



Improving the arithmetic intensity of multigrid with the help of polynomial smoothers

P. Ghysels
P. Kłosiewicz
W. Vanroose

Report 11.2011.1, November 2011

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



Improving the arithmetic intensity of multigrid with the help of polynomial smoothers

P. Ghysels^{1,2*}, P. Kłosiewicz^{1,2}, W. Vanroose¹

¹*Department of Mathematics and Computer Science,
University of Antwerp, Middelheimlaan 1, B-2020 Antwerp, Belgium*

²*Intel ExaScience Lab, Kapeldreef 75, B-3001 Leuven, Belgium*

SUMMARY

The basic building blocks of a classic multigrid algorithm, which are essentially stencil computations, all have a low ratio of executed floating point operations per byte fetched from memory. This important ratio can be identified as the arithmetic intensity. Applications with a low arithmetic intensity are typically bounded by memory traffic and achieve only a small percentage of the theoretical peak performance of the underlying hardware. We propose a polynomial Chebyshev smoother, which we implement using cache-aware tiling, to increase the arithmetic intensity of a multigrid V-cycle. This tiling approach involves a trade-off between redundant computations and cache misses. Unlike common conception, we observe optimal performance for higher degrees of the smoother. The higher degree polynomial Chebyshev smoother can be used to smooth more than just the upper half of the error frequencies, leading to better V-cycle convergence rates. Smoothing more than the upper half of the error spectrum allows a more aggressive coarsening approach where some levels in the multigrid hierarchy are skipped. Copyright © 2011 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: multigrid; Chebyshev iteration; arithmetic intensity; bound by memory bandwidth

1. INTRODUCTION

With the ever increasing availability of cheap compute power and an incessant hunger of the scientific community to solve bigger and bigger problems, the importance of algorithms that scale well in terms of problem size grows as well. Multigrid has the appealing property that its cost scales only linear in the number of unknowns, i.e. multigrid applied to a set of linear equations is an $\mathcal{O}(N)$ method where N is the total number of unknowns. The building blocks of classical multigrid methods, the smoother, the interpolation and the restriction, are all stencil computations that have this $\mathcal{O}(N)$ complexity. Furthermore, the number of iterations is theoretically $\mathcal{O}(1)$.

In contrast to direct methods that are compute bound and typically scale as $\mathcal{O}(N^3)$, multigrid and iterative methods in general stress the memory bandwidth of the computer hardware. Iterative linear solvers are built on matrix-vector, vector-vector and scalar-vector operations, such as the SpMV (Sparse Matrix Vector multiplication), the dot product and the AXPY operation ($y = \alpha x + y$ with y and x vectors and α a scalar). These are $\mathcal{O}(N)$ operations which, except for the SpMV, can be found in the BLAS1 (Basic Linear Algebra Subroutines). They are all bounded by memory latency.

A good measure to quantify numerical methods is their associated arithmetic intensity, sometimes called operational or computational intensity and from hereon sometimes referred to as the q value.

*Correspondence to: Department of Mathematics and Computer Science, University of Antwerp, Middelheimlaan 1, B-2020 Antwerp, Belgium. E-mail: pieter.ghysels@ua.ac.be

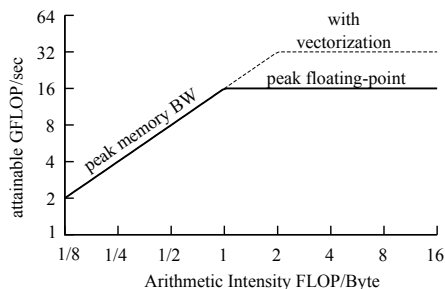


Figure 1. Roofline model for the maximum attainable floating point performance for numerical algorithms [1, 2] as function of the arithmetic intensity. Algorithms with a low arithmetic intensity (sparse matrix computations, stencil operations) are memory bound, while those with high arithmetic intensity are compute bound. Both regimes together form the roofline.

The arithmetic intensity of an algorithm is typically a fixed number and is defined as the number of floating point operations executed per byte fetched from main memory. Iterative solvers typically have a very low q value, whereas dense linear algebra methods have a much larger q .

The roofline model by Williams, Waterman and Patterson [1, 2] predicts the maximum attainable performance of a computer algorithm, measured in floating point operations per second (or in GFLOP/s), as a function of its arithmetic intensity. This simple model is illustrated in Figure 1. Applications with low arithmetic intensity are typically memory bound. The resulting code does not perform enough floating point operations per byte fetched from memory to hide all the memory latencies, and the program is often spending time waiting for data to return from main memory. Examples of algorithms or numerical kernels with very low arithmetic intensity are sparse matrix-vector products, dot products and AXPY operations. Stencil operations on structured grids have a slightly higher q value. Spectral methods such as a 3D FFT (Fast Fourier Transform) have an intermediate q value and dense matrix codes (as found in the BLAS3) have a high arithmetic intensity. Among the problems with the highest q values are N-body particle methods. For these higher q values, the algorithms are no longer memory bound but compute bound and the FPU (Floating Point Unit) can be kept busy almost all the time. The two regimes, memory bound and compute bound, form the roofline. The roofline is specific for each computer architecture and depends on the machine's memory system, clock frequency and many other factors. Writing code that achieves performance close to the theoretical roofline is very hard since this requires careful tuning and optimizing to take advantage of memory affinity, cache prefetching, instruction level parallelism and vector instructions.

Many authors have addressed the fact that certain numerical kernels, such as stencil operations, are memory bound and various approaches have been developed to improve efficiency of such codes. Just to name a few, we mention cache aware techniques based on the idea of tiling [3, 4, 5], possibly with auto-tuning of parameters [6], cache oblivious approaches [7, 8] and wavefront parallelization [9, 10]. Another approach that recently gained a lot of popularity is the use of GPGPU (General-Purpose computation on Graphics Processing Units) through CUDA[†] or OpenCL[‡], exploiting the GPU's large bandwidth and large number of compute cores. See for instance [11, 12, 13, 14, 15].

Douglas *et al* [16, 17, 18] and Kowarschik *et al* [19] have presented a number of multigrid code optimizations to exploit cache structure and reduce the number of cache misses. They combine the different multigrid steps into one and apply a blocking technique to this one big step. This approach, a complex form of loop fusion, enhances the temporal locality of the code and ensures better cache reuse. Impressive performance improvements have been reported using this techniques. Strout *et al* [20] apply a tiling technique at runtime for Gauss-Seidel smoothing on unstructured meshes.

[†]http://www.nvidia.com/object/cuda_home_new.html

[‡]<http://www.khronos.org/opencl>

Alternatively, to increase the arithmetic intensity of multigrid, one can try to switch to dense linear algebra techniques as much as possible, for instance by applying a block smoother with dense sub-blocks of the matrix. However, in this work, we will solely focus on stencil computations.

However, most of the optimizations mentioned are merely clever code reorganizations that do not affect the total number of floating point operations or the numerical properties, like convergence rate or stability, of the algorithms. An important trend in recent hardware is the increasing gap between performance of the FPU and performance of the memory system. CPU performance has been increasing for years now according to Moore's law, while memory latencies have only been improving by 10%/year. In other words: floating point operations are cheap, communication is expensive. Moving data from one part of the chip can cost more than simply recomputing the result locally. This is also true in terms of energy use; improving data locality will become critical for energy efficiency [21]. This allows much more freedom in designing parallel algorithms with reduced communication and synchronization.

Demmel *et al* [22] for instance introduce the matrix-powers kernel which computes s subsequent SpMV's without synchronization after each step by dividing the vector in tiles and performing redundant work at the tile boundaries. In this case, arithmetic intensity can be increased by trading cache misses for redundant computations. Also, the matrix has to be fetched from memory only once per s steps. However, the matrix-powers kernel is not a basic building block of any classical iterative method. In order to exploit the benefits of this new numerical kernel, s -step variants of CG (Conjugate Gradients) and GMRES (Generalized Minimal RESiduals) have been developed [23, 24]. These methods are not new, but have recently regained some attention. They have the additional benefit of reducing global communication. However, they often incur too much extra computations to be beneficial on current hardware and are hard to combine with many of the more efficient preconditioners.

In this paper, we study the classical multigrid algorithm applied to the 2D Poisson problem on a regular grid. For the smoother we use Chebyshev iteration and we propose a tiled implementation, similar to the matrix-powers kernel from Demmel *et al* [22] to improve temporal locality of the polynomial Chebyshev smoother. This improves data cache reuse and hence minimizes communication to off-chip memory as well as communication and synchronization between threads. By introducing two extra parameters, a tile size B (dependent on the machine's cache size) and the number of smoothing steps s , we create an algorithm with a variable arithmetic intensity. By varying the q value, the smoother becomes compute bound rather than memory bound and overall performance increases.

This paper is organized as follows. In section 2 we briefly introduce the model problem, the multigrid scheme and polynomial Chebyshev smoothing. Section 3 describes how the arithmetic intensity of the Chebyshev iteration can be increased by tiling. We give experimental results for the attained speedups for standard Chebyshev iteration, as well as for a multigrid V -cycle using Chebyshev smoothing, both with tiles. In the experiments described in section 4, the lower bound for the eigenvalue estimate, as needed by the Chebyshev iteration, is varied. This allows one level of the V -cycle to smooth more than the upper half of the frequencies in the error. In section 5 a more aggressive coarsening strategy is used. Since also the mid-frequency error terms are smoothed by the higher degree Chebyshev smoother, the error can be accurately represented on a $4h$ grid instead of a $2h$ grid. Finally, section 6 concludes the paper and gives a short outlook.

2. MULTIGRID FOR 2D POISSON WITH POLYNOMIAL SMOOTHERS

In this paper we look at a 2D Poisson problem, which has extensively been studied in the literature [25, 26, 27]. The Poisson equation, with Dirichlet boundary conditions, is discretized using a 5-point stencil on a structured grid with grid spacing $h = 1/(n + 1)$, where n is the number of unknowns per spatial direction, over the spatial domain $[0, 1]^2$. The system to be solved can be written in the matrix vector form

$$Ax = b, \tag{1}$$

with A the 2D Poisson matrix (using standard lexicographic ordering of the grid points), b the right-hand-side vector and x the vector of unknowns. This set of linear equations will be solved with multigrid starting from an initial guess $x_0 = [1, 1, \dots, 1]^T$, which, on a 1023×1023 grid, corresponds to an initial residual $\|b - Ax_0\|_2 = 6.7 \times 10^7$.

We assume the reader is familiar with multigrid and refer to the books [26, 27] for details. A standard $V(\nu_1, \nu_2)$ -cycle will be used with a hierarchy of mesh sizes $h, 2h, \dots, 1/2$, with linear interpolation and full weighting restriction.

In the $V(\nu_1, \nu_2)$ -cycle, ν_1 iterations of the smoother are applied before solving the error equation on the coarser grid, and ν_2 iterations are applied after correction with the interpolated error. A well known smoother is the weighted-Jacobi iteration

$$x^{(k+1)} = [I - \omega D^{-1}A] x^{(k)} + \omega D^{-1}b, \quad (2)$$

where I is the identity matrix and D is the diagonal of A . For our model problem, the optimal weighting parameter ω can be determined to be $2/3$ [26].

Besides Jacobi iteration, we also consider a polynomial Chebyshev smoother. Chebyshev iteration [25, 28, 29] is an iterative method for the solution of a system of linear equations and, unlike Jacobi, it is not a stationary method. However, it does not require inner products like many other nonstationary methods (most Krylov methods). These inner products can be a performance bottleneck on certain distributed memory architectures. Furthermore, Chebyshev iteration is, like Jacobi, easier to parallelize than for instance Gauss-Seidel smoothers, see [30] for a quantitative comparison. The Chebyshev iteration requires some information about the spectrum of the matrix A . For symmetric matrices it needs an upper bound for the largest eigenvalue and a lower bound for the smallest eigenvalue. For the extension to nonsymmetric matrices, an ellipse enveloping the spectrum should be identified. Manteuffel [31, 32] and Ashby [33] have developed and implemented an adaptive approach that estimates these eigenvalue bounds during the Chebyshev iteration, based on a modified power method. Elman *et al* [34] have developed a more sophisticated approach based on an Arnoldi iteration. However, for our model problem, the extreme eigenvalues are known analytically to be $\lambda_{\min} = 8/h^2 \sin^2(\pi/(2(n+1))) \approx 2\pi^2$ and $\lambda_{\max} = 8/h^2 \cos^2(\pi/(2(n+1))) \approx 8/h^2$.

Algorithm 1 Chebyshev smoothing

```

for  $k = 0$  to  $s - 1$  do
   $r \leftarrow b - Ax$ 
   $\alpha \leftarrow \begin{cases} d^{-1}, & \text{if } k = 0 \\ 2d(2d^2 - c^2)^{-1}, & \text{if } k = 1 \\ (d - \alpha c^2/4)^{-1}, & \text{if } k > 1 \end{cases}$ 
   $\beta \leftarrow \alpha d - 1$ 
   $p \leftarrow \alpha r + \beta p$ 
   $x \leftarrow x + p$ 
end for

```

The Chebyshev smoother is shown as Algorithm 1. The scalar input parameter $d > 0$ is the center of the ellipse containing the eigenvalues of A . The foci of this ellipse are located at $d + c$ and $d - c$. For purely real eigenvalues, parameters d and c are related to the smallest and largest eigenvalues λ_{\min} and λ_{\max} respectively through

$$d = \frac{\lambda_{\max} + \lambda_{\min}}{2} \quad \text{and} \quad c = \frac{\lambda_{\max} - \lambda_{\min}}{2}. \quad (3)$$

Traditionally, multigrid implementations use only a few smoothing steps, ranging between zero and two, since the additional cost of one extra smoother step does not outweigh the better convergence. This is illustrated in Figure 2, where on the left the approximate asymptotic convergence rate for a $V(\nu_1, \nu_2)$ cycle is shown as a function of ν_1 for $\nu_2 = 0, 1, 2$ for both weighted-Jacobi and Chebyshev smoothing. The asymptotic convergence rate is approximated by

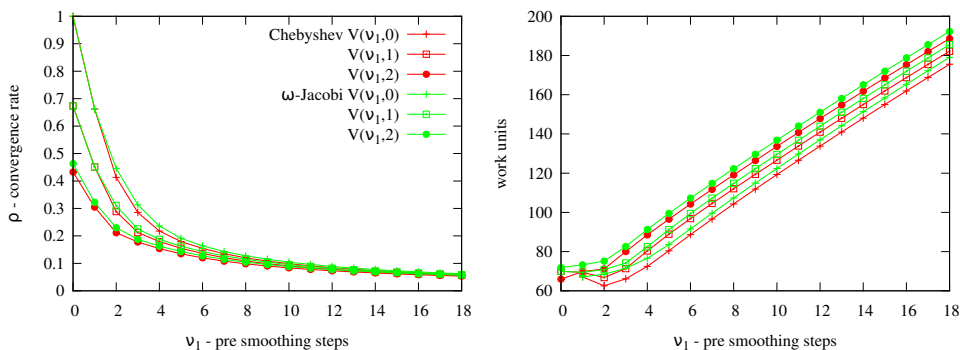


Figure 2. Left: Approximate asymptotic convergence rate of a V -cycle using a weighted Jacobi smoother with $\omega = 2/3$ and a Chebyshev smoother as a function of the number of pre-smoothing steps for 0, 1 and 2 post-smoothing steps. For moderate numbers of pre-smoothing steps, the extra gain in convergence rate for extra steps becomes marginal. Right: The work units for a V -cycle to reach a tolerance $\text{tol} = 10^{-9}$ increase approximately linearly as a function of the number of pre-smoothing steps.

the convergence rate of the last V -cycle in a multigrid solve with tolerance $\|Ax - b\|_2 < 10^{-9}$. For larger values of ν_1 , the improvement in convergence rate for increasing ν_1 becomes marginal. The Chebyshev smoother uses $\lambda_{\min} = 4/h^2$ and $\lambda_{\max} = 8/h^2$, and hence damps only the oscillatory modes. Figure 2 (right) shows the number of work units required to reach $\|Ax - b\|_2 < 10^{-9}$ with the same V -cycles as used in Figure 2 (left). Alternatively, a relative stopping criterion could be used. The required number of V -cycles is computed as $\log 10^{-9} / \log \rho$, where ρ is the asymptotic convergence rate of the V -cycle. We define work units as the number of matrix-vector products (smoothing steps) at the finest level, neglecting the cost of intergrid transfer and residual computation as in [26]. The total cost for the V -cycle to reach a tolerance 10^{-9} , expressed in terms of work units is thus

$$(\nu_1 + \nu_2) (1 + 4^{-1} + 4^{-2} + \dots + 4^{-l}) \frac{\log 10^{-9}}{\log \rho} \text{WU} \approx (\nu_1 + \nu_2) \frac{4 \log 10^{-9}}{3 \log \rho} \text{WU}, \quad (4)$$

with $l = \log_2(n + 1)$ the number of levels in the multigrid hierarchy. Due to the extra cost for each additional smoother step, and the slow improvement in convergence rate, the cost for a $V(\nu_1, \nu_2)$ cycle has a minimum at around $\nu_1 = 2$ and increases approximately linearly for larger ν_1 . This confirms the common knowledge that traditional multigrid with few smoothing steps works efficiently.

Another very popular smoother is red-black Gauss-Seidel, which has a slightly better convergence rate and requires less number of work units compared to ω -Jacobi and Chebyshev smoothers. However, also for red-black Gauss-Seidel, the work units start to increase approximately linearly for larger ν_1 .

3. TILING TO IMPROVE TEMPORAL LOCALITY

In order to improve the arithmetic intensity of the smoother, the grid is divided in square tiles of size $B \times B$ at each level. For both the Chebyshev and the Jacobi iterations, each tile holds its corresponding part of the field of unknowns x , the right-hand-side b and an additional search direction vector p . The residual is not stored explicitly. When the tile is created, its data (the corresponding parts from x and b) are read from main memory and loaded in the core's data cache. To improve cache data reuse, several smoothing steps are performed per tile before writing the smoothed x back to the global x . However, the matrix-vector product, which is based on a 5-point stencil for the Poisson equation, causes a data dependency at the edges of the tiles. In order to resolve this data dependency, the tile is extended with a buffer when it is created. The width of this

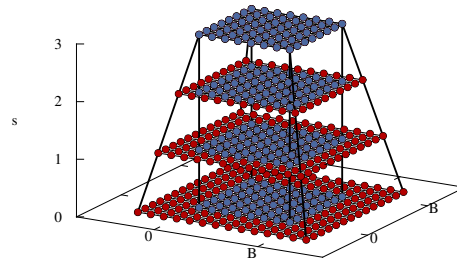


Figure 3. Illustration of 3 iterations, e.g. Chebyshev or Jacobi smoothing steps, applied to an 8 by 8 tile. To satisfy the data dependencies up to the third smoother step, redundant computations (on the red dots) are performed at the boundaries of the tile, giving the computational domain the shape of a square frustum.

buffer is equal to the number of smoothing steps. In Figure 3 this is illustrated for 3 smoothing steps applied to an 8×8 tile. The red dots lying in the buffer are needed for the data dependency after 3 smoothing steps. These red dots are computed redundantly by the neighboring tile(s). The computational domain inside a tile has the form of a square frustum (a pyramid with its top cut off).

The tile size parameter B should be chosen with care. Smaller tiles incur a larger overhead due to tile creation and require more redundant work. On the other hand, tiles should be small enough to fit in the machine's cache and to allow for an even distribution of the load over different processing cores. Each tile is evolved by a separate thread, which is spawned by the Intel® Threading Building Blocks (TBB) library. In TBB, a work stealing scheduler distributes the tiles over the available cores. This allows for an easy parallelization of the smoother, with only a single synchronization point for s steps of the iterative procedure.

When the Chebyshev iteration is used as a standalone solution procedure, its arithmetic intensity can be increased by applying the iteration in tiles and varying the two parameters B and s . To determine the arithmetic intensity for a given combination of B and s we proceed as follows. Let V_{cd} denote the volume of the computational domain of a tile and let γ be the number of floating point operations required in each grid point. The volume of the frustum is given by

$$V_{\text{cd}} = \frac{1}{6} \left((B + 2s)^3 - B^3 \right). \quad (5)$$

The average total number of floating point operations per iteration for the entire domain can be computed as

$$\#\text{flop}_{\text{Cheb}}^{\text{total}} = \gamma V_{\text{cd}} \frac{\#\text{tiles}}{s}, \quad (6)$$

where $\#\text{tiles}$ is the number of tiles in the domain. The average number of useful floating point operations per iteration, i.e. excluding the redundant computations, is given by

$$\#\text{flop}_{\text{Cheb}}^{\text{useful}} = \gamma V_{\text{useful}} \frac{\#\text{tiles}}{s}, \quad (7)$$

where $V_{\text{useful}} = B^2 s$ is the volume of the beam enclosed by the tile. For the Chebyshev iteration, $\gamma = 11$ (6 for the matrix-vector product, 3 vector-vector additions and 2 scalar-vector multiplications). Furthermore, in the standalone Chebyshev iteration, where several runs with s steps are performed consecutively, the vectors x , b and p have to be fetched from main memory and have to be written

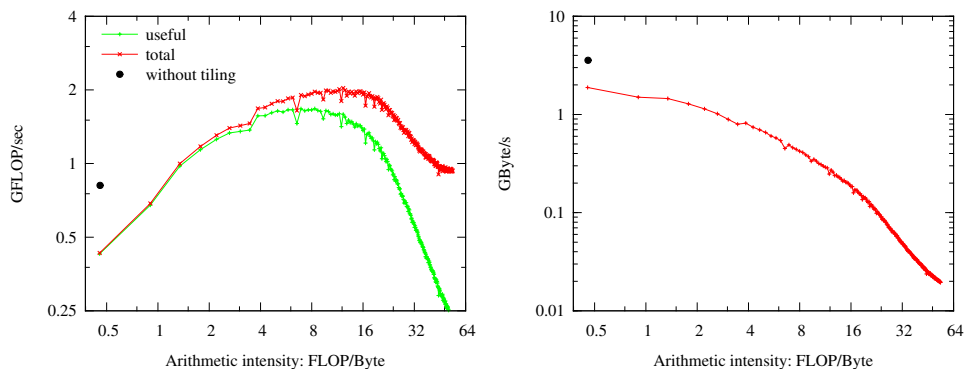


Figure 4. Chebyshev iteration applied to the 2D model problem using 256^2 tiles, with $s = 1 \dots 256$. The black dot represents Chebyshev iteration implemented without tiles. Left: The attained performance is expressed in GFLOP/s as a function of the arithmetic intensity (q_{Cheb}). The green curve shows the useful attained performance while for the red curve the redundant computations are included as well. From these data, the roofline as illustrated in Figure 1 becomes apparent. Right: The bandwidth (GByte/s) is plotted against the arithmetic intensity.

back to main memory[§]. Discarding the memory writes, the number of bytes fetched from main memory per iteration is

$$\# \text{byte}_{\text{Cheb}} = 8 \cdot 3 \cdot (B + 2s)^2 \# \text{tiles}, \quad (8)$$

when 8-byte double-precision floating point numbers are used. The arithmetic intensity of a tiled Chebyshev iteration can thus be computed as

$$q_{\text{Cheb}} = \frac{\# \text{flop}_{\text{Cheb}}^{\text{total}}}{\# \text{byte}_{\text{Cheb}}}. \quad (9)$$

In Figure 4, Chebyshev iteration with $\lambda_{\min} = 4/h^2$ was applied to a Poisson problem ($n = 2^{10}$) using a tile size $B = 256$, while varying s . The total number of iterations for each run was $5s$, and each run was repeated 3 times. From the minimum execution time over these 3 runs the average execution time per iteration was computed. For each s , the attained performance, expressed as $\# \text{flop}_{\text{Cheb}}^{\text{total}}/\text{second}$ and as $\# \text{flop}_{\text{Cheb}}^{\text{useful}}/\text{second}$ are plotted against the corresponding q_{Cheb} value in Figure 4 (left). In this figure, the roofline becomes apparent. For small values of s , the arithmetic intensity is low: there is little reuse of the data in the cache and the algorithm is bound by memory traffic. As s increases, the arithmetic intensity increases together with the overall performance. For modest values of s , the performance starts to level off as the algorithm becomes compute bound rather than memory bound. For increasingly larger s the redundant work starts to dominate, the useful performance drops quickly and the gap between attained useful flops and total flops increases. The additional drop in performance for the total flops is caused by cache effects. At some point, the basis of the frustum, which increases in size with increasing s , will no longer fit in the cache. We should stress that at this point ($q_{\text{Cheb}} > 16$), the arithmetic intensity of the algorithm is overestimated by our simple model which assumes that only the ground surface of the frustum has to be fetched from main memory and that all consecutive iterations on the tile are performed completely in the cache. However, this regime is beyond the region of interest for practical computations, since the impact of the redundant computations is too big. The black dot in Figure 4 denotes a naive Chebyshev iteration without tiling. Standard Chebyshev iteration does not include any tiling parameters and hence has a fixed arithmetic intensity ($q_{\text{Cheb}} = 11/(8 \cdot 3) = 0.45$). The overhead of copying data from and to the tiles causes a noticeable penalty in performance between the standard implementation and the tiled version with low q values.

[§]When only s steps are needed, for instance when applied as a smoother, the p vector does not need to be stored in main memory explicitly, since it is only needed inside the tile.

Figure 4 (right) shows, for the same benchmark, the attained bandwidth to off-chip memory, again as a function of arithmetic intensity. The attained bandwidth is estimated as $(\# \text{byte}_{\text{Cheb}} + 24 B^2 \# \text{tiles})/\text{second}$, i.e. the number of bytes fetched from and written to main memory per iteration, divided by the average runtime for a single iteration. As to be expected, the bandwidth usage declines as the arithmetic intensity increases, since data loaded into the processors cache can be used more efficiently. This can possibly solve scaling problems for stencil computations on multi/many core shared memory machines, where several cores have to share the available memory bandwidth. Note that here again, the arithmetic intensity and hence the bandwidth are no longer correctly predicted by the model for $q_{\text{Cheb}} > 16$.

This benchmark was performed on a single core from an Intel®Core™i7 CPU M640 @2.80GHz. Compared to the non-tiled implementation, a maximum speedup of $2.2\times$ is realized for $B = 256$ and $s = 20$. Our Intel®Core™i7 test system has two memory banks, each with a bandwidth of 8.5 GB/s [¶]. Since the entire domain fits in one of these banks, 8.5 GB/s is also the maximum achievable bandwidth. The measured maximum achieved bandwidth, 3.6 GB/s, is about 57% of the bandwidth as measured by the stream benchmark ^{||}, which was 6.3 GB/s.

A naive implementation of a red-black Gauss-Seidel smoother would require two sweeps over the computational domain, one for the red points, one for the black points. Hence a greater benefit might be expected from tiling the red-black Gauss-Seidel smoother. However, the red-black update sequence leads to a larger data dependency, i.e., a wider basis for the frustum, which leads to more redundant computations. Hence, we expect a similar pay-off for red-black Gauss-Seidel smoothing.

3.1. The V cycle with tiled smoother

For Chebyshev iteration as such, an optimal combination of the parameters B and s can be determined by looking at the maxima of the rooflines for different B values, cf. Figure 4 (left). However, in section 2 (Figure 2 (right)), it was shown that the cost of a multigrid V -cycle is at its lowest for few smoothing steps. This is due to the fact that for a standard, non-tiled, implementation of the smoother the convergence rate of the V -cycle does not significantly improve with increasing number of smoothing steps, while the computational cost increases with each additional application of the smoother.

Figure 5 shows the measured wall clock time to solve a 2D Poisson problem ($n = 2^{10}$, $\|r\|_2 < 10^{-9}$) with a multigrid $V(\nu_1, \nu_2)$ -cycle as a function of the number of pre-smoothing steps ν_1 . Data for both Chebyshev and Jacobi smoothing are shown. The weighted Jacobi smoother with $\omega = 2/3$ effectively damps the upper half of the error frequencies. Similarly, for the Chebyshev smoother, $\lambda_{\min} = 4/h^2$ and $\lambda_{\max} = 8/h^2$ minimizes the amplification factor of the Chebyshev smoother over the upper half of the error spectrum. Figures 5 (top), (left) and (right) show timings for respectively a $V(\nu_1, 0)$, $V(\nu_1, 1)$ and a $V(\nu_1, 2)$ -cycle. The pre-smoother was tested without tiles and with tiles of sizes $B = 128$ and 256. For a low number of post-smoothing steps, it does not pay off to use tiling, therefore post-smoothing in these experiments is done without tiling. However, the tiling can easily be turned on for parallel execution.

For the $V(\nu_1, 0)$ cycle, the optimum with Chebyshev smoothing shifts from $\nu_1 = 3$ and 1.96 s for the non-tiled version to $\nu_1 = 6$ and 1.21 s, an improvement of 38 %. For Jacobi smoothing, the optimum shifts from $\nu_1 = 5$ and 1.53 s to $\nu_1 = 5$ and 1.21 s, a 21 % improvement. Note that also for the non-tiled code, the optimal number of smoothing steps does not correspond to that predicted by the work units. The Jacobi the $V(5, 0)$ -cycle needs approximately 25% more work units than the $V(1, 0)$ -cycle, while in the non-tiled implementation it is faster by 30%. We speculate that this is due to cache effects on the coarser levels and overhead of starting the smoother. For the $V(\nu_1, 1)$ and $V(\nu_1, 2)$ -cycles the post-smoother, which is not tiled, takes a larger part of the total execution time and the gain is less apparent. The sudden drops in execution time with for instance the Chebyshev smoother and the $V(\nu_1, 0)$ -cycle when going from 12 to 13 and from 16 to 17 pre-smoothing steps are due to the fact that the number of V -cycles required to reach the 10^{-9} tolerance changes from

[¶]<http://ark.intel.com/products/49666>

^{||}<http://www.cs.virginia.edu/stream/>

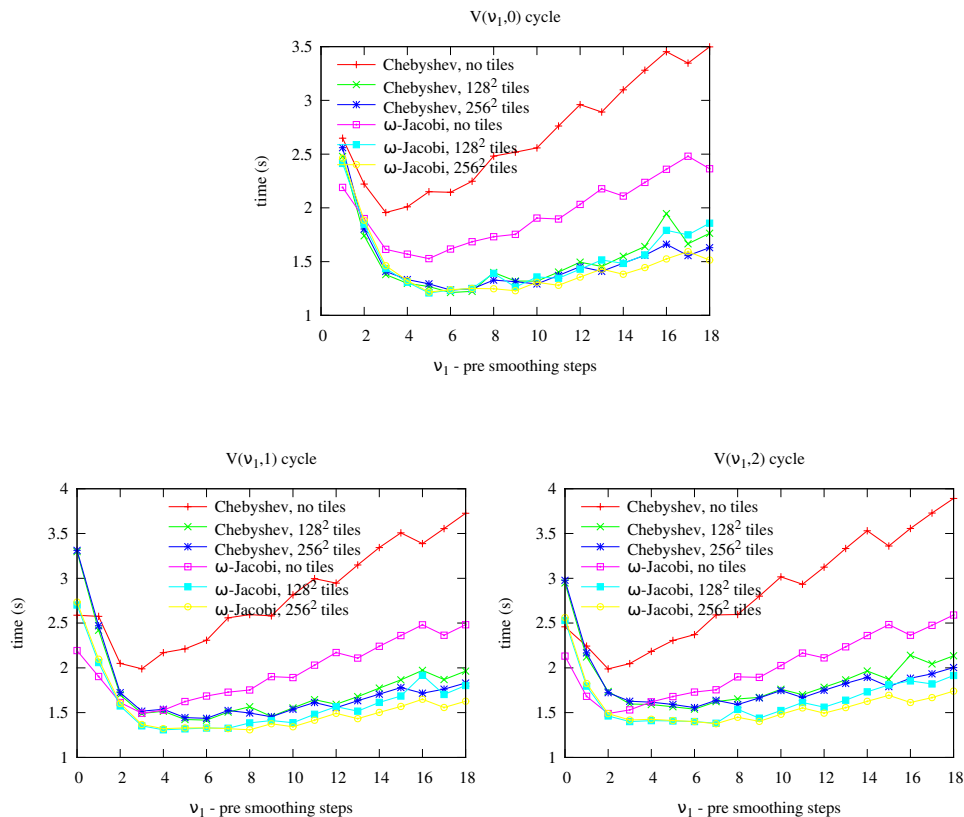


Figure 5. Timings to solve a 2D Poisson problem ($n = 2^{10}$) using a multigrid $V(\nu_1, \nu_2)$ -cycle with a 10^{-9} tolerance as function of the number of pre-smoothing steps ν_1 , for both Chebyshev and Jacobi smoothing. Top: a $V(\nu_1, 0)$ -cycle. Left: $V(\nu_1, 1)$ -cycle. Right: $V(\nu_1, 2)$ -cycle. Implementation with tiled pre-smoothing is compared to standard implementation without tiles. For the post-smoothing, no tiling is used since for these low number of smoothing steps tiling does not pay off. For the $V(\nu_1, 0)$ -cycle with Chebyshev smoother, applying tiling shifts the optimum from $\nu_1 = 3$ and 1.96 s to $\nu_1 = 6$ and 1.21 s, an improvement of 38 % and with Jacobi smoothing from $\nu_1 = 5$ and 1.53 s to $\nu_1 = 5$ and 1.21 s, a 21 % improvement.

11 to 10 and from 10 to 9 respectively. One might wonder why in the non-tiled implementation the Jacobi smoother is clearly faster than the Chebyshev smoother, while with tiling both perform equally. This can be explained as follows. Compared to Jacobi iteration, Chebyshev iteration creates and loops over an extra vector (the search direction p), which causes extra memory traffic. In the tiled implementation, no off-chip memory is allocated for p since it is only used in the tiles, making the tiled Chebyshev iteration comparable to the tiled Jacobi iteration in terms of memory traffic. Note that also the computation of the residual vector and the intergrid transfer take up a considerable part of the total execution time, which is not affected by the tiling in the current implementation.

In Figure 6, both the pre and post-smoother are implemented with tiling. The execution time for $V(\nu_1, \nu_2)$ -multigrid is now shown as a function of both ν_1 and ν_2 . The symmetry along the diagonal in these figures can intuitively be explained as follows. In a two-grid correction scheme, the post-smoothing of one correction cycle is followed immediately by the pre-smoothing of the next cycle. The convergence rate is determined by the sum of the number of pre and post-smoothing steps, which is constant along the diagonal. The optimal choices for ν_1 and ν_2 and the corresponding timings are given in Table I.

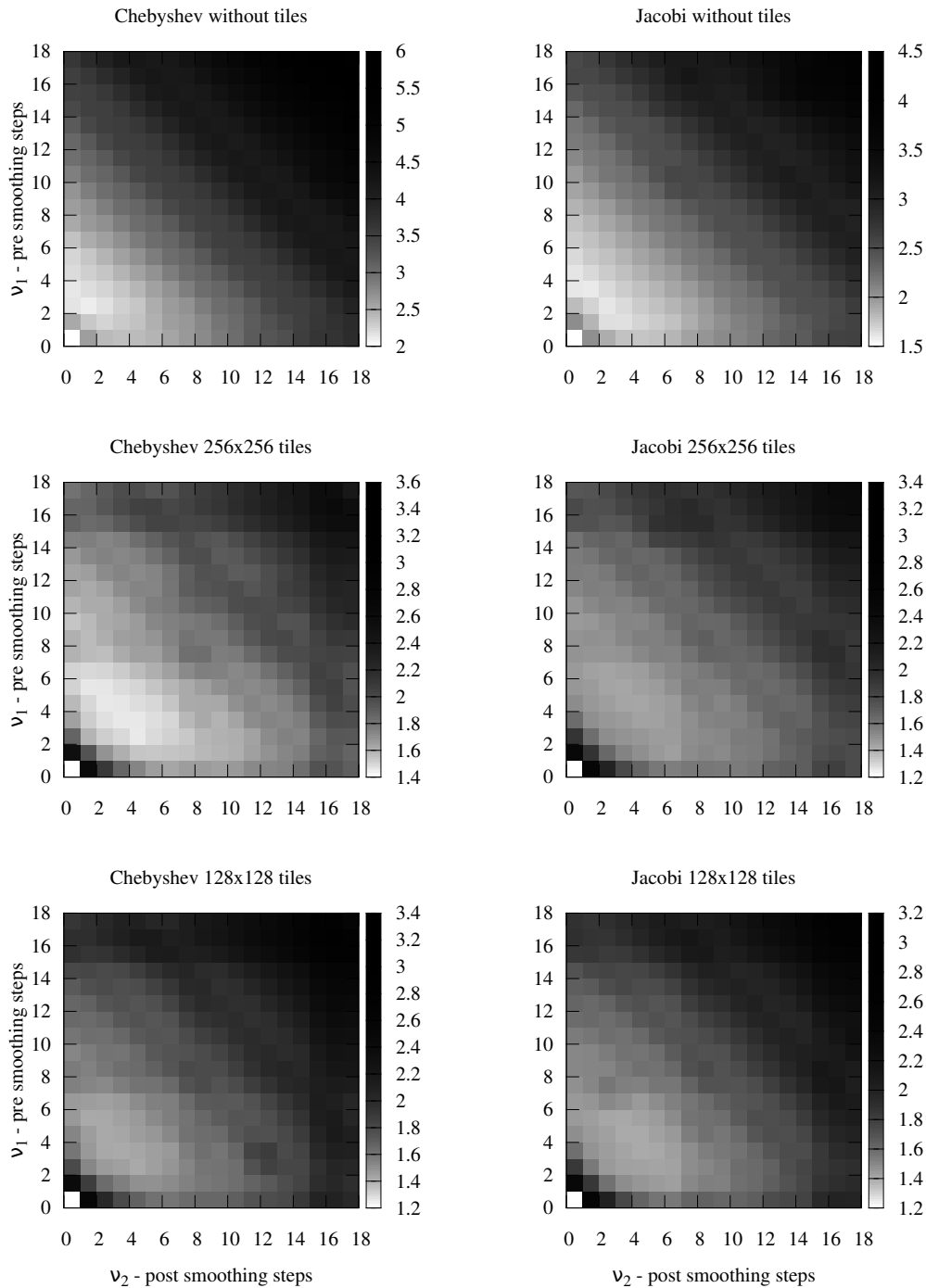


Figure 6. Timings for a similar experiment as shown in Figure 5, but now with tiling applied to both pre and post-smoothing. Table I lists the optimal choices for ν_1 and ν_2 and the corresponding timings. From these figures, it is clear that the minima are shifted towards higher degrees of the smoother when tiles are used. Note the symmetry along the diagonal.

smoother	B	(ν_1, ν_2)	time (s)
	-	(3, 0)	2.045
Chebyshev	128	(5, 3)	1.391
	256	(3, 5)	1.425
	-	(3, 1)	1.534
Jacobi	128	(5, 4)	1.367
	256	(6, 3)	1.397

Table I. For Jacobi and Chebyshev smoothing, this table shows the optimal number of smoothing steps and timings for different tile size B , as determined from Figure 6.

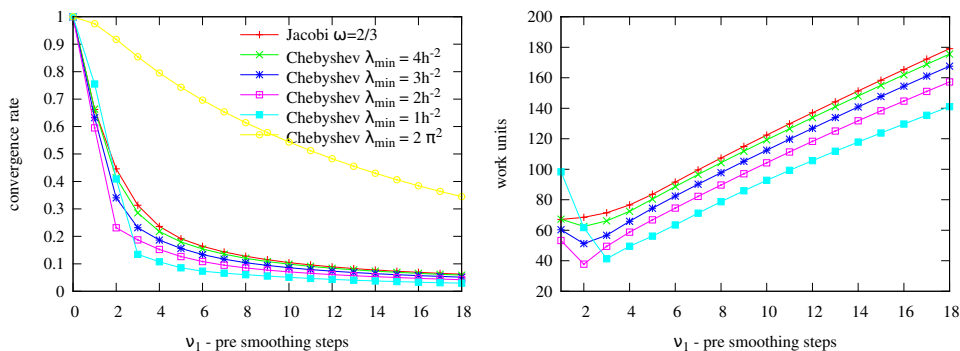


Figure 7. Left: Convergence rates for a $V(\nu_1, 0)$ -cycle with Chebyshev and Jacobi smoothers, as function of the number of pre-smoothing steps ν_1 . The lower bound λ_{\min} for the eigenvalues, as used in the Chebyshev smoother, is varied. Right: Estimate for the work units required to reach a 10^{-9} tolerance.

4. SMOOTHING THE MID-FREQUENCY ERROR TERMS

Both the ω -Jacobi and the Chebyshev iteration have tunable parameters. For Jacobi this is the weight ω and for Chebyshev these are c and d , which are related to λ_{\min} and λ_{\max} through relation (3). Traditional choices are $\omega = 2/3$ for Jacobi and $\lambda_{\min} = 4/h^2$ and $\lambda_{\max} = 8/h^2$ for Chebyshev. This choice makes sure that half of the spectrum is damped efficiently.

Changing ω will adversely affect the convergence rate. However, changing λ_{\min} while keeping λ_{\max} fixed will improve the multigrid convergence rate for higher order polynomials. By changing the lower bound, the smoother can efficiently damp more modes than just those in the upper half of the frequency range.

In Figure 7 (left) we show the asymptotic convergence rate of multigrid as a function of the order of the Chebyshev polynomial for $\lambda_{\min} = 2\pi^2, 1/h^2, 2/h^2, 3/h^2$ and $4/h^2$. From a Chebyshev order of three on, the convergence rate improves as λ_{\min} is lowered. The same result is reflected in the number of work units required to converge to a given tolerance, shown in Figure 7 (right). However, decreasing λ_{\min} beyond a certain point deteriorates the convergence rate, as can be seen from Figure 7 (left) for $\lambda_{\min} = 2\pi^2$.

This improved convergence rate for lower λ_{\min} is also observable in the timings for the multigrid solver. In Figure 8 we show the timings for increasing order of the polynomial. For the results without tiles (left) we already see better timings with a third order polynomial with $\lambda_{\min} = 1/h^2$.

When tiling is included, again the best results are obtained with $\lambda_{\min} = 1/h^2$. However, the optimal choice is not necessarily a low order polynomial. Also higher order polynomials give good performance. For example, with 256×256 tiles we get best performance for $\nu_1 = 6$. This indicates that the increased computational cost of taking higher order polynomials, as predicted by Figure 7 (right), is compensated by the better arithmetic intensity of the higher order version. The problem is solved in approximately the same time for $\nu_1 = 3 \dots 6$ with the optimum at $\nu_1 = 6$ with

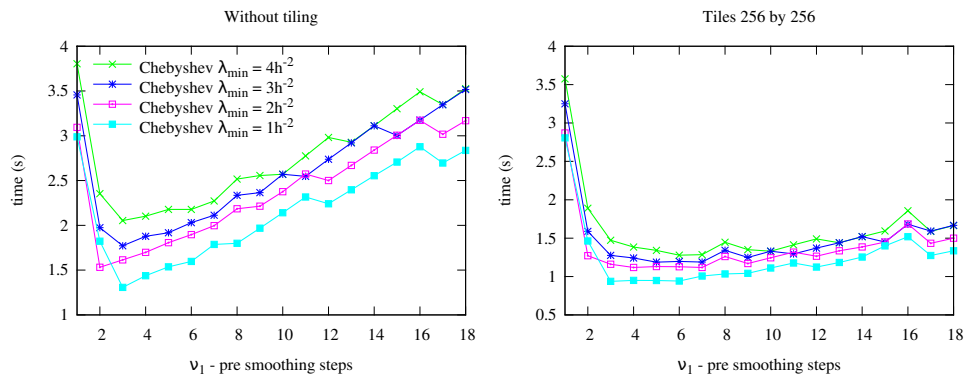


Figure 8. Timings for a multigrid $V(\nu_1, 0)$ -cycle to solve the 2D Poisson model problem ($n = 2^{10}$) with a 10^{-9} tolerance. The lower bound λ_{\min} for the eigenvalues, as used in the Chebyshev smoother, is varied. Left: Without tiling, the cost increases approximately linearly with the number of pre-smoothing steps ν_1 . Right: The smoother is implemented with tiles of size $B = 256$.

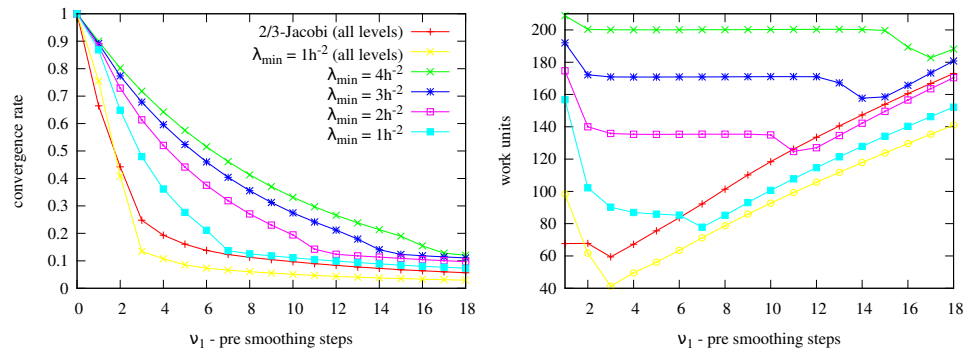


Figure 9. Left: Convergence rates for a $V(\nu_1, 0)$ -cycle with Chebyshev smoothing, as function of the number of pre-smoothing steps ν_1 and for different values of λ_{\min} . The V -cycle interpolates (cubic) from $4h$ to h and coarsens from h to $4h$. For comparison, the convergence rates for ω -Jacobi and Chebyshev ($\lambda_{\min} = 1/h^2$) with normal coarsening (denoted *all levels*) are shown. Right: Estimate for the work units required to reach a 10^{-9} tolerance. Certain levels are skipped, hence they do not contribute to the work units.

0.937 seconds. This is a 27% improvement ($1.4 \times$ speedup) compared to Chebyshev smoothing without tiles and with $\lambda_{\min} = 4/h^2$ (from 1.278 to 0.937 seconds). Similarly, there is a speedup compared to the best weighted-Jacobi smoothing without tiles ($V(3, 1)$) of $1.6 \times$ (from 1.534 to 0.937 seconds). Compared to the non-tiled Jacobi $V(1, 0)$ -cycle, which is optimal in the number of work units, we achieve a speedup of $2.3 \times$.

5. MORE AGGRESSIVE COARSENING

By taking $\lambda_{\min} < 4/h^2$ and $\lambda_{\max} = 8/h^2$, more than just the upper half of the error spectrum is damped. When λ_{\min} is sufficiently small and enough smoothing steps are applied, the error, on a grid with spacing h , will be so smooth that it can be represented accurately on a grid with $4h$. The optimum from the previous section ($\nu_1 = 6$ and $\lambda_{\min} = 1/h^2$) motivates exploration of such a more aggressive coarsening strategy.

Figure 9 (left), shows the convergence rate of a multigrid $V(\nu_1, 0)$ -cycle as a function of ν_1 for different values of λ_{\min} . For these experiments, the coarsening was done from h to $4h$ and

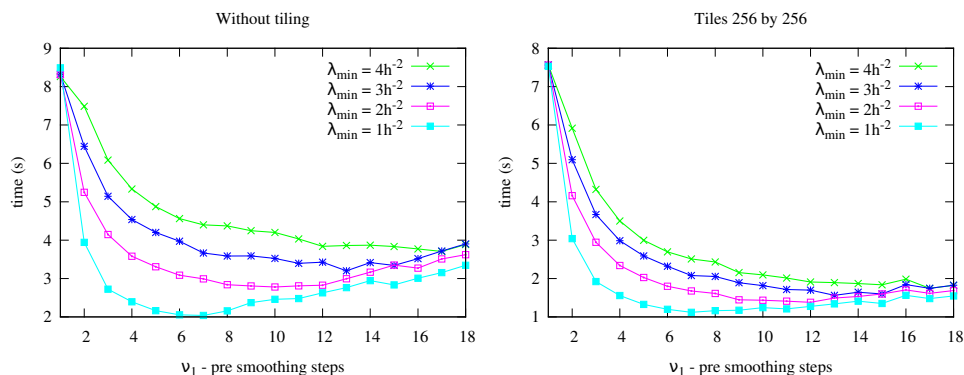


Figure 10. Timings for a multigrid $V(\nu_1, 0)$ -cycle to solve the 2D Poisson model problem ($n = 2^{10}$) with a 10^{-9} tolerance. The coarsening is done from h to $4h$, λ_{\min} is varied. Left: Without tiling. For $\lambda_{\min} = 4/h^2$, the intermediate frequency error terms are only damped slowly and hence more smoother steps lead to faster convergence of the V -cycle. Right: The smoother is implemented with tiles of size $B = 256$.

likewise, interpolation from $4h$ to h . These convergence rates are compared to those for 2/3-Jacobi and Chebyshev ($\lambda_{\min} = 4/h^2$), both with regular coarsening/interpolation between h and $2h$. All tests reported in this section use cubic interpolation. Interpolation from $4h$ to h is implemented as two consecutive interpolations, one from $4h$ to $2h$ followed by one from $2h$ to h .

Figure 9 (right) shows the corresponding number of work units required to solve the model problem with a 10^{-9} tolerance. However, since now the V -cycle uses a reduced hierarchy of levels, the cost in terms of work units is given as

$$(\nu_1 + \nu_2) \left(1 + 4^{-2} + 4^{-4} + \dots + 4^{-l}\right) \frac{\log 10^{-9}}{\log \rho} \text{WU} \approx (\nu_1 + \nu_2) \frac{16 \log 10^{-9}}{15 \log \rho} \text{WU}. \quad (10)$$

where the constant changed from $4/3$ (Eq. 4) to $16/15$. Skipping a level can also reduce the memory footprint by 25%. Figure 9 (right) also compares with 2/3-Jacobi and Chebyshev ($\lambda_{\min} = 4/h^2$), both with regular coarsening/interpolation between h and $2h$.

We observe that the cost (WU's) remains approximately constant for several iterations until the error is sufficiently smooth such that it can accurately be represented on a much coarser grid. From then, additional smoothing has no real benefit and the cost starts to increase linearly again. For enough smoothing steps and with $\lambda_{\min} = 1/h^2$, the V -cycle with the more aggressive coarsening is predicted to be faster than a normal V -cycle with standard 2/3-Jacobi smoother.

Figure 10 shows timings for the V -cycle with aggressive coarsening, left without tiling, right using $B = 256$ tiles. When $\lambda_{\min} = 4/h^2$, the frequencies that can be represented on the intermediate $2h$ grid but not on the coarser $4h$ grid only decline slowly. Hence, more smoothing steps are required than for the $\lambda_{\min} = 1/h^2$ case. From Figure 10 (right), we observe an optimum of 1.118 seconds ($\nu_1 = 7$), which is a 27% improvement compared to the best normal V -cycle (all levels) using 2/3-Jacobi smoothing (1.534 seconds) and a $1.96 \times$ improvement compared to the $V(1, 0)$, which is optimal in the number of work units. For the non-tiled case, we also observe a performance improvement for increasing ν_1 (for small ν_1), which is not predicted by the work units but might be due to cache effects on the coarser levels.

6. CONCLUSIONS

It is an important trend in computer hardware that the increase in memory bandwidth of a CPU will lag behind the increase of available flops. This will lead to a loss of relative performance of

multigrid solvers that are build on stencil computations whose performance is typically bound by the available memory bandwidth.

This work explores the possibilities to increase the arithmetic intensity of a multigrid solver by trading memory traffic for redundant computations. The focus is on replacing the standard ω -Jacobi smoother with s -step smoothers. In general, additional smoothing steps at each level of the multigrid hierarchy do not improve performance enough to justify the cost of additional matrix-vector products. However, and counter intuitively, when the smoother is applied repeatedly the loops can be reorganized such that data in fast memory is reused, and overall performance increases.

Classical smoothers like ω -Jacobi and Gauss-Seidel are stencil computations with a fixed arithmetic intensity. Polynomial smoothers, however, can be implemented with a variable arithmetic intensity. Indeed, with each order of the Chebyshev polynomial s and tile size B we can associate a q -value. Varying this q allows one to probe the roofline model for a particular architecture. Changing these parameters also transforms the smoother operator from a memory bandwidth bound problem into a compute bound problem. However, in practice one would not want to search for the optimal parameters manually. Auto-tuning of such architecture dependent parameters is common practice in BLAS libraries for dense linear algebra, e.g. ATLAS [35]. Similar techniques can be applied here.

By tuning the lower bound for the eigenvalues and selecting optimal B and s parameters, we achieved a $2.2 \times$ speedup compared to applying the Chebyshev smoother without tiles on the upper half of the error frequencies only. Tiling results in a larger optimal degree for the polynomial smoother, which in turn allows more freedom in selecting a suitable range of error frequencies to be smoothed.

Currently, the arithmetic intensity of the smoother is improved through tiling. However, the restriction and interpolation operators still have a low arithmetic intensity. To increase overall performance they also need to be more efficient with memory bandwidth. One way is to incorporate the restriction/interpolation into the tiled smoother routine such that the restriction or interpolation is calculated when the data is already in fast memory. This can avoid the penalty caused by copying the floor of the frustum to the separate memory for the tile, as observed in Figure 4. Such cache optimizations, similar to those presented by Douglas *et al* [16, 17, 18], might give the additional performance boost that is required to ensure that performance increases as the degree of the smoother increases.

In future work we will report on the scalability of the tiling approach on multi-core and many-core chips, e.g. GPU's. By copying the floor of the frustum in separate memory for each thread, we avoid the need for communication and synchronization between threads. This leads to a batch of fine grained independent tasks that can be easily scheduled over many cores. The lower bandwidth usage can also avoid bandwidth congestion. However, some care is needed in the choice of tile size since several threads may share a cache level.

An extension to 3D or unstructured meshes might lead to completely different results. In 3D a tile becomes a cube which has to fit in the cache. With current cache layouts the size of the cube is severely limited. Every additional smoothing step increases the redundant work and for small cube sizes the ratio of useful work to redundant work rapidly declines. However, for 3D, a wavefront implementation [11] can significantly reduce the redundant work. We will investigate this in future work as well.

ACKNOWLEDGEMENT

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

REFERENCES

1. Williams S, Waterman A, Patterson DA. Roofline: An insightful Visual Performance model for multicore Architectures. *Communications of the ACM* 2009; **52**(4):65–76.

2. Patterson DA, Hennessy JL. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, 2008.
3. Buttari A, Langou J, Kurzak J, Dongarra J. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing* 2009; **35**(1):38–53.
4. Datta K, Kamil S, Williams S, Olike L, Shalf J, Yelick K. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM review* 2009; **51**(1):129–159.
5. Williams S, Olike L, Vuduc R, Shalf J, Yelick K, Demmel J. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing* Mar 2009; **35**(3):178–194, doi:10.1016/j.parco.2008.12.006.
6. Vuduc R, Demmel JW, Yelick Ka. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series* Jan 2005; **16**:521–530, doi:10.1088/1742-6596/16/1/071.
7. Frigo M, Strumpen V. Cache oblivious stencil computations. *Proceedings of the 19th annual international conference on Supercomputing - ICS '05* 2005; **1**(212):361, doi:10.1145/1088149.1088197.
8. Frigo M, Leiserson C, Prokop H, Ramachandran S. Cache-oblivious algorithms. *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)* 1999; :285–297doi:10.1109/SFCS.1999.814600.
9. Wellein G, Hager G, Zeiser T, Wittmann M, Fehske H. Efficient Temporal Blocking for Stencil Computations by Multicore-Aware Wavefront Parallelization. *2009 33rd Annual IEEE International Computer Software and Applications Conference* 2009; :579–586doi:10.1109/COMPSAC.2009.82.
10. Treibig J, Wellein G, Hager G. Efficient multicore-aware parallelization strategies for iterative stencil computations. *Journal of Computational Science* 2011; .
11. Schäfer A, Fey D. High Performance Stencil Code Algorithms for GPGPUs. *Procedia Computer Science* 2011; **4**:2027–2036.
12. Baskaran M, Bordawekar R. Optimizing sparse matrix-vector multiplication on GPUs. *Technical Report*, IBM 2008.
13. Nguyen A, Satish N, Chhugani J, Kim C, Dubey P. *3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs*. IEEE, 2010, doi:10.1109/SC.2010.2.
14. Choi J, Singh A, Vuduc R. Model-driven autotuning of sparse matrix-vector multiply on GPUs. *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel computing*, ACM, 2010; 115–126.
15. Vuduc R, Chandramowlishwaran A, Choi J, Guney M, Shringarpure A. On the limits of GPU acceleration. *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, USENIX Association, 2010; 13–13.
16. Douglas C. Caching in with multigrid algorithms: problems in two dimensions. *International Journal of Parallel, Emergent and Distributed Systems* 1996; **9**(3):195–204.
17. Douglas C, Hu J, Kowarschik M, Rude U, Weiß C. Cache optimization for structured and unstructured grid multigrid. *Electronic Transactions on Numerical Analysis* 2000; **10**:21–40.
18. Douglas C, Hu J, Karl W, Kowarschik M, Rude U, Weiß C. Fixed and adaptive cache aware algorithms for multigrid methods. *Multigrid methods VI: proceedings of the Sixth European Multigrid Conference, held in Gent, Belgium, September 27-30, 1999*, vol. 14, Springer Verlag, 2000; 87.
19. Kowarschik M, Rude U, Weiß C, Karl W. Cache-aware multigrid methods for solving Poisson's equation in two dimensions. *Computing* 2000; **64**(4):381–399.
20. Strout M, Carter L, Ferrante J. Rescheduling for locality in sparse matrix computations. *Computational Science/CCS 2001* 2001; :137–146.
21. Kogge P, Bergman K, Borkar S, et al.. Exascale computing study: technology challenges in achieving exascale systems, september 2008, darpa 2008.
22. Demmel J, Hoemmen M, Mohiyuddin M, Yelick K. Avoiding communication in sparse matrix computations. *2008 IEEE International Symposium on Parallel and Distributed Processing* Apr 2008; :1–12doi:10.1109/IPDPS.2008.4536305.
23. Chronopoulos A, Gear C. S-Step Iterative Methods for Symmetric Linear Systems. *Journal of Computational and Applied Mathematics* Feb 1989; **25**(2):153–168, doi:10.1016/0377-0427(89)90045-9.
24. Hoemmen M. Communication-avoiding krylov subspace methods. PhD Thesis, EECS Department, University of California, Berkeley Apr 2010.
25. Golub G, Van Loan C. *Matrix computations*. Johns Hopkins Univ Pr, 1996.
26. Briggs WL, Henson VE, McCormick SF. *A Multigrid Tutorial, 2nd Edition*. SIAM, 2000.
27. Trottenberg U, Oosterlee C, Schüller A. *Multigrid*. Academic Pr, 2001.
28. Barrett R, Berry M, Chan TF, Demmel J, Donato J, Dongarra J, Eijkhout V, Pozo R, Romine C, der Vorst HV. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM: Philadelphia, PA, 1994.
29. Saad Y. *Iterative methods for sparse linear systems*. Society for Industrial Mathematics, 2003.
30. Adams M, Brezina M, Hu J, Tuminaro R. Parallel multigrid smoothing: polynomial versus Gauss–Seidel. *Journal of Computational Physics* 2003; **188**(2):593–610.
31. Manteuffel T. The Techebychev iteration for nonsymmetric linear systems. *Numerische Mathematik* 1977; **28**(3):307–327.
32. Manteuffel T. Adaptive procedure for estimating parameters for the nonsymmetric Techebychev iteration. *Numerische Mathematik* 1978; **31**(2):183–208.
33. Ashby S. ChebyCode, a FORTRAN implementation of Manteuffel's adaptive Chebyshev algorithm. *Technical Report*, University of Illinois, Department of Computer Science 1985.
34. Elman HC, Saad Y, Saylor PE. A Hybrid Chebyshev Krylov Subspace Algorithm for Solving Nonsymmetric Systems of Linear Equations. *SIAM Journal on Scientific and Statistical Computing* 1986; **7**(3):840, doi:10.1137/0907057.
35. Whaley RC, Dongarra J. Automatically Tuned Linear Algebra Software. *Technical Report UT-CS-97-366*, University of Tennessee December 1997.