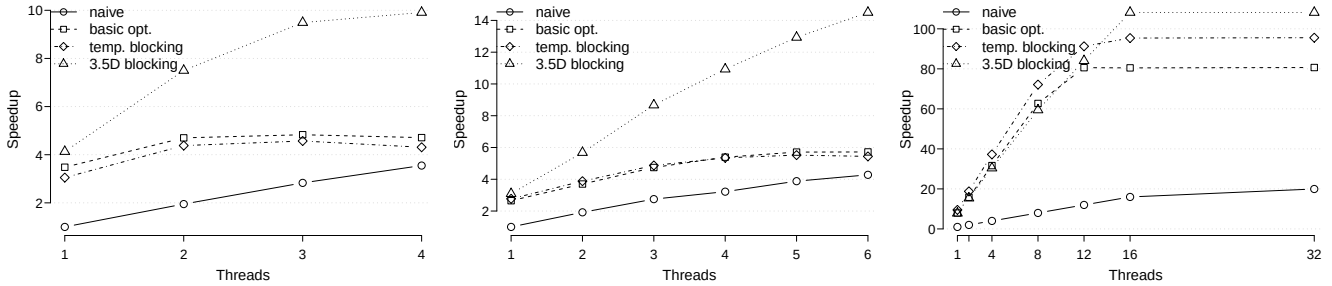
The benefits of our optimizations of the 3rd-grade smoother are depicted in Figure 7. Again, we started with a *naive* baseline implementation as shown in Figure 4, parallelized the same way as described in the previous subsection. The implementation is just as inefficient, which renders this code compute bound on the tested platforms. After applying all basic transformations, the code becomes memory-bound but, in contrast to the 1st-grade smoother, the 3rd-grade *basic opt.* version is more compute-intensive: it requires more processor cores to load the memory bandwidth to capacity.

Combining temporal and spatial blocking to the *3.5D blocking* version first requires to calculate the bandwidth-to-compute ratio γ of the 3rd-grade smoother in order to determine the value of N in Inequality 1. Consider again the code in Figure 4. 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 over the bus. With



(a) Intel Ivy Bridge (b) AMD Thuban (c) IBM BlueGene/Q

Figure 6: Performance results for the 1st-grade smoother on three platforms.



(a) Intel Ivy Bridge (b) AMD Thuban (c) IBM BlueGene/Q

Figure 7: Performance results for the 3rd-grade smoother on three platforms.

a bandwidth-to-compute ratio of $\gamma = 32$ bytes / 45 op = 0.71 bytes/op and

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

for all platforms, we combined two smoother applications in the *3.5D blocking* technique.

As in case of the 1st-grade smoother, the combination of temporal and spatial blocking is essential for x86 processors to maximize the throughput of the code. Figures 7(a) and 7(b) show a speedup of more than factor two compared to the *basic opt.* version.

A purely temporal blocking approach requires even larger temporary buffers than for the 1st-grade kernel, since more neighbouring planes have to be stored for the computation of a single output plane. Consequently, on both the Intel Ivy Bridge and the AMD Thuban, the *temp. blocking* scheme shows no benefit over the *basic opt.* scheme: both are memory bound. On the IBM BlueGene/Q, a simple temporal blocking leads to a faster code because the cache is large enough to store the entire buffer. But, when using all 16 cores, the *3.5D blocking* version performs better, even though both versions combine two smoother applications. That is, theoretically, the cache is large enough but, practically, cache lines corresponding to the buffer are accidentally evicted from the L2 cache and have to be reloaded for the next access.

4. CONCLUSIONS

We have conducted a small 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. In order not to let it take hold, the optimization of the codes has to be customized for the execution platform. We have studied platforms of two types:

- x86 architectures have comparatively small highest-level caches with a low latency. This requires the fragmentation of the data into comparatively small blocks. The best partitioning scheme is *3.5D blocking*. The AVX/SSE instructions for unaligned loads and stores are an additional help. However, for highest performance, vectorization has to be specified explicitly.
- BlueGene/Q is a comparatively simple architecture aimed at high processor numbers. This is reflected in the fact that performance on a single processor is poor and additional processors result in a comparatively high speedup. The highest-level cache is larger and has higher latency. This makes a purely temporal blocking scheme most suitable.

In our experiments, execution speed on the three architectures differed by at most a factor of 2, with Ivy Bridge being the fastest. Also, it must be said that theoretical peak performance is not within reach because of the low arithmetic intensity of stencil codes.

5. ACKNOWLEDGEMENTS

This work is part of project ExaStencils in the DFG priority programme SPPEXA, grant no. LE 912/15-1.

6. 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] 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, 2009.

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

[5] R. Strzodka, M. Shaheen, D. Pajak, and H.-P. Seidel. Cache oblivious parallelograms in iterative stencil computations. In *Proc. 24th Int. Conf. on Supercomputing (ICS)*, pages 49–59. ACM, 2010.

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

[7] 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, 2000.