SIAM J. SCI. COMPUT.
Vol. 37, No. 2, pp. C194–C216

MODELING THE PERFORMANCE OF GEOMETRIC MULTIGRID STENCILS ON MULTICORE COMPUTER ARCHITECTURES*

PIETER GHYSELS† AND WIM VANROOSE‡

Abstract. The basic building blocks of the classic geometric multigrid algorithm all have a low ratio of executed floating point operations per byte fetched from memory. On modern computer architectures, such computational kernels are typically bound by memory traffic and achieve only a small percentage of the theoretical peak floating point performance of the underlying hardware. We suggest the use of state-of-the-art (stencil) compiler techniques to improve the flop per byte ratio, also called the arithmetic intensity, of the steps in the algorithm. Our focus will be on the smoother which is a repeated stencil application. With a tiling approach based on the polyhedral loop optimization framework, data reuse in the smoother can be improved, leading to a higher effective arithmetic intensity. For an academic constant coefficient Poisson problem, we present a performance model for the multigrid V-cycle solver based on the tiled smoother. For increasing numbers of smoothing steps, there is a trade-off between the improved efficiency due to better data reuse and the additional flops required for extra smoothing steps. Our performance model predicts time to solution by linking convergence rate to arithmetic intensity via the roofline model. We show results for two-dimensional (2D) and three-dimensional (3D) simulations on Intel Sandy Bridge and for 2D simulations on Intel Xeon Phi architectures. The actual performance is compared with the theoretical predictions.

Key words. multigrid, performance model, multicore, bandwidth

AMS subject classifications. 65Y20, 65M55

DOI. 10.1137/130935781

1. Introduction.

For a while now, the increase in processor clock frequency has stalled; instead, Moore's law that dictates decreasing feature lengths is translating into more on-chip parallelism, sustaining Moore's self-fulfilling prophecy. For instance, Intel AVX instructions [20] operate on eight single precision floating point numbers simultaneously, while the Intel Xeon Phi accelerator chip has 512 bit wide vector registers [21], good for 16 single precision floating point numbers. Other sources of parallelism are the increasing core count and the use of hyperthreading. However, for many scientific codes it turns out to be very hard to get optimal performance from current highly parallel processors. There are two main reasons for this: one is that with increasing parallelism the synchronization between threads becomes more costly, and the other is that it is very hard to supply sufficient data to the computational units in order to keep them busy at all times. The latter is due to the limited bandwidth to DRAM memory that is shared between cores. For nonuniform memory access

*Submitted to the journal's Software and High-Performance Computing section September 5, 2013; accepted for publication (in revised form) January 30, 2015; published electronically March 31, 2015. This work is funded by Intel and by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT). All authors from Lawrence Berkeley National Laboratory were supported by the DOE Office of Advanced Scientific Computing Research under contract number DE-AC02-05CH11231. The U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes. Copyright is owned by SIAM to the extent not limited by these rights.
http://www.siam.org/journals/sisc/37-2/93578.html

†Department of Mathematics and Computer Science, University of Antwerp, Middelheimlaan 1, B-2020 Antwerp, Belgium, Intel ExaScience Lab, Kapeldreef 75, B-3001 Leuven, Belgium, and Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720 (pghysels@lbl.gov).

‡Department of Mathematics and Computer Science, University of Antwerp, Middelheimlaan 1, B-2020 Antwerp, Belgium (wim.vanroose@ua.ac.be).

architectures (NUMA), different memory banks can be addressed simultaneously by different cores, which effectively increases the bandwidth but complicates memory management. In their recently introduced Xeon Phi accelerator, Intel has bumped the memory bandwidth to 320 GB/s. However, this theoretical throughput seems impossible to achieve, given that even a synthetic code such as the STREAM [29] benchmark achieves only a fraction of this. This indicates that the bandwidth bottleneck is not going to be magically overcome by technological advances, including so-called three-dimensionally stacked memory [27].

Many scientific computer codes are bandwidth limited computational kernels. Examples of such kernels are stencil applications, dot products, and vector additions. These are the typical building blocks for algorithms from sparse linear algebra, such as Jacobi iteration [2], geometric multigrid [6, 39, 17], and Krylov solvers [33, 16] that appear in scientific codes that solve problems modeled by partial differential equations [14]. Since the underlying kernels are highly bandwidth limited, so are the algorithms built on them. Numerical kernels can be classified based on their so-called arithmetic intensity, the number of floating point operations performed per byte fetched from the main memory. An algorithm typically has a fixed arithmetic intensity that we shall refer to as its q value, where a low q value suggests a bandwidth limited kernel. Algorithms with higher q values, on the other hand, like dense linear algebra algorithms and particle methods, are typically bound not by memory bandwidth but by the floating point performance of the hardware.

Fusing several computational kernels into a single new kernel can lead to a higher theoretical arithmetic intensity because there is more opportunity for data reuse within a larger kernel. For instance, the arithmetic intensity of a single kernel that performs ν successive stencil applications is not necessarily the same as that of the kernel that only executes a single stencil application, applied ν times. Furthermore, parallelism is typically only exploited at the level of the algorithms building blocks, leading to very fine-grained parallelism and high synchronization costs. In addition to increasing the theoretical arithmetic intensity, fusing different computational kernels can also reduce synchronization overhead.

In this paper we study the arithmetic intensity of geometric multigrid and look for ways to improve it. Although all steps in the algorithm have low arithmetic intensity, we focus on the smoother since it is a repeated call to a stencil operation, and by fusing multiple steps the arithmetic intensity increases, and the synchronization overhead is reduced. There is a large literature on optimizing stencil computations, and many tools are readily available. Stencil compilers such as Pochoir [36] and Patus [8] have shown impressive results, generating optimized code from a stencil specified in a domain-specific language. The generated code might enable many optimizations, such loop unrolling, cache and register blocking, array padding, software prefetching, and single instruction multiple date (SIMD) vectorization. All these optimizations have also been studied, for instance, in [42]. The goal of this paper is not to present a highly optimized geometric multigrid code but to illustrate the possible performance improvements by increasing the arithmetic intensity of the smoother and also to quantify these benefits with a simple performance model. We shall restrict ourselves to plain C code and use a tool called Pluto [4, 5, 1], which is a source-to-source code translator. Pluto optimizes nested loops for data locality and parallelism and can apply tiling of stencil iterations over multiple stencil steps. Numerical results for two-dimensional (2D) and three-dimensional (3D) simulations are presented on modern multicore computer systems, such as a 16 core compute node based on the Intel Sandy Bridge architecture and a 61 core Intel Xeon Phi (KNC generation) accelera-

tor. Although our current approach is restricted to stencils on regular grids, geometric multigrid is not restricted to this simple geometry. We discuss possible generalization to more complex problems and geometries in later chapters.

Geometric multigrid is a very widely used algorithm for the solution of systems of equations, either as a stand-alone method or as a preconditioner in a Krylov solver, due to its excellent complexity and good opportunities for parallelization. Because of the popularity of multigrid, many authors have focused on improving performance of multigrid codes (see, for instance, [11, 13, 12, 23]), or the performance of just stencil applications [9, 8], including temporal blocking strategies [41, 38, 36]. The Ph.D. dissertation of Kowarschik [22] is probably the first detailed discussion of the memory access patterns in multigrid. These results are extended in the dissertation by Treibig [37], which also briefly considers temporal blocking. Tiling or blocking over multiple smoothing steps was also already used in [15] and in [42]. These previous works have focused typically on separate computational kernels, like the smoother. It is observed that blocking over different applications of the stencil (the smoother) leads to better performance for more consecutive iterations (smoothing steps), since this allows for more data reuse. However, changing the number of smoothing steps changes the multigrid cycle convergence rate. In this paper we discuss in detail the trade-offs between convergence properties and improved floating point performance when applying such tiling. Using a model problem, we can quantify this trade-off and accurately predict performance gains, which are especially dramatic on multicore architectures, where the better data reuse leads to lower memory bandwidth usage, better scalability with the number of cores, and most likely a lower energy footprint. In industrial applications, such as, for instance, real-time animation of characters for a new computer animated movie [28], this better scalability might make a crucial difference.

There is a growing awareness that minimization of flops should no longer drive algorithm design; rather, minimization of data movement and synchronization should. The behavior of geometric multigrid applied to the 2D Poisson problem is well known [6] in terms of floating point operations to solution. However, this completely ignores the hardware aspects and is thus not relevant as a measure of time to solution. The multigrid performance model presented in this paper builds on these results and additionally takes into account the data movements. This work is meant as a motivating example to start rethinking algorithm complexities with data movement in mind. Our performance model is original in the sense that it includes both numerical properties (the multigrid convergence rate) as well as machine characteristics (peak performance and bandwidth).

The paper is outlined as follows. Section 2 reviews the roofline model [43], which links floating point performance to arithmetic intensity. Section 3 briefly introduces the standard geometric multigrid V-cycle applied to an academic model problem. Section 3.1 discusses the different computational kernels with their arithmetic intensity. Section 3.2 looks at the multigrid convergence rate and section 3.2.3 at the required number of multigrid cycles, both as a function of the number of smoothing steps. Section 4 describes our multigrid performance model that takes into account numerical properties as well as data movement. In section 5 we show how a stencil compiler can be used to improve the performance of the smoother by tiling over multiple smoothing steps. In section 5.2 timings of the multigrid code with tiled smoother are compared with the performance model from section 4. Section 6 has some concluding remarks.

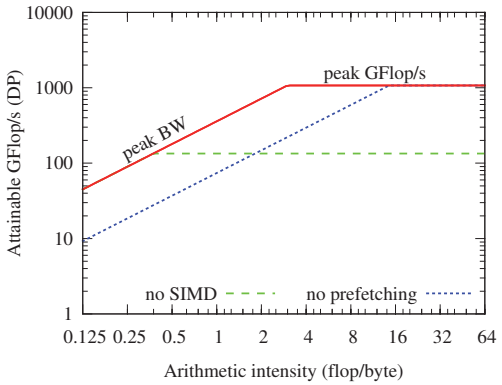


FIG. 1. *The roofline model predicting floating point performance based on arithmetic intensity. Low arithmetic intensity kernels are bandwidth bound, whereas high arithmetic intensity kernels are compute bound. Many lower rooflines can be added which can be overcome by corresponding code optimizations.*

2. Arithmetic intensity and the roofline model. The number of floating point operations performed per byte fetched from main memory is typically fixed for a computational kernel. This ratio, denoted q, is called the arithmetic intensity and can be used to predict the floating point performance of the kernel through the roofline model [43, 32]:

$$\text{Attainable GFlop/sec}\,(q) = \min(\text{Peak GFlop/s}, \text{Peak Mem BW} \times q)\,. \tag{2.1}$$

The model depends on the theoretical peak floating point performance and peak bandwidth of the hardware being considered. The roofline model for an Intel Xeon Phi coprocessor, Knight's Corner (KNC) architecture, is illustrated in Figure 1. The KNC has a peak double precision floating point performance of 1073 GFlop/s (see section 5 for more information on our test systems). Note in this figure the use of the logarithmic scale on both the horizontal axis (arithmetic intensity) and the vertical axis (maximal attainable floating point performance). The roofline consists of two parts: one 45° line for low arithmetic intensities corresponding to the bandwidth bound regime, and a horizontal line (at the peak floating point performance) corresponding to the compute bound regime. The maximum memory bandwidth can be derived from the value of the roofline at $q = 1$ flop/byte. This most basic roofline gives the theoretical maximum attainable performance for a given arithmetic intensity. However, an actual implementation of a useful algorithm almost never achieves the maximum floating point performance or memory bandwidth. There are many bottlenecks in achieving this peak performance, and these translate into different rooflines, below the theoretical maximum roofline. For instance, not optimally using the SIMD vector units on the processor limits the maximum attainable floating point performance. A separate roofline can be associated with kernels that do not use SIMD. Likewise, when the required data is not contiguous in memory or is not prefetched on time, the maximum memory bandwidth will not be obtained, which also corresponds to a lower roofline.

This simple model already can tell us that for computational kernels with very low arithmetic intensity, such as the building blocks of multigrid, SIMD computation is not needed to reach the maximum memory bandwidth and cannot help in overcoming this performance bound. Typically, code tuning tries to eliminate all lower rooflines

in order to get as close as possible to the theoretical maximum roofline without altering the arithmetic intensity. In this paper we shall mainly focus on increasing the arithmetic intensity to improve performance.

3. Multigrid. Multigrid is an iterative solver that consists of repeated application of stencil operators. Its building blocks are the smoother, interpolation, and restriction operators that can each be written as stencil operations. A standard implementation, however, will result in a low arithmetic intensity code. Since codes with low arithmetic intensity lose performance relative to high arithmetic intensity solvers (dense solvers), multigrid is at a disadvantage.

Section 3.1 presents the different steps of geometric multigrid applied to the 2D model problem with a discussion of the arithmetic intensity of each step. Section 3.2 gives a derivation of the multigrid convergence rate based on a two-grid correction scheme.

3.1. Model problem. The model problem used in this paper is the 2D Poisson problem

$$-\Delta u = f \quad \forall\, x, y \in [0,1]^2 \tag{3.1}$$

with homogeneous Dirichlet boundary conditions on the four sides of the domain; see also [6]. The equation is discretized with finite differences with $n-1 = 2^\ell - 1$ interior grid points in each direction and a grid spacing $h = 1/n$. Let $N = n^2$ denote the total number of grid points. The resulting discrete problem is

$$A^h_{\mathrm{2D}} u^h = f^h\,, \tag{3.2}$$

where $u^h \in \mathbb{R}^N$ is the representation of the solution on the grid and $f^h \in \mathbb{R}^N$ is the right-hand side evaluated in the grid points. The matrix $A^h_{\mathrm{2D}} \in \mathbb{R}^{N\times N}$ for the 2D Poisson problem can be written as a Kronecker product of the one-dimensional (1D) operators as

$$A^h_{\mathrm{2D}} = I \otimes A^h_{\mathrm{1D}} + A^h_{\mathrm{1D}} \otimes I, \tag{3.3}$$

where $I \in \mathbb{R}^{N\times N}$ is the unit operator and

$$A^h_{\mathrm{1D}} = -\frac{1}{h^2}\begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 1 & -2 \end{bmatrix}. \tag{3.4}$$

This linear algebra problem is solved iteratively with a standard geometric multigrid algorithm [6]. The building blocks for multigrid are the smoother, which removes oscillatory errors from the guess for the solution, and the intergrid operators, restriction and interpolation, that transfer the error between the grids.

A simple smoother operator is weighted Jacobi (ω-Jacobi) iteration, which takes a guess v^h for the solution u^h and applies the stationary iteration

$$v^h \leftarrow (I - \omega D^{-1} A^h_{\mathrm{2D}}) v^h + \omega D^{-1} f^h, \tag{3.5}$$

where D is the diagonal of the operator A^h_{2D} and ω is a weight, usually taken to be $2/3$ in one dimension and $4/5$ in two dimensions. Weighted Jacobi is a stencil operation,

and the cost of an application is linear in the number of grid points, $\mathcal{O}(N)$. In pseudocode, a naive implementation of ν applications of the weighted Jacobi smoother for the 2D Poisson problem is a triple nested loop:

LISTING 1
ω-Jacobi smoother.

```
1 c = h^2/4*omega;
2 for (k = 1 to nu) {
3   for (i = 1 to n)
4     for (j = 1 to n)
5       w[i,j]=v[i,j]-c*((4*v[i,j]-v[i-1,j]-v[i+1,j]-v[i,j-1]
                         -v[i,j+1])/h^2-f[i,j]);
6   swap(w, v);
7 }
```

The residual $r^h = f^h - A^h v^h$ is also computed with a straightforward stencil application:

LISTING 2
Residual calculation.

```
1 for (i = 1 to n)
2   for (j = 1 to n)
3     r[i,j] = f[i,j] - (4*v[i,j] - v[i-1,j] - v[i+1,j] - v[i,j-1]
                         - v[i,j+1])/h^2;
```

To transfer v^h between the levels of the hierarchy we make use of standard full weighting $v^{2h} = {}^{2\mathrm{D}}I_h^{2h} v^h$, where the 2D restriction operator ${}^{2\mathrm{D}}I_h^{2h}$ is the Kronecker product of the 1D restriction, ${}^{2\mathrm{D}}I_h^{2h} = {}^{1\mathrm{D}}I_h^{2h} \otimes {}^{1\mathrm{D}}I_h^{2h}$. The 1D operator is

$$ {}^{1\mathrm{D}}I_h^{2h} = \frac{1}{4} \begin{bmatrix} 1 & 2 & 1 & & & & & & \\ & & 1 & 2 & 1 & & & & \\ & & & & \ddots & \ddots & \ddots & & \\ & & & & & & 1 & 2 & 1 \end{bmatrix}. \tag{3.6} $$

The interpolation operator is, up to a multiplication constant, the transpose of the restriction operator and can also be written as a Kronecker product ${}^{2\mathrm{D}}I_{2h}^{h} = {}^{1\mathrm{D}} I_{2h}^{h} \otimes {}^{1\mathrm{D}} I_{2h}^{h}$. Again, application of the restriction I_h^{2h} and interpolation I_{2h}^h operators can be written as stencil operations, and their cost is linear in the number of grid points, $\mathcal{O}(N)$.

Pseudocode for the 2D restriction of the fine grid residual `rf` ($= r^h$) to the coarse grid residual `rc` ($= r^{2h}$) is as follows.

LISTING 3
Restriction.

```
1 for (i = 2 to n step 2)
2   for (j = 2 to n step 2)
3     rc[i/2,j/2] = (rf[i-1,j-1] + rf[i+1,j-1] + rf[i-1,j+1]
                     + rf[i+1,j+1])/16
4           + (rf[i-1,j] + rf[i+1,j] + rf[i,j-1] + rf[i,j+1])/8
            + rf[i,j]/4;
```

Interpolation from a coarse grid error `ec` ($= e^{2h}$) to the fine grid error `ef` ($= e^h$) can be implemented with a single sweep over the grid as follows.

LISTING 4
Interpolation.

```
1 for (i = 0 to nc)
2   for (j = 0 to nc) {
3     if (i>0 and j>0) ef[2*i  ,2*j  ] = ec[i,j];
4     if (j>0)         ef[2*i+1,2*j  ] = (ec[i+1,j] + ec[i,j]) / 2;
5     if (i>0)         ef[2*i  ,2*j+1] = (ec[i,j+1] + ec[i,j]) / 2;
6     ef[2*i+1,2*j+1] = (ec[i,j] + ec[i,j+1] + ec[i+1,j]
                         + ec[i+1,j+1]) / 4;
7   }
```

These are the building blocks for a V-cycle that traverses the multigrid hierarchy from top to bottom and back. A multigrid solver is then constructed by repeated application of a V-cycle. A recursive formulation of a V-cycle is given in Algorithm 1. The algorithm starts from an initial guess for the solution v^h and a given right-hand side f^h. It then recursively traverses the multigrid hierarchy as follows: first, it applies ν_1 ω-Jacobi steps in line 1, then calculates the residual, r^h, and applies the restriction operator to arrive at the residual r^{2h} at the $2h$ level; see line 6. It then calls itself for a coarser level to solve the error equation $A^{2h}e^{2h} = r^{2h}$ at the $2h$ level. When the result arrives it corrects the guess v^h with the interpolated error $I^h_{2h}e^{2h}$, where e^{2h} is an approximation for the error at the $2h$ grid. Finally, in line 11 it applies the smoother again with ν_2 ω-Jacobi iterations. The first ν_1 applications of the smoother are called presmoothing, and the last ν_2 applications are called postsmoothing.

Algorithm 1. V-Cycle$^h(v^h, f^h)$.

1: Relax ν_1 iterations: $v^h \leftarrow (1 - \omega D^{-1}A^h)v^h + \omega D^{-1}f^h$
2: **if** Coarsest level **then**
3: go to line 11
4: **end if**
5: $r^h \leftarrow f^h - A^h v^h$
6: $r^{2h} \leftarrow I^{2h}_h r^h$
7: $e^{2h} \leftarrow 0$
8: $e^{2h} \leftarrow V$-cycle$^{2h}(e^{2h}, r^{2h})$
9: $e^h \leftarrow I^h_{2h} e^{2h}$
10: $v^h \leftarrow v^h + e^h$
11: Relax ν_2 iterations: $v^h \leftarrow (1 - \omega D^{-1}A^h)v^h + \omega D^{-1}f^h$

A straightforward implementation of this algorithm will have a very low arithmetic intensity if separate routines are used for each application of the ω-Jacobi smoother, for the interpolation and for the restriction. Indeed, these stencil routines will each traverse all the grid points of the domain and apply a local stencil operation. Since each of these consecutive steps has low arithmetic intensity, so has the V-cycle. For instance, a single application of the ω-Jacobi smoother performs $9n^2$ flops, reads $2n^2$ numbers from main memory, and writes n^2 numbers to main memory. This gives an arithmetic intensity for the smoother of $q = 3/8$ flop/byte. If there is no data reuse between multiple applications of the smoother, then smoothing with ν ω-Jacobi steps has the same, low, arithmetic intensity. Similarly, one can count the number of flops and memory transfers for the interpolation and restriction operators, which gives $q = 1/5$ and $q = 11/40$ for interpolation and restriction, respectively. The computation of the residual has an arithmetic intensity $q = 7/24$ flop/byte and for the error correction $q = 1/24$.

Only on the coarsest levels of the V-cycle, where the grids are so small that they fully fit in the cache, can we benefit from data reuse.

3.2. Two-grid convergence for the Poisson problem. In order to build an accurate performance model and explore the possibilities to increase the arithmetic intensity, we need an estimate for the multigrid convergence rate. The convergence rate determines the number of V-cycle iterations necessary to reach a given tolerance. This section gives a traditional multigrid cost model that predicts a rising cost as more smoothing steps are applied. In section 4 we modify the cost model to take bandwidth into account.

A frequently used model in the analysis of multigrid is a two-grid scheme that approximates the V-cycle, which traverses all the levels of the hierarchy, by just two levels; see, for instance, [6, 39]. The two levels are the finest level with grid distance h and the level below with grid distance $2h$, often called the coarse level. On these two levels the following operators are applied in sequence on a guess v^h with error $e^h = u^h - v^h$:

$$e^{h(i+1)} = \left(I - I_{2h}^h \left(A^{2h}\right)^{-1} I_h^{2h} A^h\right) R^\nu e^{h(i)} . \tag{3.7}$$

First, on the finest level the smoother operator R is applied ν times. Then the residual is calculated and restricted to the coarse level $r^{2h} = I_h^{2h}(f^h - A^h v^h)$. This results in the application of $I_h^{2h} A^h$. On the coarser grid the error equation is solved exactly by applying $(A^{2h})^{-1}$. The resulting error e^{2h} is then interpolated back to the fine level, where it corrects the guess $v \leftarrow v + I_{2h}^h e^{2h}$. This sequence of operations can be summarized by a single two-grid operator $\mathrm{TG} = (I - I_{2h}^h (A^{2h})^{-1} I_h^{2h} A^h) R^\nu$, such that $e^{h(i+1)} = \mathrm{TG}\, e^{h(i)}$. The convergence rate of a V-cycle is approximated by the spectral radius $\rho(\mathrm{TG})$ of the two-grid operator. This spectral radius has been frequently analyzed in the literature; see, for instance, the multigrid tutorial [6] for an easily accessible introduction.

To estimate the spectral radius for these modifications we need to analyze the eigenvalues of the two-grid operator for the model problem (3.2). The eigenvalues and eigenvectors for the 1D Poisson matrix are

$$\lambda_k^h = \frac{4}{h^2} \sin^2\left(\frac{k\pi}{2n}\right) , \quad w_k^h(j) = \sin\left(\frac{jk\pi}{n}\right) , \quad k = 1, \ldots, n-1 , \tag{3.8}$$

where $j = 1, \ldots, n$ for the interior points. For the 2D Poisson matrix we have

$$A_{2\mathrm{D}}^h = I \otimes A_{1\mathrm{D}}^h + A_{1\mathrm{D}}^h \otimes I , \tag{3.9}$$

and the corresponding eigenvalues and eigenvectors are

$$\lambda_{k,\ell}^h = \lambda_k^h + \lambda_\ell^h \quad \text{and} \quad w_{k,\ell}^h = w_k^h \otimes w_\ell^h . \tag{3.10}$$

3.2.1. 1D analysis. The TG operator can be analyzed by expanding the initial error e^h in eigenvectors of $A_{1\mathrm{D}}^h$ and applying the TG operator on these eigenmodes. Since the smoother R and A_{1D}^h share the same eigenvectors, the action of ν smoother steps with the iteration matrix $R = (I - \omega D^{-1} A_{1\mathrm{D}}^h)$ on an eigenvector is

$$R^\nu w_k^h = \left(1 - 2\omega \sin^2\left(\frac{k\pi}{2n}\right)\right)^\nu w_k^h . \tag{3.11}$$

The restriction operator I_h^{2h} works on pairs of eigenmodes, so-called complementary modes, with the complementary eigenmode of w_k^h defined as $w_{k'}^h \equiv w_{n-k}^h$. The restriction operator maps two of these eigenmodes, $\text{span}\{w_k^h, w_{k'}^h\}$ to $\text{span}\{w_k^{2h}\}$, and the action on these modes is

(3.12)
$$I_h^{2h} w_k^h = c_k^h w_k^{2h}\,, \quad k = 1, \ldots, \frac{1}{2}\,, \quad \text{and} \quad I_h^{2h} w_{k'}^h = -s_k^h w_k^{2h}\,, \quad k = 1, \ldots, \frac{n}{2} - 1,$$

where $c_k^h = \cos^2(\frac{k\pi}{2n})$ and $s_k^h = \sin^2(\frac{k\pi}{2n})$. The interpolation maps $\text{span}\{w_k^{2h}\}$ to the space of complementary modes $\text{span}\{w_k^h, w_{k'}^h\}$ as

$$I_{2h}^h w_k^{2h} = c_k^h w_k^h - s_k^h w_{k'}^h\,, \quad k = 1, \ldots, \frac{n}{2} - 1\,. \tag{3.13}$$

Using (3.12) and (3.13), the two-grid operator as defined in (3.7) but without smoother and acting upon an eigenmode w_k^h and its complementary mode $w_{k'}^h$ can be written as

(3.14)
$$\text{TG} \begin{bmatrix} w_k^h \\ w_{k'}^h \end{bmatrix} = \left(I - \left(\begin{bmatrix} c_k^h \\ -s_k^h \end{bmatrix} \otimes \begin{bmatrix} c_k^h & -s_k^h \end{bmatrix} \right) \frac{1}{\lambda_k^{2h}} \begin{bmatrix} \lambda_k^h & \\ & \lambda_{k'}^h \end{bmatrix} \right) \begin{bmatrix} w_k^h \\ w_{k'}^h \end{bmatrix}\,, \quad k = 1, \ldots, \frac{n}{2} - 1\,.$$

Simplifying this and including the smoother leads to

$$\text{TG} \begin{bmatrix} w_k^h \\ w_{k'}^h \end{bmatrix} = \begin{bmatrix} s_k^h & s_k^h \\ c_k^h & c_k^h \end{bmatrix} \begin{bmatrix} \left(1 - 2\omega \sin^2\left(\frac{k\pi}{2n}\right)\right)^\nu & 0 \\ 0 & \left(1 - 2\omega \sin^2\left(\frac{(n-k)\pi}{2n}\right)\right)^\nu \end{bmatrix} \begin{bmatrix} w_k^h \\ w_{k'}^h \end{bmatrix},$$
$$k = 1, \ldots, \frac{n}{2} - 1\,. \tag{3.15}$$

By computing the eigenvalues of this 2×2 matrix, we find an expression for the spectral radius of the TG operator:

$$\rho(\text{TG}) = \max_{k=1,\ldots,\frac{n}{2}-1} \left| s_k \left(1 - 2\omega s_k\right)^\nu + c_k \left(1 - 2\omega c_k\right)^\nu \right|\,. \tag{3.16}$$

3.2.2. 2D analysis. A similar analysis holds for the 2D problem. In two dimensions the eigenmodes $w_{k,\ell}^h$ now have two indices k and ℓ, and the TG operator maps the subspace $\text{span}\{w_{k,\ell}^h, w_{k',\ell}^h, w_{k,\ell'}^h, w_{k',\ell'}^h\}$ onto itself, where the indices of the complementary modes are denoted as $k' = n - k$ and $\ell' = n - \ell$. Let

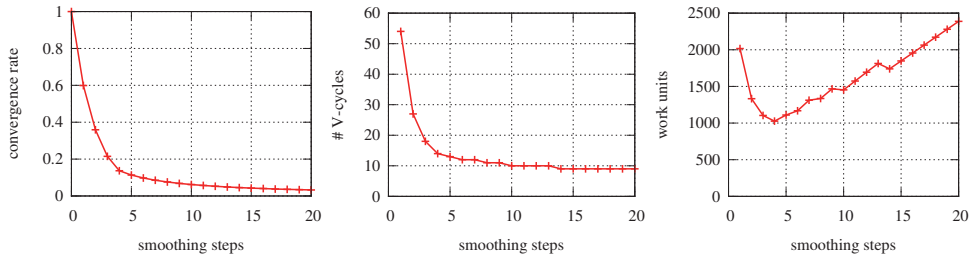
$$T_k^h = \begin{bmatrix} c_k^h & -s_k^h \end{bmatrix} \quad \text{and} \quad T_{k,\ell}^h = T_k^h \otimes T_\ell^h = \begin{bmatrix} c_k^h c_\ell^h & -c_k^h s_\ell^h & -s_k^h c_\ell^h & s_k^h s_\ell^h \end{bmatrix}\,, \tag{3.17}$$

where T_k^h is the action of the 1D interpolation on an eigenmode and $T_{k,\ell}^h$ its 2D equivalent. Also, let

(3.18)
$$W_{k,l}^h = \begin{bmatrix} w_{k,\ell}^h & w_{k',\ell}^h & w_{k,\ell'}^h & w_{k',\ell'}^h \end{bmatrix}^T \quad \text{and} \quad \Lambda_{k,\ell}^h = \begin{bmatrix} \lambda_{k,\ell}^h & \lambda_{k',\ell}^h & \lambda_{k,\ell'}^h & \lambda_{k',\ell'}^h \end{bmatrix}^T\,.$$

Then the 2D TG operator, with ω-Jacobi smoother, applied on a mode and its complementary modes gives

$$^{\text{2D}}\,\text{TG}\, W_{k,l}^h = \left(I - \left(\left(T_{k,\ell}^h\right)^T \otimes T_{k,\ell}^h \right) \frac{1}{\lambda_{k,\ell}^{2h}} \text{diag}\left(\Lambda_{k,\ell}^h\right) \right) \left(I - 2\omega\, \text{diag}\left(\Lambda_{k,\ell}^h\right)\right)^\nu W_{k,l}^h\,,$$
$$k = 1, \ldots, \frac{n}{2} - 1\,, \quad \ell = 1, \ldots, \frac{n}{2} - 1\,. \tag{3.19}$$



(a) Convergence rate (b) V-cycles (c) work units

FIG. 2. *A two-grid correction scheme with ω-Jacobi smoother applied to the 2D Poisson problem.* (a) *Convergence rate as a function of the number of smoothing steps.* (b) *The number of V-cycles required to reach a relative tolerance of 10^{-12}.* (c) *The computational cost to reach the same tolerance expressed in Work Units rises as soon as more than four smoothing steps are applied.*

The maximum eigenvalue of this 4×4 matrix ${}^{2D}\,\mathrm{TG}$ leads to a good estimate for the convergence rate of a multigrid V-cycle with ω-Jacobi smoother applied to the 2D Poisson problem. Figure 2 (a) shows this convergence rate as a function of the number of smoothing steps ν for the ω-Jacobi smoother.

3.2.3. Required number of iterations and cost of the V-cycle. The number of V-cycles required to reduce the norm of the error by a given tolerance ϵ can now be estimated as

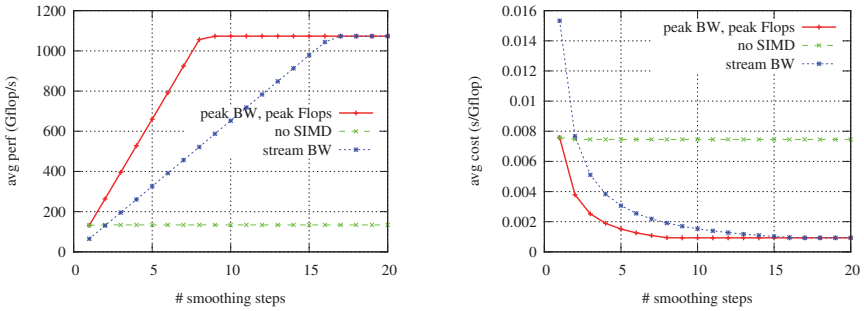
$$m = \left\lceil \frac{\log \epsilon}{\log \rho(\mathrm{TG})} \right\rceil . \tag{3.20}$$

The number of iterations to reach a tolerance of $\epsilon = 10^{-12}$ as a function of the number of smoothing steps is given in Figure 2 (b). As the number of smoothing steps increases, fewer iterations are required. However, if the cost of additional smoothing steps is taken into account, a different picture emerges. When the cost of each smoothing step is assumed to be linear in the number of grid points, i.e., $\mathcal{O}(N)$, the cost rises. We define a Work Unit to be the cost of applying a stencil to the fine grid, regardless of the exact number of floating point operations per stencil. The total cost for the V-cycle to reach a tolerance ϵ, expressed in terms of Work Units, is then

$$(9\nu + 19)\left(1 + \frac{1}{4} + \cdots + \frac{1}{4^{\log_2(n)-1}}\right)\left\lceil \frac{\log \epsilon}{\log \rho(\mathrm{TG})} \right\rceil \mathrm{WU} \leq (9\nu + 19)\frac{4}{3} m \,\mathrm{WU}\,, \tag{3.21}$$

where 9 is the number of flops per grid point for the ω-Jacobi smoother and 19 the total number of flops per grid point for residual computation, restriction, interpolation, and error correction, i.e., lines 5, 6, 9, and 10, respectively, in Algorithm 1. This cost to reach a 10^{-12} tolerance is shown in Figure 2 (c). The rising cost reflects that the application of only a few (4) smoothing steps (pre+postsmoothing) is sufficient to remove the oscillatory modes from the error. The resulting error is thus relatively smooth and can be efficiently removed by the coarse grid corrections. Using more smoothing steps does not make the V-cycle more efficient.

4. Multigrid cost model taking into account the arithmetic intensity. The naive multigrid cost model presented in section 3.2.3, inequality (3.21), does not take into account communication costs. Since each step in the multigrid algorithm has a low arithmetic intensity, the algorithm is bandwidth limited, and this communication cost is important for an accurate prediction of the performance. We look for



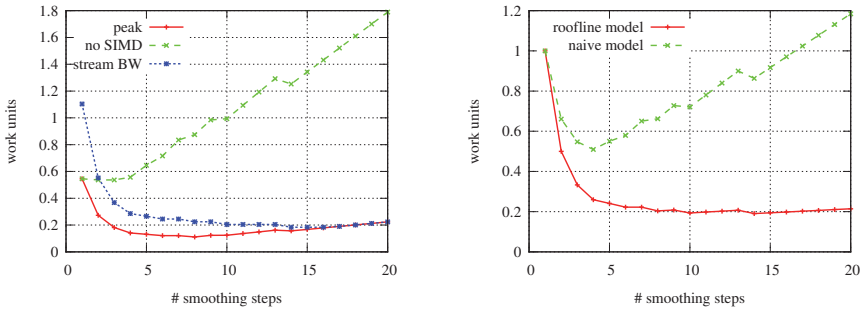
(a) Attainable GFlop/sec (b) Average cost in sec/GFlop

FIG. 3. (a) *The roofline performance model translated into the performance of a k-step dependent SpMV kernel. As the number of matrix-vector products increases, the increasing arithmetic intensity leads to better performance. When only a few matrix-vector products are used the kernel is memory bandwidth limited. There is only a benefit from using SIMD instructions when the number of SpMV's is sufficiently high, and only for even higher numbers of steps can the vector unit be fully be exploited.* (b) *This model then translates into a decreasing average cost per SpMV in the kernel until peak performance is reached. This diminishing cost is the result of decreasing communications costs.*

opportunities to reduce the communication by increasing the arithmetic intensity and present a modified performance model that takes into account the arithmetic intensity of the different steps in the algorithm.

In Algorithm 1, the residual computation, the restriction, the interpolation, and the error correction are only applied once per V-cycle, so it is not possible to increase the arithmetic intensity of any of those steps individually. The smoother, however, is applied repeatedly, although ν_1 and ν_2 are typically small—just 1, 2, or 3. The smoother is a k-step dependent sparse matrix-vector (SpMV) kernel so the arithmetic intensity of the smoother can be improved by tiling over different smoothing steps. For k consecutive smoothing steps it can theoretically be increased by a factor k over that of a single smoothing step, since intermediate results can be kept in fast memory instead of written to main memory and read back again for the next step. It is to be expected that for an increasing number of smoothing steps, a trade-off will occur between the reduced communication costs and the loss of effectiveness for the multigrid algorithm.

In Figure 3 (a) the roofline model is translated into a performance model for a k-step dependent SpMV kernel. Let $q_1(\text{SpMV})$ be the arithmetic intensity of a single application of the SpMV. With each k we associate an arithmetic intensity $q(k \times \text{SpMV}) = k\, q_1(\text{SpMV})$, based on the assumption that tiling over different SpMVs gives perfect data reuse, which is probably a good approximation of a small number of steps. The arithmetic intensity, in its turn, is linked to performance in GFlop/s through the roofline. This can then be used to calculate the expected average cost per SpMV in the kernel, as illustrated in Figure 3 (b). The cost per SpMV diminishes with each additional multiplication until we hit peak performance. Figure 3 shows three rooflines: the roofline based on the theoretical maximum peak bandwidth and maximum floating point performance of a hypothetical machine, as it would be advertised by the manufacturer; a roofline where the bandwidth as measured by the STREAM benchmark is used instead; and a roofline that uses maximum floating point performance without the use of SIMD instructions. We shall use this model with the three rooflines in a multigrid performance model to study the impact



(a) MG cost model based on roofline (b) Roofline-based vs. naive cost model

FIG. 4. (a) *Prediction of the cost for multigrid to reduce the error with a given tolerance based on a performance model that takes into account the arithmetic intensity of the different stencils and the lowering average cost of the smoother for repeated applications. The cost is plotted as a function of the number of presmoothing steps ν_1 and without postsmoothing, i.e., $\nu_2 = 0$. There is only a small benefit on the minimum cost from using SIMD instructions. Increasing the bandwidth has a bigger impact for a small number of smoothing steps.* (b) *Comparing the multigrid cost model based on the arithmetic intensity with the naive model.*

of the bandwidth and vectorization on the performance.

Consider again the multigrid cost model with fixed cost for each smoother application as given by (3.21) and shown in Figure 2. We now let the cost of applying a stencil depend on the arithmetic intensity of the specific stencil operation. For instance, the cost of interpolating the coarse grid error back to the fine grid becomes

$$\langle I_{2h}^h e^{2h} \rangle = \frac{\text{flops}\left(I_{2h}^h e^{2h}\right)}{\text{roof}\left(q(I_{2h}^h e^{2h})\right)}, \tag{4.1}$$

where flops(...) denotes the number of flops per grid point required for the given operation. The modified cost model for the V-cycle multigrid solver becomes

$$m\frac{4}{3}\left(\langle \nu_1 \times \omega\text{-Jac}\rangle + \langle \nu_2 \times \omega\text{-Jac}\rangle + \langle f^h - A^h v^h\rangle + \langle I_h^{2h} r^h\rangle + \langle I_{2h}^h e^{2h}\rangle + \langle v^h + e^h\rangle\right) \text{WU} \tag{4.2}$$

with m the required number of V-cycles to reach a given tolerance as defined in (3.20). Note that for the cost of the (pre)smoother

$$\langle \nu_1 \times \omega\text{-Jac}\rangle = \frac{\text{flops}\left(\nu_1 \times \omega\text{-Jac}\right)}{\text{roof}\left(q(\nu_1 \times \omega\text{-Jac})\right)} \approx \frac{\nu_1 \text{flops}\left(\omega\text{-Jac}\right)}{\text{roof}\left(\nu_1 q_1(\omega\text{-Jac})\right)}, \tag{4.3}$$

where we use that the roofline of ν_1 applications can be related to the arithmetic intensity of a single step. The cost as predicted by (4.2) is illustrated in Figure 4 (a). Whereas the original model shows a rising cost as soon as more than a few smoothing steps are used, the updated model shows rapidly diminishing cost for the first few smoother applications. Then, at around 3 or 4 smoothing steps the slope changes. Additional smoothing steps do not bring a benefit to the multigrid algorithm since the error is already smooth enough to be represented at a coarser level. However, the additional cost of the extra smoothing is mostly compensated by the diminishing cost per smoothing step due to data reuse (the slope of the roofline). The result is that the cost still declines with additional smoothing, up to the point where the smoother kernel reaches peak performance. From then on (around 6 or 13 smoothing steps without

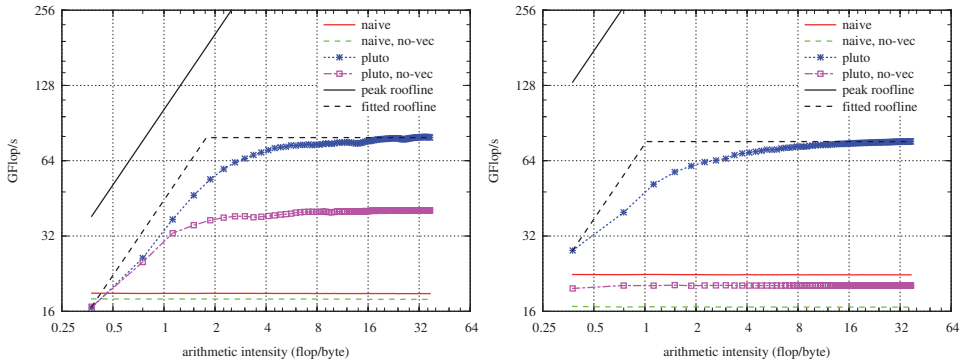
and with vectorization, respectively), the total cost starts to rise approximately linearly. Figure 4 (a) shows the cost model with and without vectorization and for both the theoretical peak bandwidth and the lower stream bandwidth. This shows that although the speedup from vectorization for large numbers of smoothing steps can be significant, the overall reduction in cost for the multigrid solver is marginal. Finally, in Figure 4 (b) the new cost model, based on arithmetic intensity, is compared with the naive model (3.21). This suggests that significant speedups can be obtained on a machine corresponding to this particular roofline by tiling the smoother.

For the convergence rate, only the sum $\nu = \nu_1 + \nu_2$ of the number of presmoothing steps, ν_1, and the number postsmoothing steps, ν_2, matters, because post- and presmoothing can be combined since the postsmoothing step of one V-cycle is immediately followed by the presmoothing of the next cycle. However, in the performance model we need to distinguish between presmoothing and postsmoothing phases since they are in different function calls and splitting them alters arithmetic intensity. To avoid confusion, the numerical results will only use presmoothing, i.e., $\nu_2 = 0$. However, this might not be the optimal choice in practice when, for instance, only a single multigrid cycle is used as a preconditioner.

5. Numerical results. In this section we implement a smoother kernel that reuses data and avoids redundant read from slow memory. As discussed in section 2, several tools are available for generating optimized stencil code. Here, we use the source-to-source translation tool Pluto [4]. Section 5.2 discusses how the smoother is optimized with the help of Pluto by improving its arithmetic intensity. The optimized smoother is used in a multigrid V-cycle, and the measured timings are compared with the performance model from section 4.

5.1. Pluto loop optimization framework. Pluto [4, 5, 1] is a framework for automatic loop parallelization and locality optimizations based on the polyhedral model [26], also referred to as the polytope model. Pluto was chosen over the alternative stencil compilers for its capability to tile over different smoother iterations (time-tiling) and for its ease of use, flexibility, and overall performance of the generated code. As Pluto is a C/C++ source-to-source translation tool, the input is a regular piece of C code surrounded by preprocessor pragmas. The transformations used in Pluto's polyhedral optimizer apply to nested loops with affine loop bounds and subscripts. Such nested loops can be reordered, including tiling of the iteration space, and automatically made parallel by simply adding the appropriate OpenMP pragmas around the outermost loop. Furthermore, the resulting code is made amenable for vectorization by adding the compiler pragmas `ivdep` and `vector always`. This means that vectorization relies completely on the compiler's (guided) autovectorization capabilities.

It should be noted that the polyhedral framework and Pluto are more generally applicable than just stencil applications; for instance, most linear algebra kernels can be written as nested loops with affine loop bounds and subscripts, and they typically benefit heavily from cache blocking. Pluto can be used as a stand-alone tool, taking C code and generating optimized C code. The Polly plugin adds support for polyhedral optimizations to the LLVM compiler by using the same algorithms as Pluto. PPCG [40] is a tool similar to Pluto, using some of the LLVM infrastructure. Likewise, GCC includes a polyhedral framework called Graphite. Since Polly and especially Graphite are in early development phases, we found that using just Pluto in combination with GCC and ICC to compile the generated code gives the best workflow. Although it can be expected that these compiler technologies will improve



(a) Dual socket Intel Sandy Bridge, 16 × 2 threads

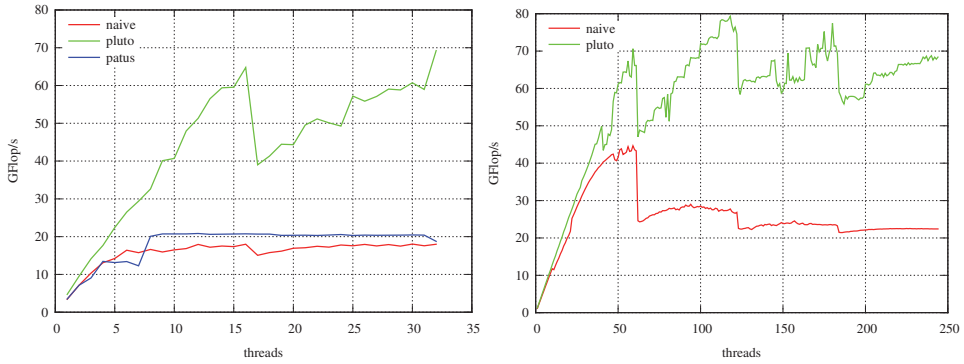
(b) Intel Xeon Phi, 61 × 4 threads

FIG. 5. (a) *The roofline as experimentally measured with the Jacobi smoother, tiled with Pluto, on a dual-socket Intel Sandy Bridge machine, using* 32 *threads.* (b) *Similar experiment on an Intel Xeon Phi using* 244 *threads. The smoother is applied on an* 8000^2 *domain. The naive code has a fixed arithmetic intensity, while for the Pluto tiled code the arithmetic intensity and the performance increase as more smoothing steps are performed. Both the naive code and the tiled code are run with and without autovectorization. Note that vectorization only pays off at higher arithmetic intensity.*

significantly over the coming years, one should always have a basic understanding of what is going on under the hood in order to help the compiler by making code easier to optimize.

5.2. Tiling the smoother. We have implemented the weighted Jacobi smoother in plain C, optimized for data locality and parallelism with Pluto. See Appendix A for an example piece of code generated by Pluto. Figure 5 shows the performance for different numbers of repeated applications of these smoothers. Figure 5 (a) shows an experiment on a dual socket Intel Sandy Bridge (SB) compute node with 16 hardware cores in total, and Figure 5 (b) shows the same experiment on an Intel Xeon Phi (KNC generation) with 61 cores. For the tiled code a tile size of $64 \times 64 \times 64$ was used for the two spatial dimensions and the smoother iteration dimension on the SB and $8 \times 128 \times 8$ on the KNC. The horizontal axis of these figures denotes the optimal arithmetic intensity of the kernel, which is computed as $q(k \times \omega\text{-Jac}) = kq_1(\omega\text{-Jac})$, i.e., the number of smoother iterations times the arithmetic intensity of a single application of the smoother. The number of smoother steps ranges from 1 to 100. For the naive implementation, the arithmetic intensity does not change with the number of iterations, and therefore performance does not depend on the number of iterations. For the Pluto tiled code the performance improves as more smoothing steps are performed and the arithmetic intensity increases. The trend of the line clearly approximates a roofline. The roofline that is fitted to the data in Figure 5 is described by only two parameters: maximum attained memory bandwidth and maximum attained floating point performance. For the SB the measured roofline is given by 44.5 GB/s and 79.6 GFlop/s, and for the KNC we found 74.7 GB/s and 76.3 GFlop/s for memory bandwidth and floating point performance, respectively.

Figure 5 shows both experiments with and without autovectorization, and clearly there is only a gain from vectorization when the arithmetic intensity is high enough. On the SB machine the maximum vectorization speedup of 1.97 × is a reasonable fraction of the theoretical factor 4 that can be attained for double precision calculations.



(a) Dual socket Intel Sandy Bridge (b) Intel Xeon Phi

FIG. 6. *Comparison of the scalability for* 10 *iterations of the Jacobi smoother on an* 8000^2 *domain for both the naive implementation and the Pluto tiled code. The Sandy Bridge results are also compared with code generated by the Patus stencil compiler. Especially on the general purpose Xeon, the tiled code scales significantly better due to the lower bandwidth usage compared to the naive code. The Sandy Bridge can run* 2 *threads per core using hyperthreading, and the Xeon Phi can run* 4 *threads per core. The performance drops after* 16 *threads for the Sandy Bridge and after* 61, 122, *and* 183 *for the Xeon Phi are due to the hyperthreading not scaling perfectly.*

On the KNC the vectorization speedup was $3.77\times$ with a theoretical maximum of $8\times$.

This improved usage of the available bandwidth directly results in a better parallel performance. Figure 6 shows the parallel scalability of applying 10 smoothing steps with both the naive implementation and the tiled code on the two test machines; (a) shows the SB and (b) the KNC. The SB has a total of 16 hardware cores but can run two threads per core using hyperthreading. The KNC has 61 cores which can support a total of 244 threads. On the SB we used `numactl` to interleave the data over the NUMA memory banks to increase the available memory bandwidth. On both test machines, we see better scalability of the tiled code compared to the naive implementation due to lower memory bandwidth usage. On the SB the tiled code runs $3.85\times$ faster, while on the KNC a speedup of $3.05\times$ is achieved from tiling. These speedups are for 10 smoothing steps and using the default OpenMP number of threads, 32 and 244 for the SB and KNC, respectively. The naive code scales remarkably well on the KNC, up to 61 cores, due to its immense peak memory bandwidth of 352 GB/s compared to only 102.4 GB/s for the SB. The naive code performed best with 44.67 GFlop/s with 59 threads on the KNC compared to 18.02 GFlop/s with 30 threads on the SB. Hence, for the naive implementation, the KNC outperforms a dual socket SB by a factor $2.48\times$. In Figure 6, a comparison with optimized code generated by the Patus stencil compiler [8] is also given for the SB. For this platform, Patus generates inline assembly code enabling AVX and prefetching with a few parameters that are determined by autotuning. The Patus code is highly optimized but only outperforms our naive handwritten C code slightly. The tiled code outperforms the Patus code by a factor ~ 3.

However, scaling on the KNC still levels off when using all 61 cores and especially when using more than one thread per core. This is also due to memory bandwidth congestion. The tiled code does not suffer from this bottleneck. Since the loop over the tiles is only parallel in one spatial dimension, one starts to see load imbalances for many threads. Pluto can be forced to extract an additional level of parallelism,

but the resulting code is much harder to read and performance is lower. The theoretical bandwidth of the dual socket SB system is 102.4 GB/s (51.2 GB/s per socket), and that of the KNC is 352 GB/s. However, the STREAM benchmark measures 78.5 GB/s on the SB and 174 GB/s on the KNC, so the roofline fitted to the tiled smoother corresponds to 56.7 % and 42.9 % of STREAM bandwidth on the SB and the KNC, respectively. On the KNC, the ring bus network connecting the cores acts as a bandwidth bottleneck. This, together with overhead from error correcting codes (ECC), limits the effectively available bandwidth.

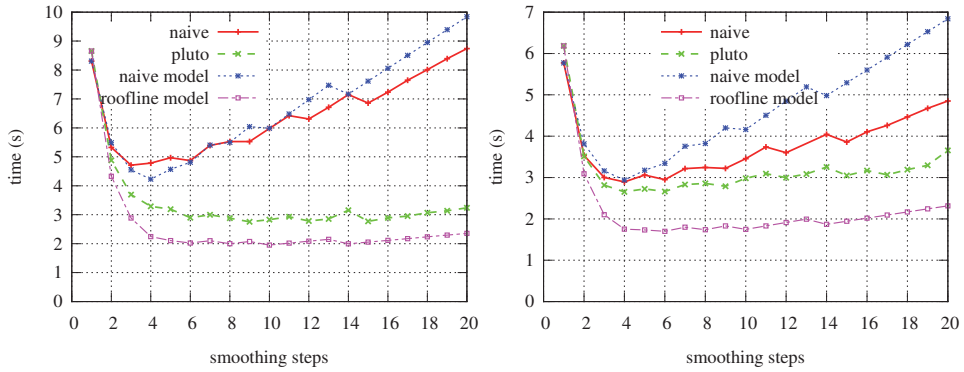
The dual socket SB system has a theoretical peak performance of 332.8 GFlop/s (2.6 GHz $\times$ 2 fma[1] $\times$ 4 SIMD $\times$ 16 cores) and 1073 GFlop/s (1.1 GHz $\times$ 2 fma $\times$ 8 SIMD $\times$ 61 cores) for the KNC. On the SB we get 23.9 % and on the KNC 7.11 % of the theoretical peak performance. To further improve performance, a number of optimizations could be considered. On the KNC, it is likely that prefetching and vectorization are suboptimal because of the complicated data access pattern of the tiled code. Manually writing SIMD intrinsics and adding software prefetching might improve performance, but this is outside the scope of this work. On the KNC, the shape of the tile was already chosen such that it is longer in the direction of the inner loop to get a more regular data access pattern and allow better vectorization. An additional requirement for reaching peak performance is an equal balance of multiply and add instructions since most modern CPUs have fused multiply-add instructions. Low level optimizations such as array padding, cache alignment, and manual loop unrolling are all left to the compiler.

Have a look at the theoretical roofline for the KNC architecture (Figures 1 and 5) and keep in mind the low arithmetic intensity of all steps in the multigrid code ($q_1(\omega\text{-Jac}) = 3/8$). The bandwidth bound for the smoother predicts a maximum of only 132 GFlop/s, while the machine has a peak performance of 1073 GFlop/s. Although we are still far from this peak for several reasons, this means tiling for data locality has huge potential.

5.3. Full *V*-cycle. The tiled smoother was incorporated in a multigrid V-cycle code. Figure 7 shows timings for a multigrid solve on a 2D $8191^2 = (2^{13}-1)^2$ Poisson problem with homogeneous Dirichlet boundary conditions. A relative stopping criterion was used with a tolerance of 10^{-12}. In these results, the number of smoothing steps refers to the presmoothing phase, and no postsmoothing is applied. In Figure 7 the timings are compared with predictions based on the performance model presented in section 3.2.3, the naive performance model, and in section 4, the performance model based on the roofline. For the roofline, the fit from Figure 5 was used. To match the model with the timings, the scaling factor relating Work Units to flops was determined by fitting the model through the first point of the curve.

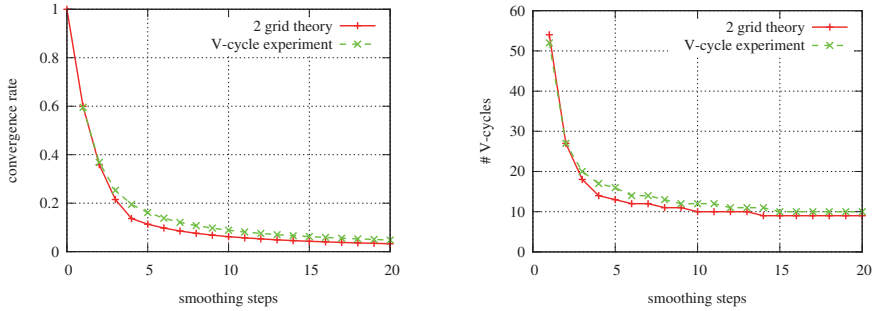
Some observations can be made from Figure 7. With the Pluto optimized smoother, initially the cost declines with an increasing number of smoothing steps and then increases slowly when adding more steps, unlike for the naive implementation of the smoother, where the cost starts to increase rapidly after the first few smoothing steps. As a result, the optimal number of smoothing steps is shifted to the right, completely in line with what was predicted by the performance model. For the SB experiments, the optimum shifts from $(3, 4.7\,\text{s})$ to $(15, 2.8\,\text{s})$, a $1.7\times$ speedup. At 20 smoothing steps, the total speedup is $2.7\times$. For the KNC experiments, the optimum shifts from $(4, 2.89\,\text{s})$ to $(4, 2.65\,\text{s})$, an improvement by 9 %. Here the benefit from tiling is much

[1] Fused multiply-add.



(a) Sandy Bridge (b) Xeon Phi

FIG. 7. *Timings for a full solve on an* 8191^2 *domain using* V*-cycles with a relative stopping tolerance* 10^{-12}*, both with the tiled and the naive smoother.* (a) *Experiments on the dual socket Sandy Bridge machine.* (b) *Similar experiment on the Xeon Phi. The timings are compared with the predictions from the analytical performance model, both the naive model and the model based on the roofline. The performance model uses bandwidth and floating point performance numbers taken from the fit to the experimentally measured roofline from Figure* 5*. The performance model gives a lower bound for the total solver time.*

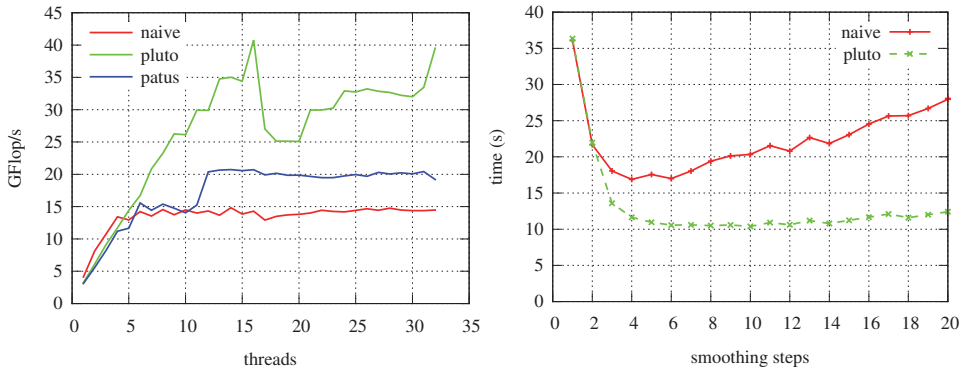


(a) Convergence rate (b) # V-cycles

FIG. 8. *Left: Comparison of the two-grid correction scheme convergence rate and the full* V*-cycle convergence rate. For the two-grid scheme, convergence is slightly faster than for the recursive multigrid solver, except for only a single smoothing step. Right: The corresponding number of* V*-cycles required to reach a* 10^{-12} *relative stopping criterion for both two-grid and multigrid.*

smaller due to the relatively good performance of the naive smoother, thanks to the high memory bandwidth; see Figure 6.

Our theoretical models seem to correspond well with the V-cycle experiments. One could expect the models to give a lower-bound for the time for two reasons. The model is based on a two-grid scheme instead of a recursive (V-cycle) solve, and, as shown in Figure 8 (a), the convergence rate for a two-grid scheme is typically slightly better than that of a V-cycle. The number of V-cycles required to reach the given tolerance is slightly underestimated by the model, as can be seen from Figure 8 (b), except when only one smoothing step is used. The other reason for expecting a lower-bound is that the roofline that was used in the model is a fit to our experiments from Figure 5. This fit is an upper-bound for the performance of the tiled smoother. However, in Figure 7, the naive model does not always bound the performance of the



(a) smoother scaling (b) V-cycle timing

FIG. 9. *Experiments with the tiled and the naive smoother on the dual socket Sandy Bridge machine for a 3D domain. Left: Scaling of* 10 *iterations of the* ω*-Jacobi smoother on a* 500^3 *domain. A comparison with Patus generated code is also given. Right: Timings for a full solve on a* 511^3 *domain using* V*-cycles with a relative stopping tolerance* 10^{-12}.

naive implementation from below. This could be explained by the fact that the model was scaled based on the first point (one smoothing step). An alternative reasoning might be that at coarser levels in the multigrid hierarchy, performance of the naive code improves because the entire domain fits in cache, leading to better data locality without having to tile the domain.

5.4. 3D results. From an application point of view, 3D codes are much more important, so without comparing with a theoretical model this section presents 3D results. All previous conclusions appear to be valid in the 3D case as well. Figure 9 (a) shows scaling of the ω-Jacobi smoother on a 500^3 domain on the SB compute node. Performance is less than in the 2D case, but again tiling the code with the help of Pluto leads to significant speedups compared to the naive (pure C) implementation, as well as compared to code generated by Patus. The Pluto tiled code uses $8 \times 8 \times 8 \times 64$ tiles (8 high in the time dimension). To avoid cache thrashing [18], the domain size was chosen as 500^3 rather than 512^3.

Figure 9 (b) shows the time for a full V-cycle solve on a 511^3 domain on the SB machine. The behavior is very similar to that of the 2D case. Using tiling, the optimum shifts from $(4, 16.9\,\text{s})$ to $(10, 10.3\,\text{s})$, a $1.64\times$ speedup.

For the 3D code, the arithmetic intensity is $q = 11/24$ flop/byte, which is approximately the same order as that of the 2D code (3/8 flop/byte). In more realistic applications, the stencil will likely include variable coefficients which also have to be loaded from memory. This adds additional floating point operations that were not required in the constant coefficient Poisson stencil. For a typical 2D 5-point stencil with a single scalar variable coefficient field, the arithmetic intensity is 13/32 flop/byte, and for a 3D 7-point stencil it is 17/32 flop/byte. Those values are all the same order as for the 2D constant coefficient case considered above, and hence tiling can improve performance; see also [42], where temporal blocking is applied over four iterations of the smoother for a variable coefficient Helmholtz problem.

For the 3D experiments on the KNC the Pluto generated code did not yield a speedup compared to the naive implementation. However, considering the low arithmetic intensity of the 3D code and the roofline of the KNC (see Figure 1), tiling

theoretically still has big potential. A better understanding of Xeon Phi performance bottlenecks is required to achieve this.

The tile sizes were selected manually. Finding good tile sizes is possible with some common sense: make sure a tile fits in cache, and a larger tile dimension along the innermost loop dimension can lead to better vectorization. Finding the optimal tile size is very hard as the search space is huge, and for this reason autotuning can take a very long time. A recent article [30] proposes a tile size selection strategy for the Pluto tool that is based on a performance model. This strategy takes into account the effects of vectorization and of temporal blocking.

6. Conclusions. The main contribution of this paper is a new multigrid performance model that takes into account communication with the slow main memory. This model also shows that there is a potential performance benefit from clever implementation of the smoother. By applying state-of-the-art compiler optimizations (tiling) to the smoother, the arithmetic intensity can indeed be increased and performance can be improved. We compare timings of a multigrid solver with such a tiled smoother with predictions using our performance model. This performance model is intended to serve as a motivating example of how one should reason about algorithm optimization—not just in terms of floating point operations, but also taking into account data movement. By considering the data movements, we came to the conclusion, both from our model and from the experiments, that, contrary to common belief, performing more smoothing steps actually pays off despite the extra floating point operations that are required but gives only a marginal improvement in convergence rate.

Tiling of stencil applications can lead to huge performance gains in, for instance, straightforward explicit time integration. However, when used in more complex algorithms such as geometric multigrid or Krylov solvers, only a few stencil operations are applied consecutively, and the benefit is less obvious. With our performance model, we hope to make the situation clearer and more predictive.

For our experiments we have restricted ourselves to plain C code and readily available code optimization tools without losing ourselves in low-level code optimizations. These tools, such as stencil compilers, loop optimizers, and automatic vectorizers, have improved a great deal over the last few years. We believe it is crucial for development cost and code maintainability that such tools be used since a hand-written tiled stencil code can become a nightmare very quickly. However, these tools are far from perfect, both in their usability and features.

Although all steps in the multigrid algorithm have low arithmetic intensity, we have focused only on the smoother. For the other steps, there is no easy way to improve data locality. Possible approaches to improving performance further could be to combine, for instance, the computation of the residual and the restriction with the presmoothing. Likewise, the interpolation, error correction, and postsmoothing could be combined into a single high arithmetic intensity kernel. An alternative approach to minimizing synchronization cost for all the steps might be the use of directed acyclic graphs [24] or dynamic task graphs [3] to better schedule the work over the different cores.

Our current experiments are limited to stencils on regular grids. Tiling of consecutive SpMVs on unstructured grids is much harder. However, the *sparse polyhedral framework* (SPF) [35, 25], an extension of *full sparse tiling* [34], attempts to generalize polyhedral loop transformations to codes with indirect memory references, as used in general SpMVs. While the standard polyhedral framework is restricted to compiling

time optimizations, the SPF allows run-time data reordering transformations using generated inspector/generator code. This approach might be used in the future to add tiling to smoothers on unstructured grids.

Optimization of the smoother in multigrid is related to optimization of the matrix powers kernel in s-step Krylov methods [31, 19]. However, the main difference is that the s vectors generated by the matrix powers kernel need to be stored in memory, whereas for a smoother only the last vector is required. Hence a smoother has more potential data reuse. Approaches proposed for optimization of the matrix powers kernel [10, 7] also apply for a smoother, but, at the moment, no optimized implementation of the matrix powers kernel for general sparse matrices is available.

Appendix A. Code example. The plain C implementation of the 2D Jacobi stencil applied nu times on an $N \times N$ domain, saved in memory with leading dimension `lda`, looks like

```
  for (int k=0; k<nu; k++) {
#pragma omp parallel for
    for (int i=1; i<=N; i++) {
      int lbv= i*lda+1; int ubv= i*lda+N;
      int ju = lbv+lda; int jb = lbv-lda;
      int jl = lbv-1;   int jr = lbv+1;
      if (k%2==0) {
#pragma ivdep
#pragma vector always
        for (int j=lbv; j<=ubv; j++) {
          w[j] = a[j] - D_omega*((4.0*a[j]-a[ju]-a[jl]-a[jb]-a[jr])
                                 *ih2 - b[j]);
          jr++; jl++; ju++; jb++;
        }
      } else {
#pragma ivdep
#pragma vector always
        for (int j=lbv; j<=ubv; j++) {
          a[j] = w[j] - D_omega*((4.0*w[j]-w[ju]-w[jl]-w[jb]-w[jr])
                                 *ih2 - b[j]);
          jr++; jl++; ju++; jb++;
        }
      }
    }
  }
  if (nu%2==1) swap(a,w);
```

The code tiled with Pluto looks like

```
#define ceild(n,d)  ceil(((double)(n))/((double)(d)))
#define floord(n,d) floor(((double)(n))/((double)(d)))
if ((N >= 1) && (nu >= 1)) {
 for (int t1=-1;t1<=floord(nu-1,32);t1++) {
  int lbp=max(ceild(t1,2),ceild(64*t1-nu+2,64));
  int ubp=min(floord(nu+N-1,64),floord(32*t1+N+31,64));
#pragma omp parallel for
  for (int t2=lbp;t2<=ubp;t2++) {
   for (int t3=max(max(0,ceild(t1-1,2)),ceild(64*t2-N-62,64));
        t3<=min(min(floord(nu+N-1,64),floord(32*t1+N+63,64)),
                floord(64*t2+N+62,64));t3++) {
    for (int t4=max(max(max(max(0,32*t1),64*t2-N),64*t3-N),
                    64*t1-64*t2+1);
         t4<=min(min(min(min(nu-1,32*t1+63),64*t2+62),64*t3+62),
                 64*t1-64*t2+N+63);t4++) {
#pragma loop_count min(1),max(64),avg(32)
```

```
16      for (int t5=max(max(64*t2,t4+1),-64*t1+64*t2+2*t4-63);
17           t5<=min(min(64*t2+63,t4+N),-64*t1+64*t2+2*t4);t5++) {
18       int lbv=(-t4+t5)*lda-t4+max(64*t3,t4+1);
19       int ubv=(-t4+t5)*lda-t4+min(64*t3+63,t4+N);
20       int t6l=lbv-1;      int t6r=lbv+1;
21       int t6u=lbv+lda;    int t6b=lbv-lda;
22       if (t4%2==0) {
23  #pragma loop_count min(1),max(64),avg(32)
24  #pragma ivdep
25  #pragma vector always
26         for (int t6=lbv;t6<=ubv;t6++) {
27          w[t6]=a[t6]-D_omega*((4.0*a[t6]-a[t6b]-a[t6l]-a[t6u]-a[t6r])
                                 *ih2-b[t6]);
28          t6l++; t6r++; t6u++; t6b++;
29       }} else {
30  #pragma loop_count min(1),max(64),avg(32)
31  #pragma ivdep
32  #pragma vector always
33         for (int t6=lbv;t6<=ubv;t6++) {
34          a[t6]=w[t6]-D_omega*((4.0*w[t6]-w[t6b]-w[t6l]-w[t6u]-w[t6r])
                                 *ih2-b[t6]);
35          t6l++; t6r++; t6u++; t6b++;
36  }}}}}}}}
37  if (nu%2==1) swap(a,w);
```

This code uses $64 \times 64 \times 64$ tiles, 64×64 in the spatial dimensions, and the iteration dimension is always taken to be equal to the smallest spatial dimension and is hence also 64. Note that since the tile size is known at compile time, the `loop_count` can be specified to help the compiler optimize software prefetching. The specific tiling approach used is described in detail in [1].

Acknowledgments. We would like to thank Siegfried Cools for help with the local Fourier analysis. Also, thanks to Zubair Wadood Bhatti and Sven Verdoolaege for interesting discussions.

REFERENCES

[1] V. Bandishti, I. Pananilath, and U. Bondhugula, *Tiling stencil computations to maximize parallelism*, in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society Press, Los Alamitos, CA, 2012, 40.

[2] R. Barrett, M. W. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, 2nd ed., SIAM, Philadelphia, 1994.

[3] Z. W. Bhatti, R. Wuyts, P. Costanza, D. Preuveneers, and Y. Berbers, *Efficient synchronization for stencil computations using dynamic task graphs*, Procedia Computer Science, 18 (2013), pp. 2428–2431; available online from http://www.sciencedirect.com/science/article/pii/S1877050913005590.

[4] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, *Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model*, in Compiler Construction, Springer, New York, 2008, pp. 132–146.

[5] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, *A practical automatic polyhedral parallelizer and locality optimizer*, in ACM SIGPLAN Notices, ACM, New York, 2008, pp. 101–113.

[6] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A Multigrid Tutorial*, 2nd ed., SIAM, Philadelphia, 2000.

[7] E. Carson, N. Knight, and J. Demmel, *Hypergraph partitioning for computing matrix powers*, in Proceedings of SIAM Workshop on Combinatorial Scientific Computing, SIAM, Philadelphia, 2011.

[8] M. Christen, O. Schenk, and H. Burkhart, *Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures*, in 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS), IEEE, Washington, DC, 2011, pp. 676–687.

[9] 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 (2009), pp. 129–159.

[10] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick, *Avoiding communication in sparse matrix computations*, in 2008 IEEE International Symposium on Parallel and Distributed Processing, IEEE, Washington, DC, 2008, pp. 1–12.

[11] C. Douglas, *Caching in with multigrid algorithms: Problems in two dimensions*, Internat. J. Parallel Emergent Distributed Syst., 9 (1996), pp. 195–204.

[12] C. Douglas, J. Hu, W. Karl, M. Kowarschik, U. Rüde, and C. Weiss, *Fixed and adaptive cache aware algorithms for multigrid methods*, in Multigrid Methods VI: Proceedings of the Sixth European Multigrid Conference (Gent, Belgium, 1999), Lect. Notes Comput. Sci. Eng. 14, Springer-Verlag, 2000, pp. 87–93.

[13] C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiss, *Cache optimization for structured and unstructured grid multigrid*, Electron. Trans. Numer. Anal., 10 (2000), pp. 21–40.

[14] H. C. Elman, D. J. Silvester, and A. J. Wathen, *Finite Elements and Fast Iterative Solvers: With Applications in Incompressible Fluid Dynamics*, Oxford University Press, Oxford, UK, 2005.

[15] P. Ghysels, P. Kłosiewicz, and W. Vanroose, *Improving the arithmetic intensity of multigrid with the help of polynomial smoothers*, Numer. Linear Algebra Appl., 19 (2012), pp. 253–267.

[16] A. Greenbaum, *Iterative Methods for Solving Linear Systems*, Frontiers Appl. Math. 17, SIAM, Philadelphia, 1997.

[17] W. Hackbusch, *Multigrid Methods and Applications*, Springer Ser. Comput. Math. 4, Springer-Verlag, Berlin, 1985.

[18] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*, CRC Press, Boca Raton, FL, 2010.

[19] M. Hoemmen, *Communication-Avoiding Krylov Subspace Methods*, Ph.D. thesis, EECS Department, University of California Berkeley, Berkeley, CA, 2010.

[20] Intel Corporation, *Intel Instruction Set Architecture Extensions*, http://software.intel.com/en-us/intel-isa-extensions.

[21] Intel Corporation, *Intel® Xeon Phi™ Coprocessor Vector Microarchitecture*, http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-vector-microarchitecture.

[22] M. Kowarschik, *Data Locality Optimizations for Iterative Numerical Algorithms and Cellular Automata on Hierarchical Memory Architectures*, Ph.D. thesis, Universität Erlangen-Nürnberg, Erlangen, Nürnberg, Germany, 2004.

[23] M. Kowarschik, U. Rüde, C. Weiss, and W. Karl, *Cache-aware multigrid methods for solving Poisson's equation in two dimensions*, Computing, 64 (2000), pp. 381–399.

[24] J. Kurzak, H. Ltaief, J. Dongarra, and R. M. Badia, *Scheduling dense linear algebra operations on multicore processors*, Concurrency and Computation: Practice and Experience, 22 (2010), pp. 15–44.

[25] A. LaMielle and M. M. Strout, *Enabling Code Generation within the Sparse Polyhedral Framework*, Tech. rep. CS-10-102, Colorado State University, Fort Collins, CO, 2010.

[26] C. Lengauer, *Loop parallelization in the polytope model*, in CONCUR'93, Springer, New York, 1993, pp. 398–416.

[27] G. H. Loh, *3D-stacked memory architectures for multi-core processors*, in ACM SIGARCH Computer Architecture News, IEEE Computer Society, Los Alamitos, CA, 2008, pp. 453–464.

[28] A. McAdams, Y. Zhu, A. Selle, M. Empey, R. Tamstorf, J. Teran, and E. Sifakis, *Efficient elasticity for character skinning with contact and collisions*, ACM Trans. Graphics, 30 (2011), 37.

[29] J. D. McCalpin, *STREAM: Sustainable Memory Bandwidth in High Performance Computers*, http://www.cs.virginia.edu/stream/.

[30] S. Mehta, G. Beeraka, and P.-C. Yew, *Tile size selection revisited*, ACM Trans. Architecture Code Optim., 10 (2013), p. 35.

[31] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick, *Minimizing communication in sparse matrix solvers*, in Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, ACM, New York, 2009, 36.

[32] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fourth Edition: The Hardware/Software Interface*, The Morgan Kaufmann Series in Computer Architecture and Design, Morgan Kaufmann, San Francisco, CA, 2008.
[33] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed., SIAM, Philadelphia, 2003.
[34] M. M. Strout, L. Carter, J. Ferrante, and B. Kreaseck, *Sparse tiling for stationary iterative methods*, Internat. J. High Performance Comput. Appl., 18 (2004), pp. 95–113.
[35] M. M. Strout, A. LaMielle, L. Carter, J. Ferrante, B. Kreaseck, and C. Olschanowksy, *An Approach for Code Generation in the Sparse Polyhedral Framework*, Tech. rep. CS-10-102, Colorado State University, Fort Collins, CO, 2013.
[36] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, *The pochoir stencil compiler*, in Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures, ACM, New York, 2011, pp. 117–128.
[37] J. Treibig, *Efficiency Improvements of Iterative Numerical Algorithms on Modern Architectures*, Ph.D. thesis, Universität Erlangen-Nürnberg, Erlangen, Nürnberg, Germany, 2008.
[38] J. Treibig, G. Wellein, and G. Hager, *Efficient multicore-aware parallelization strategies for iterative stencil computations*, J. Comput. Sci., 2 (2011), pp. 130–137.
[39] U. Trottenberg, C. Oosterlee, and A. Schüller, *Multigrid*, Academic Press, New York, 2001.
[40] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor, *Polyhedral parallel code generation for cuda*, ACM Trans. Archit. Code Optim., 9 (2013), 54.
[41] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske, *Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization*, in 33rd Annual IEEE International Computer Software and Applications Conference, IEEE, Washington, DC, 2009, pp. 579–586.
[42] S. Williams, D. D. Kalamkar, A. Singh, A. M. Deshpande, B. Van Straalen, M. Smelyanskiy, A. Almgren, P. Dubey, J. Shalf, and L. Oliker, *Optimization of geometric multigrid for emerging multi- and manycore processors*, in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society Press, Los Alamitos, CA, 2012, 96.
[43] S. Williams, A. Waterman, and D. A. Patterson, *Roofline: An insightful visual performance model for multicore architectures*, Comm. ACM, 52 (2009), pp. 65–76.