Modeling the performance of geometric multigrid on many-core computer architectures

P. Ghysels
W. Vanroose

Report 09.2013.1, Sept 2013

This work is funded by Intel and by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT).

MODELING THE PERFORMANCE OF GEOMETRIC MULTIGRID ON MANY-CORE COMPUTER ARCHITECTURES

PIETER GHYSELS*†‡ AND WIM VANROOSE*

Abstract. The basic building blocks of the classic geometric multigrid algorithm are all essentially stencil computations and have a low ratio of executed floating point operations per byte fetched from memory. On modern computer architectures, such computational kernels are typically bounded 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 problem, we present a performance model for the multigrid V-cycle solver based on the tiled smoother. For increasing number 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 2D and 3D simulations on Intel Sandy Bridge and Intel Xeon Phi architectures. The actual performance is compared with the theoretical predictions.

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 in more on-chip parallelism, sustaining Moore's self-fulfilling prophecy. For instance Intel AVX instructions [16] operate on 8 single precision floating point numbers simultaneously, while the Intel Xeon Phi accelerator chip has 512 bit wide vector registers [17], 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. This has two main reasons: 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 non-uniform memory access 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 [23] benchmark only achieves 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 [21].

Many scientific computer codes are bandwidth limited computational kernels. Examples of such kernels are stencil applications, dot products, vector additions and so on. These are the typical building blocks for algorithms from sparse linear algebra, such as Jacobi iteration [2], geometric multigrid [6, 27, 14] and Krylov solvers [25, 13] that appear in scientific codes that solve problems modeled by partial differential equations [11]. Since the underlying kernels are highly bandwidth limited, so are the algorithms build 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 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 not bound by memory bandwidth but by the floating point performance of the hardware.

Fusing several computational kernels in 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

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

†Intel ExaScience Lab - Kapeldreef 75, B-3001 Leuven - Belgium

‡Lawrence Berkeley National Laboratory - Computational Research Division, 1 Cyclotron Road Berkeley, CA 94720 - USA (pghysels@lbl.gov)

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. Besides 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 compiler such as Pochoir [26] and Patus [7] 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, SIMD vectorization and so on. All these optimization have also been studied for instance in [28]. 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 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 2D and 3D simulations are presented on modern multi-core 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) accelerator.

Geometric multigrid is a very widely used algorithm for the solution of systems of equations, either as a standalone 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 [8, 10, 9, 18]. Also, the idea to apply tiling or blocking over multiple smoothing steps was already used previously in [12] and in [28]. However, 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. These benefits are especially dramatic on multi-core 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, like for instance real-time animation of characters for a new computer animated movie [22], this better scalability might make a crucial difference.

There is a growing awareness that minimization of flops should no longer drive algorithm design, but rather minimization of data movement and synchronization. The behavior of geometric multigrid applied to the 2D Poisson problem is well known [6] in terms of floating point operations 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.

The paper is outlined as follows. Section 2 reviews the roofline model [29], 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.

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, called the arithmetic intensity, can be used to predict the floating point performance

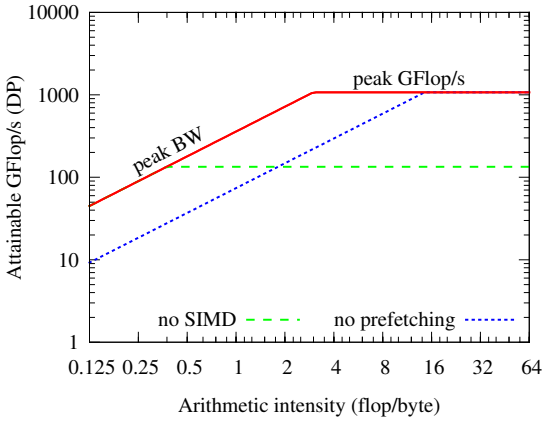


FIG. 2.1. *The roofline model predicting floating point performance based on arithmetic intensity. Low arithmetic intensity kernels are bandwidth bound whereas high arithmetic intensity kernels are compute bound. Many lower rooflines can be added which can be overcome by corresponding code optimizations.*

of the kernel through the roofline model [29, 24]:

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

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 2.1. The KNC has a peak double precision floating point performance of 1073 GFlop/s (see Section 5 for more info 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 in 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 can not 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 out of repeated application of stencils 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

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

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 $A^h_{\text{2D}} \in \mathbb{R}^{N\times N}$ for the 2D Poisson problem can be written as a Kronecker product of the one dimensional operators as

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

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

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

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 guess 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

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

where D is the diagonal of the operator A^h_{2D} and ω is a weight, usually taken to be $2/3$ in 1D and $4/5$ in 2D. 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

```
c = h^2/4*omega;
for (k = 1 to nu) {
  for (i = 1 to n)
    for (j = 1 to n)
      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]);
  swap(w, v);
}
```

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

LISTING 2
Residual Calculation

```
for (i = 1 to n)
  for (j = 1 to n)
    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

Listing 3
Restriction

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

and 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

Listing 4
Interpolation

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

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 follow: 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 $I_{2h}^h e^{2h}$, 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 is called pre-smoothing, the last ν_2 applications post-smoothing.

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 have 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 $q = 11/40$

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: $r^{2h} \leftarrow I_h^{2h} r^h$
7: $e^{2h} \leftarrow 0$
8: $e^{2h} \leftarrow V\text{-cycle}^{2h}(e^{2h}, r^{2h})$
9: $e^h \leftarrow I_{2h}^h e^{2h}$
10: $v^h \leftarrow v^h + e^h$
11: Relax ν_2 iterations: $v^h \leftarrow (1 - \omega D^{-1} A^h) v^h + \omega D^{-1} f^h$

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, we can 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. 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 $e^{h\,(i+1)} = \mathrm{TG}\, e^{h\,(i)}$. 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

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

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 $A^h_{1\mathrm{D}}$ and applying the TG operator on these eigenmodes. Since the smoother R and A^h_{1D} share the same eigenvectors, the action of ν smoother steps with the iteration matrix $R = (I - \omega D^{-1} A^h_{1\mathrm{D}})$ on an eigenvector is

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

The restriction operator I^{2h}_h works on pairs of eigenmodes, so-called complementary modes with the complementary eigenmode of w^h_k defined as $w^h_{k'} \equiv w^h_{n-k}$. The restriction operator maps two of these eigenmodes $\mathrm{span}\{w^h_k, w^h_{k'}\}$ to $\mathrm{span}\{w^{2h}_k\}$ and the action on these modes is

$$I^{2h}_h w^h_k = c^h_k w^{2h}_k \,, \quad k = 1, \ldots, \frac{1}{2} \quad \text{and} \quad I^{2h}_h w^h_{k'} = -s^h_k w^{2h}_k \,, \quad k = 1, \ldots, \frac{n}{2} - 1, \tag{3.12}$$

where $c^h_k = \cos^2(\frac{k\pi}{2n})$ and $s^h_k = \sin^2(\frac{k\pi}{2n})$. The interpolation maps $\mathrm{span}\{w^{2h}_k\}$ to the space of complementary modes $\mathrm{span}\{w^h_k, w^h_{k'}\}$ as

$$I^h_{2h} w^{2h}_k = c^h_k w^h_k - s^h_k w^h_{k'} \,, \quad k = 1, \ldots, \frac{n}{2} - 1 \,. \tag{3.13}$$

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

$$\mathrm{TG} \begin{bmatrix} w^h_k \\ w^h_{k'} \end{bmatrix} = \left(I - \left(\begin{bmatrix} c^h_k \\ -s^h_k \end{bmatrix} \otimes \begin{bmatrix} c^h_k & -s^h_k \end{bmatrix} \right) \frac{1}{\lambda^{2h}_k} \begin{bmatrix} \lambda^h_k & \\ & \lambda^h_{k'} \end{bmatrix} \right) \begin{bmatrix} w^h_k \\ w^h_{k'} \end{bmatrix} \,, \quad k = 1, \ldots, \frac{n}{2} - 1 \,. \tag{3.14}$$

Simplifying this and including the smoother leads to

$$\mathrm{TG} \begin{bmatrix} w^h_k \\ w^h_{k'} \end{bmatrix} = \begin{bmatrix} s^h_k & s^h_k \\ c^h_k & c^h_k \end{bmatrix} \begin{bmatrix} \left(1 - 2\omega \sin^2\left(\frac{k\pi}{2n}\right)\right)^\nu & 0 \\ 0 & \left(1 - 2\omega \sin^2\left(\frac{(n-k)\pi}{2n}\right)\right)^\nu \end{bmatrix} \begin{bmatrix} w^h_k \\ w^h_{k'} \end{bmatrix}, \; k = 1, \ldots, \frac{n}{2} - 1 \,. \tag{3.15}$$

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

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

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

$$T^h_k = \begin{bmatrix} c^h_k & -s^h_k \end{bmatrix} \quad \text{and} \quad T^h_{k,\ell} = T^h_k \otimes T^h_\ell = \begin{bmatrix} c^h_k c^h_\ell & -c^h_k s^h_\ell & -s^h_k c^h_\ell & s^h_k s^h_\ell \end{bmatrix} \,, \tag{3.17}$$

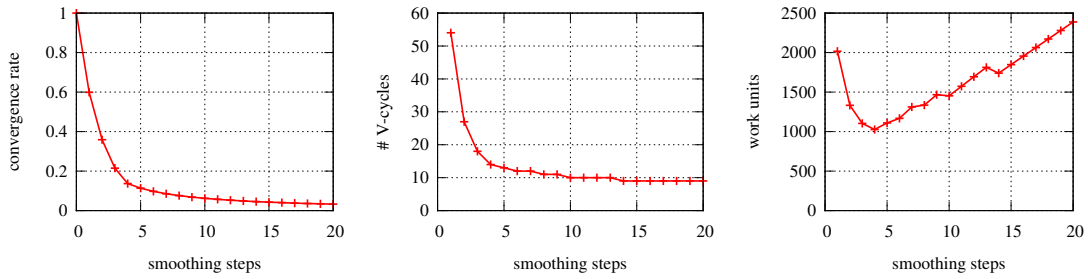
where T^h_k is the action of the 1D interpolation on an eigenmode and $T^h_{k,\ell}$ its 2D equivalent. Also, let

$$W^h_{k,l} = \begin{bmatrix} w^h_{k,\ell} & w^h_{k',\ell} & w^h_{k,\ell'} & w^h_{k',\ell'} \end{bmatrix}^T \quad \text{and} \quad \Lambda^h_{k,\ell} = \begin{bmatrix} \lambda^h_{k,\ell} & \lambda^h_{k',\ell} & \lambda^h_{k,\ell'} & \lambda^h_{k',\ell'} \end{bmatrix}^T \,. \tag{3.18}$$

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

$$\begin{aligned} {}^{2\mathrm{D}}\,\mathrm{TG}\, W^h_{k,l} = \left(I - \left(\left(T^h_{k,\ell}\right)^T \otimes T^h_{k,\ell} \right) \frac{1}{\lambda^{2h}_{k,\ell}} \mathrm{diag}\left(\Lambda^h_{k,\ell}\right) \right) \left(I - 2\omega \,\mathrm{diag}\left(\Lambda^h_{k,\ell}\right) \right)^\nu W^h_{k,l} \,, \\ k = 1, \ldots, \frac{n}{2} - 1 \,, \quad \ell = 1, \ldots, \frac{n}{2} - 1 \,. \end{aligned} \tag{3.19}$$

The maximum eigenvalue of this 4×4 matrix ${}^{2\mathrm{D}}\,\mathrm{TG}$ 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 3.1 (left) shows this convergence rate as a function of the number of smoothing steps ν for the ω-Jacobi smoother.



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

FIG. 3.1. *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.*

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

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

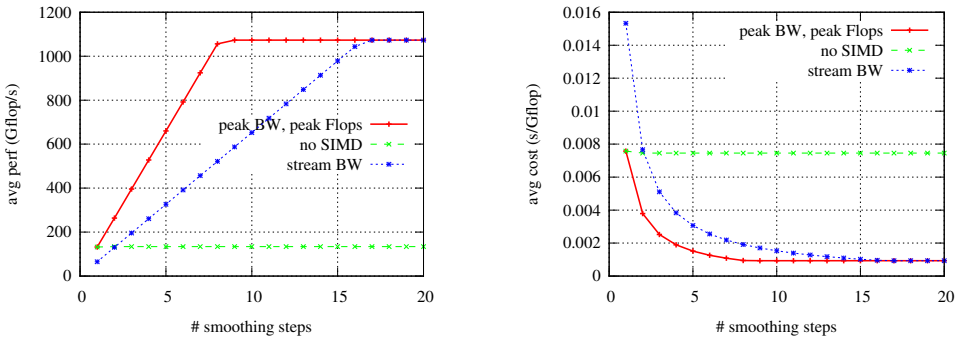
The number of iterations to reach a tolerance of $\epsilon = 10^{-12}$ as a function of the number of smoothing steps is given in the middle panel of Figure 3.1. As the number of smoothing steps increases, fewer iterations are required. However, if the cost of additional smoothing steps is taken into 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

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

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 3.1 (c). The rising cost reflects that the application of only a few (4) smoothing steps (pre+post smoothing) 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, equation (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 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 either 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



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

FIG. 4.1. *(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 number of steps the vector unit can fully be exploited. (b) This model then translates into a lowering average cost per SpMV in the kernel until peak performance is reached. This diminishing cost is the result of lowering communications costs.*

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 to be read back again for the next step. It is to be expected that for increasing number of smoothing steps, a trade-off will play between the reduced communication costs and the loss of effectiveness for the multigrid algorithm.

In Figure 4.1 (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 SpMV's gives perfect data reuse, which is probably a good approximation for a small number of steps. The arithmetic intensity, on 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 4.1 (b). The cost per SpMV diminishes with each additional multiplication until we hit peak performance. Figure 4.1 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 max 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 of the bandwidth and vectorization on the performance.

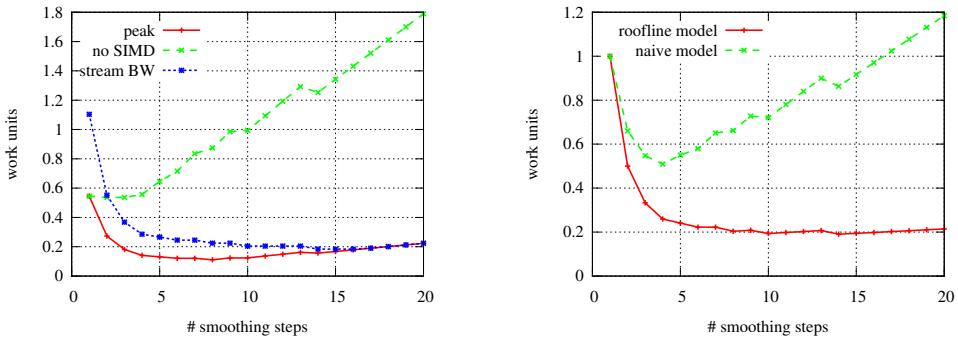
Consider again the multigrid cost model with fixed cost for each smoother application as given by equation (3.21) and shown in Figure 3.1. We now let the cost of applying a stencil depend on the arithmetic intensity of the specific stencil operation. For instance the cost of interpolating the coarse 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



(a) MG cost model based on roofline

(b) Roofline-based vs naive cost model

FIG. 4.2. *(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 function of the number of pre-smoothing steps ν_1 and without post-smoothing, 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 small number of smoothing steps. (b) Comparing the multigrid cost model based on the arithmetic intensity with the naive model.*

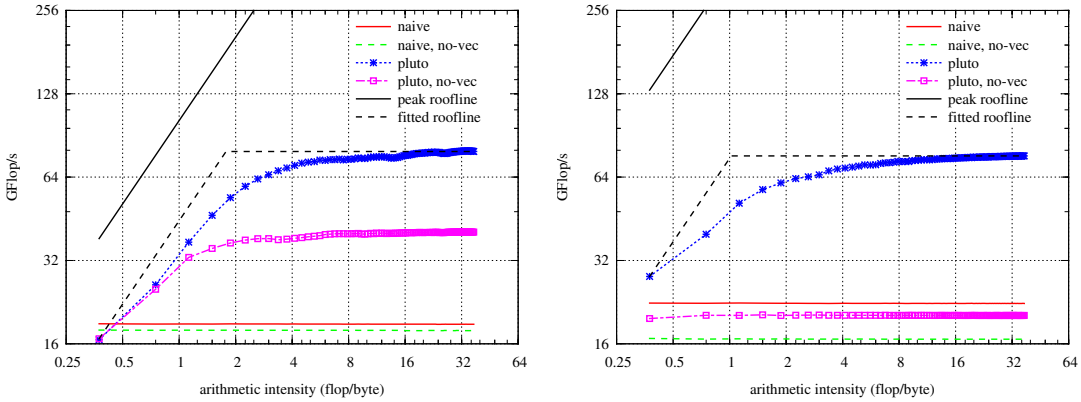
for the cost of the (pre-)smoother

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

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 4.2 (a). Whereas the original model showed 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 4.2 (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 4.2 (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 pre-smoothing with ν steps was considered because post and pre-smoothing can be combined since the post-smoothing step of one V-cycle is immediately followed by the pre-smoothing of the next cycle. However, in the performance model we need to distinguish between pre-smoothing and post-smoothing since they are in different function calls and splitting them alters the arithmetic intensity.

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.



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

(b) Intel Xeon Phi, 61 × 4 threads

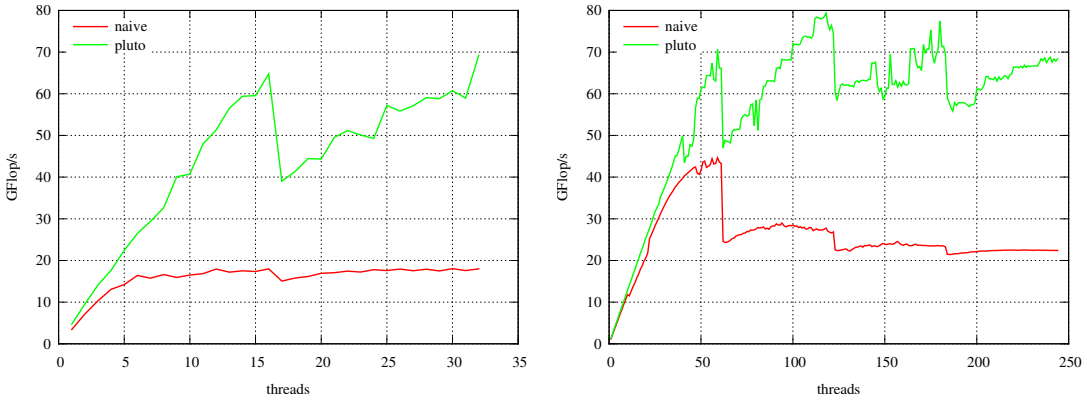
FIG. 5.1. *(a) The roofline as experimentally measured with the Jacobi smoother, tiled with Pluto, on a dual-socket Intel Sandy Bridge machine, using* 32 *threads. (b) Similar experiment on an Intel Xeon Phi using* 244 *threads. The smoother is applied on an* 8000^2 *domain. The naive code has a fixed arithmetic intensity, while for the Pluto tiled code the arithmetic intensity and the performance increases as more smoothing steps are performed. Both the naive code and the tiled code are run with and without auto-vectorization. Note that vectorization only pays off at higher arithmetic intensity.*

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 [20], 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 pragma's. 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 pragma's around the outermost loop. Furthermore, the resulting code is made amenable for vectorization by adding the compiler pragma's `ivdep` and `vector always`. This means vectorization relies completely on the compilers (guided) auto-vectorization 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. 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 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 and optimized them for data locality and parallelism with Pluto. See Appendix A for an example piece of code generated by Pluto. Figure 5.1 shows the performance for different numbers of repeated application of these smoothers. Figure 5.1 (a) shows an experiment on a dual socket Intel Sandy Bridge (SB) compute node with 16 hardware cores in total and Figure 5.1 (b) shows the same experiment on an Intel Xeon Phi (KNC generation) with 61 cores*. For the tiled code

*Our test system is the SE10 Intel Xeon Phi and the corresponding host system as described in: http://www.intel.com/content/dam/www/public/us/en/documents/performance-briefs/



(a) Dual socket Intel Sandy Bridge

(b) Intel Xeon Phi

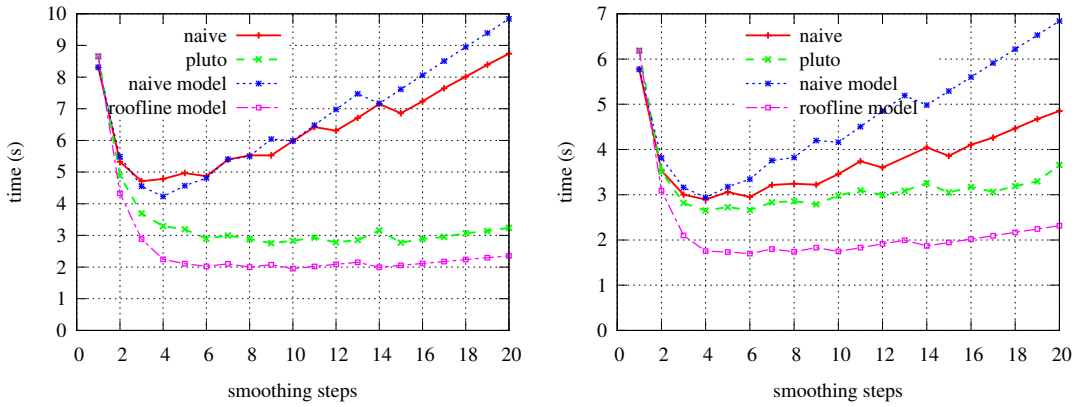
FIG. 5.2. *Comparison of the scalability for* 10 *iterations of the Jacobi smoother on an* 8000^2 *domain for both the naive implementation and the Pluto tiled code. The Sandy Bridge can run* 2 *threads per core using hyper-threading and the Xeon Phi can run* 4 *threads per core. Especially on the general purpose Xeon, the tiled code scales significantly better due to the lower bandwidth usage compared to the naive 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 therefor 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 5.1 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 5.1 shows both experiments with and without auto-vectorization 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 $1.97\times$ is a reasonable fraction of the theoretical factor 4 that can be attained for double precision calculations. 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 5.2 shows the parallel scalability of applying 10 smoothing steps with both the naive implementation and the tiled code on the two test machines, left the SB and right the KNC. The SB has a total of 16 hardware cores but can run 2 threads per core using hyper-threading. 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

`xeon-phi-product-family-performance-brief.pdf`



(a) Sandy Bridge (b) Xeon Phi

FIG. 5.3. *Timings for a full solve on a* 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 5.1. The performance model gives a lower bound for the total solver time.*

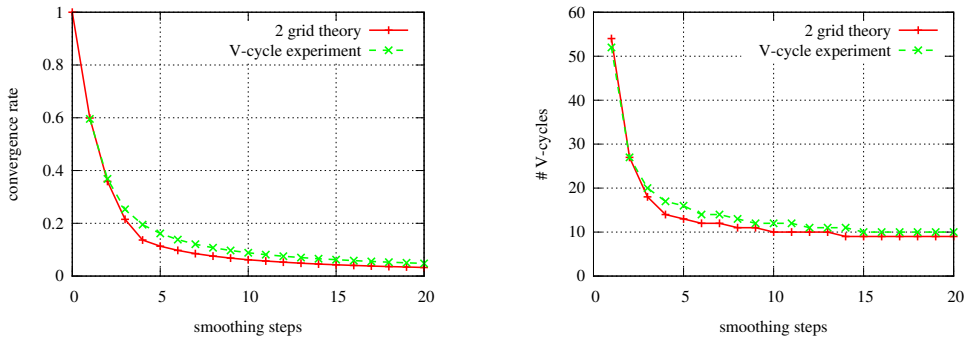
30 threads on the SB. Hence, for the naive implementation, the KNC outperforms a dual socket SB by a factor 2.48 ×.

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 less. 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.

The dual socket SB system has a theoretical peak performance of 332.8 GFlop/s (2.6 GHz × 2 fma[†] × 4 SIMD × 16 cores) and 1073 GFlop/s (1.1 GHz × 2 fma × 8 SIMD × 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 sub-optimal 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 CPU's 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 2.1 and 5.1) 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 a huge potential.

[†]Fused multiply-add.



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

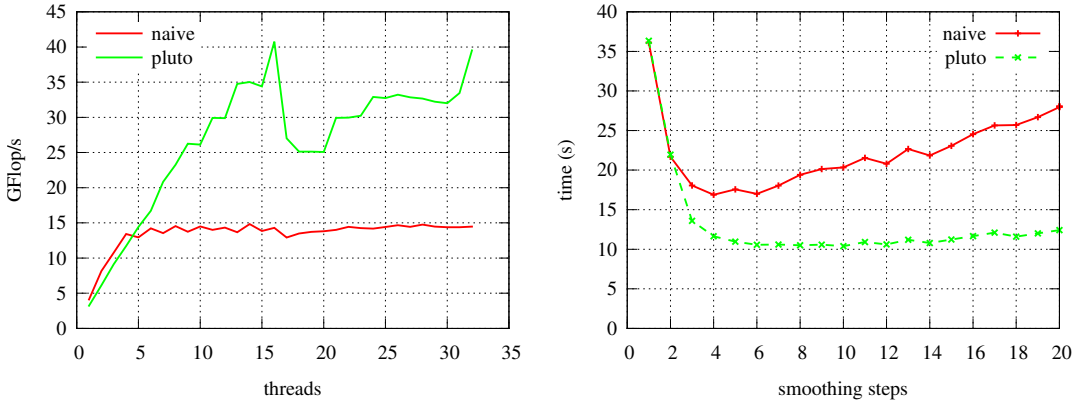
FIG. 5.4. *Left: Comparison of the two-grid correction scheme convergence rate and the full V-cycle convergence rate. For the two-grid scheme, convergence is slightly faster than for the recursive multigrid solver, except for only a single smoothing step. Right: The corresponding number of V-cycles required to reach a 10^{-12} relative stopping criterion for both two-grid and multigrid.*

5.3. Full V-cycle. The tiled smoother was incorporated in a multigrid V-cycle code. Figure 5.3 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 pre-smoothing phase and no post-smoothing is applied. In Figure 5.3 the timings are compared with predictions based on the performance model presented in Sections 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 5.1 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 5.3. With the Pluto optimized smoother, initially the cost declines with 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 $(3, 4.7\,\mathrm{s})$ to $(15, 2.8\,\mathrm{s})$, a $1.7\times$ speedup. At 20 smoothing steps, the total speedup is $2.7\times$. For the KNC experiments, the optimum shifts from $(4, 2.89\,\mathrm{s})$ to $(4, 2.65\,\mathrm{s})$, an improvement by $9\,\%$. Here the benefit from tiling is much smaller due to the relatively good performance of the naive smoother, thanks to the high memory bandwidth, see Figure 5.2.

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 5.4 (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 5.4 (b), except when only 1 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 5.1. This fit is an upper-bound for the performance of the tiled smoother. However, in Figure 5.3, the naive model does not always bound the performance of the naive implementation from below. This could be explained by the fact that the model was scaled based on the first point (1 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 5.5 (a) shows scaling of the ω-Jacobi



(a) smoother scaling

(b) V-cycle timing

FIG. 5.5. *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. Right: Timings for a full solve on a* 511^3 *domain using* V*-cycles with a relative stopping tolerance* 10^{-12}.

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. The Pluto tiled code uses $8 \times 8 \times 8 \times 64$ tiles (8 high in the time dimension). To avoid cache thrashing [15] the domain size was chosen as 500^3 rather than 512^2.

Figure 5.5 (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 the 2D case. Using tiling, the optimum shifts from $(4, 16.9\,\mathrm{s})$ to $(10, 10.3\,\mathrm{s})$, a $1.64\times$ speedup.

For the 3D code, the arithmetic intensity is approximately the same as for the 2D code. In more realistic applications, the stencil will likely include variable coefficients which also have to loaded from memory. This also adds additional floating point operations that were not required in the Poisson stencil and hence we expect that also in those cases the arithmetic intensity will be of the same order and tiling can improve performance, see also [28].

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 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 only give 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 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 loosing ourselves in low level code optimizations. These tools, such as stencil compilers, loop optimizers, automatic vectorizers etc, 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 really quickly. However, these tools are far from perfect, both in there usability and features.

Although all steps in the multigrid algorithm have low arithmetic intensity, we have only focused on the smoother. For the other steps, there is no easy way to improve data locality. Possible approaches to improve performance further could be to combine for instance the computation of the residual and the restriction with the pre-smoothing. Likewise, the interpolation, error correction and post-smoothing could be combined in a single high arithmetic intensity kernel. An alternative approach to minimize synchronization cost for all the steps might be the use of directed acyclic graphs [19] or dynamic task graphs [3] to better schedule the work over the different cores.

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. 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.

Appendix A. Code example. A simple 1D heat stencil might look like

```
#pragma scop
for (int t=0; t<T; t++)
  for (int i=1; i<N+1; i++)
    a[t+1][i]=0.125*(a[t][i+1]-2.0*a[t][i]+a[t][i-1]);
#pragma endscop
```

and when this piece of code marked by the `scop` and `endscop` pragma's is passed to the Pluto tool it becomes

```
for (t1=-1;t1<=floord(nu-1,16);t1++) {
  lbp=max(ceild(t1,2),ceild(32*t1-nu+2,32));
  ubp=min(floord(nu+N-1,32),floord(16*t1+N+15,32));
#pragma omp parallel for private(lbv,ubv)
  for (t2=lbp;t2<=ubp;t2++) {
    for (t3=max(max(max(0,16*t1),32*t2-N),32*t1-32*t2+1);
          t3<=min(min(min(nu-1,16*t1+31),32*t2+30),32*t1-32*t2+N+31); t3++)
#pragma ivdep
#pragma vector always
      for (t4=max(max(32*t2,t3+1),-32*t1+32*t2+2*t3-31);
           t4<=min(min(32*t2+31,t3+N),-32*t1+32*t2+2*t3); t4++)
        a[t3+1][-t3+t4]=0.125*(a[t3][-t3+t4+1]-2.0*a[t3][-t3+t4]+a[t3][-t3+t4-1]);
}
```

This is with the default tile size 32×32. When not all timesteps are required, but just the last, the time index `t3+1` or `t3` can be replaced by for instance `(t3+1) mod 2` and `t3 mod 2` respectively, so that only two grids need to be stored.

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, 2012, p. 40.

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

[3] Z. W. Bhatti, R. Wuyts, P. Ghysels, D. Preuveneers, and Y. Berbers, *Efficient Synchronization for Stencil Computations using Dynamic Task Graphs*. In progress.

[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, 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, 2008, pp. 101–113.

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

[7] M. Christen, O. Schenk, and H. Burkhart, *Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures*, in Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International, IEEE, 2011, pp. 676–687.
[8] C. Douglas, *Caching in with multigrid algorithms: problems in two dimensions*, International Journal of Parallel, Emergent and Distributed Systems, 9 (1996), pp. 195–204.
[9] 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, held in Gent, Belgium, September 27-30, 1999, vol. 14, Springer Verlag, 2000, p. 87.
[10] C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiss, *Cache optimization for structured and unstructured grid multigrid*, Electronic Transactions on Numerical Analysis, 10 (2000), pp. 21–40.
[11] H. C. Elman, D. J. Silvester, and A. J. Wathen, *Finite Elements and Fast Iterative Solvers: with Applications in Incompressible Fluid Dynamics: with Applications in Incompressible Fluid Dynamics*, Oxford University Press, 2005.
[12] P. Ghysels, P. Kłosiewicz, and W. Vanroose, *Improving the arithmetic intensity of multigrid with the help of polynomial smoothers*, Numerical Linear Algebra with Applications, 19 (2012), pp. 253–267.
[13] A. Greenbaum, *Iterative methods for solving linear systems*, vol. 17, Society for Industrial & Applied, 1997.
[14] W. Hackbusch, *Multi-grid methods and applications*, vol. 4, Springer-Verlag Berlin, 1985.
[15] G. Hager and G. Wellein, *Introduction to high performance computing for scientists and engineers*, CRC Press, 2010.
[16] Intel Corporation, *Intel Instruction Set Architecture Extensions*. http://software.intel.com/en-us/intel-isa-extensions.
[17] ———, *Intel® Xeon Phi™ Coprocessor Vector Microarchitecture*. http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-vector-microarchitecture.
[18] 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.
[19] 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.
[20] C. Lengauer, *Loop parallelization in the polytope model*, in CONCUR'93, Springer, 1993, pp. 398–416.
[21] G. H. Loh, *3d-stacked memory architectures for multi-core processors*, in ACM SIGARCH Computer Architecture News, IEEE Computer Society, 2008, pp. 453–464.
[22] A. McAdams, Y. Zhu, A. Selle, M. Empey, R. Tamstorf, J. Teran, and E. Sifakis, *Efficient elasticity for character skinning with contact and collisions*, in ACM Transactions on Graphics (TOG), ACM, 2011, p. 37.
[23] J. D. McCalpin, *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. http://www.cs.virginia.edu/stream/.
[24] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*, Morgan Kaufmann, 2008.
[25] Y. Saad, *Iterative methods for sparse linear systems*, Society for Industrial Mathematics, 2003.
[26] 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, 2011, pp. 117–128.
[27] U. Trottenberg, C. Oosterlee, and A. Schüller, *Multigrid*, Academic Pr, 2001.
[28] 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, 2012, p. 96.
[29] S. Williams, A. Waterman, and D. A. Patterson, *Roofline: An insightful Visual Performance model for multicore Architectures*, Communications of the ACM, 52 (2009), pp. 65–76.