A Performance Comparison of Algebraic Multigrid Preconditioners on CPUs, GPUs, and Xeon Phis

K. Rupp1,2*, J. Weinbub1, F. Rudolf1, A. Morhammer1, T. Grasser1, A. Jüngel2

1Institute for Microelectronics, TU Wien, Gußhausstraße 27–29/E360, A-1040 Wien, Austria
2Institute for Analysis and Scientific Computing, TU Wien, Wiedner Hauptstraße 8-10/E101, A-1040 Wien, Austria

SUMMARY

Algebraic multigrid preconditioners for accelerating iterative solvers are a popular choice for a broad range of applications, because they are able to obtain asymptotic optimality, yet can be applied in a black-box manner. However, only a few variants of algebraic multigrid preconditioners can fully benefit from fine-grained parallelization available on multi- and many-core architectures. Previous approaches were focused on graphics processing units from NVIDIA without a focus on fair comparisons in terms of power or price. We extend these earlier approaches to current high-end hardware from INTEL, NVIDIA, and AMD. Our results show that GPUs from NVIDIA and AMD are equally well suited for problems which are large enough to hide latencies across the PCI-Express bus, yet small enough to fit into GPU RAM. Purely CPU-based systems offer good performance across a much broader range of problem sizes and are even on par with GPUs in the regime where GPUs yield best performance. While we also demonstrate good performance on Xeon Phis, they were identified as the slowest platform overall in our benchmarks. Copyright © 2015 John Wiley & Sons, Ltd.

KEY WORDS: CPU, GPU, MIC, Multigrid, CUDA, OpenCL, OpenMP

1. INTRODUCTION

The predictive simulation of physical systems such as those in computational fluid dynamics, microelectronics, or solid mechanics usually involve the numerical solution of large partial differential equation systems. These numerical solutions are usually obtained via finite difference, finite element, or finite volume techniques, usually leading to large systems of equations. If the equations are nonlinear, a frequent choice is to use Newton’s method and solve linearized systems in each Newton step. This makes the efficient solution of sparse systems of linear equations not only important for systems described by linear partial differential equations, but also for systems described by nonlinear partial differential equations.

Sparse systems of linear equations with \( N \) unknowns of the form

\[
Ax = b
\]

with sparse system matrix \( A \in \mathbb{R}^{N \times N} \), unknown vector \( x \in \mathbb{R}^N \), and right hand side \( b \in \mathbb{R}^N \) may be solved by either sparse direct solvers, or by iterative methods. Direct solution approaches are attractive because they can be entirely run in a black-box manner and do not require the user to provide any kind of problem-specific information. While sparse direct solvers can also

*Correspondence to: Institute for Microelectronics, TU Wien, Gußhausstraße 27–29/E360, A-1040 Wien, Austria. Email: rupp@iue.tuwien.ac.at

Copyright © 2015 John Wiley & Sons, Ltd.

Prepared using nlaauth.cls (Version: 2010/05/13 v2.00)
be mapped well to multi-core architectures [1, 2], they become computationally expensive for problem sizes beyond several hundred thousands of unknowns, in which case iterative methods are preferred. The typical choice for such iterative solvers is to use a Krylov method [3] (e.g. conjugate gradients for symmetric positive definite systems, generalized minimum residuals in other cases), which is accelerated by a suitable preconditioner. Good preconditioners approximate the inverse $A^{-1}$ of the system matrix well, but are computationally cheap to apply to the residual vector $r_k = b - Ax_k$ for the $k$-th iterate $x_k$. A plethora of different preconditioners with vastly different degrees of sophistication have been proposed. We mention simple diagonal preconditioners, incomplete factorizations [4, 5], sparse approximate inverses [6, 7, 8], and algebraic multigrid (AMG) methods [9, 10, 11, 12], which have all been evaluated in the context of graphics processing units (GPUs).

AMG preconditioners are very popular for two reasons: First, they can be used as black-box preconditioners for many applications, because the grid hierarchy is built purely algebraically based on the entries in the system matrix only. Second, AMG preconditioners are asymptotically optimal for certain problem classes, i.e. the computational cost grows linearly with the number of unknowns $N$. The asymptotic optimality of AMG preconditioners is generally observed in a constant number of iterations of the preconditioned Krylov solver as the number of unknowns increases. In practice one may only obtain asymptotic optimality after adjusting certain parameters such as the type of smoother, cut-off thresholds for building coarse grids, or specify near-null-spaces for the construction of the prolongation operators. These parameters will be discussed along with the parallel AMG preconditioner considered in this work in Section 2.

The contributions of this work are as follows: First, we integrate contributions on performance enhancements on sparse matrix-vector multiplications [13], pipelined iterative solvers [14], and sparse matrix-matrix multiplications [15] into the parallel AMG framework initially outlined by Bell et al. [9]. This allows for a performance evaluation using the state-of-the-art for the individual components of AMG preconditioners. Second, all ingredients of this enhanced AMG framework are ported to the four multi- and many-core platforms considered for the benchmarks in Section 3: A dual-socket system equipped with INTEL Haswell CPUs, an AMD FirePro W9100 workstation GPU, an NVIDIA Tesla K20m GPU, and an INTEL Xeon Phi 7120 (Knights Corner). This enables a much broader evaluation of the suitability of current parallel hardware architectures for AMG preconditioners than in previous works, where usually only two platforms are compared. Because all four platforms are comparable in cost and power consumption, our results obtained for comparable implementations provide a fair performance comparison in the context of AMG preconditioners.

2. AGGREGATION-BASED ALGEBRAIC MULTIGRID

Given the original system of linear equations (1), AMG methods algebraically construct coarse grid matrices

$$A_{k+1} = R_k A_k P_k \quad \text{for } k = 0, \ldots, k_{\max}$$

with $A_0 \equiv A$ and suitable restriction operators $R_k$ and interpolation operators $P_k$. Even though $R_k$ and $P_k$ may be chosen independently for general systems, we will adhere to the usual choice $R_k = P_k^T$ such that any symmetric positive definiteness of $A_k$ is preserved in $A_{k+1}$. The coarsening is repeated until the coarse grid matrix $A_{k_{\max}}$ can be inverted directly at negligible cost. Even though no coarse grid in the geometric sense is constructed, we will continue to use the term grid when referring to the coupling of unknowns described by the nonzero values and hence the sparsity pattern in $A_k$.

The interpolation matrix $P_k$ in (2) fully describes the connection of the coarse grid to the fine grid by specifying how values defined on the coarse grid are interpolated. Therefore, the interpolation matrix is directly tied to the mechanism through which the coarse grid is constructed. Several approaches for the coarse grid construction have been proposed in the past, out of which aggregation strategies have received most attention for use with massively parallel architectures [9, 12].
A PERFORMANCE COMPARISON OF ALGEBRAIC MULTIGRID PRECONDITIONERS

Figure 1. Comparison of classical coarsening (left) and aggregation-based coarsening (right). Classical coarsening seeks to alternate coarse (blue, solid) and fine (red, dashed) nodes, whereas aggregation-based coarsening fuses a batch of neighboring nodes on the finer grid to a common coarse node.

classical approaches are sequential a-priori, so previous efforts have focused on coarse-grain parallelism for distributed memory machines [16]. Consequently, we will restrict our comparison to coarsening approaches based on aggregation and describe the state-of-the-art for the setup of aggregation-based AMG preconditioners in Section 2.1. Nevertheless, since the coarse grid construction is only one of several ingredients for a full AMG preconditioner, our findings are expected to carry over to AMG preconditioners with other coarsening strategies.

After the coarse grid operators are constructed, residuals on the finer grids are propagated to coarser grids, smoothed, and coarse grid corrections returned back to the fine grids until convergence is obtained. This solver cycle phase exposes ample of parallelism and maps well to massively parallel hardware. An overview of the operations encountered in the solver cycle phase as well as their efficient parallelization is given in Section 2.2.

2.1. Setup Phase

Summarizing the outline of AMG preconditioners above, a generic algorithmic description of the AMG preconditioner setup stage resulting in $k_{\text{max}} + 1$ levels is as follows [17]:

Algorithm 1: Setup of AMG Preconditioners

<table>
<thead>
<tr>
<th>Input: $A_0, B_0$</th>
<th>Output: $A_{k+1}, P_k, R_k$ for $k = 0, \ldots, k_{\text{max}} - 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>for $k = 0, \ldots, k_{\text{max}} - 1$ do</td>
</tr>
<tr>
<td></td>
<td>// until coarse problem can be solved directly</td>
</tr>
<tr>
<td>1</td>
<td>Determine coarse nodes</td>
</tr>
<tr>
<td>2</td>
<td>Build interpolation operator $P_k$ based on coarse nodes and $B_0$</td>
</tr>
<tr>
<td>3</td>
<td>$R_k \leftarrow P_k^T$</td>
</tr>
<tr>
<td>4</td>
<td>$A_{k+1} \leftarrow R_k A_k P_k$</td>
</tr>
</tbody>
</table>

The near null-space vectors provided by the user with $B_0$ allow for a better application-specific choice of the interpolation operator and are discussed in more details below. In the subsequent sections we discuss the parallel implementation of lines 2, 3, 4, and 5.

2.1.1. Parallel Maximal Independent Set Classical AMG variants use sequential algorithms to determine the coarse grid. A parallel coarse grid construction for AMG based on aggregation has been proposed and demonstrated for NVIDIA GPUs by Bell et al. [9], cf. Figure 1. The algorithm is based on parallel maximal independent sets (parallel MIS) using randomization for the selection of independent coarse nodes in parallel and can be seen as a variant of Luby’s method [18]. It returns a set of coarse nodes such that any two nodes are at least a distance of $d$ edge connections apart. At the same time, the set is maximal in the sense that no additional node can be added to the set without violating the distance-$d$ property.
In the following we present a variation of the original distance- \( d \) algorithm by Bell et al., where we include the weighting based on the number of strong connections as proposed by Gandham et al. [12]. The algorithm requires two sets of work arrays \( T \) and \( T' \), each holding a triple of node state, node weight, and node index for each node. A lexicographical ordering among the triples is used, i.e. a comparison is based on the first component and only on equality the next component is considered.

![Algorithm 2: Parallel Maximal Independent Set (distance \( d \))]
cycle stage. This avoids an increase in the number of nonzeros in $A_k$ at coarser levels as is observed with smoothed aggregation, but entails additional computational work in the solver cycle stage.

2.1.3. Restriction Once the interpolation operator $P_k$ is set up, the restriction operator is obtained via an explicit transposition $R_k = P_k^T$. The computational cost of such a transposition is usually low compared to other operations and may hence also be carried out sequentially on a single CPU core.

A parallel matrix transposition was described by Bell et al. [9], where the sparse matrix is converted to the coordinate format such that each nonzero entry is given by a triplet of row-index, column-index, and value. The column index array is then sorted in a stable manner such that the order of identical column indices is preserved. Then, the permutation describing the reordering of column indices is applied to the array of row indices and values. Finally, the array of row and column indices is swapped, resulting in the transposed matrix in coordinate format. A conversion to the compressed sparse row format can be applied easily at negligible cost if needed.

2.1.4. Galerkin Product The computationally most demanding step in the setup stage is the computation of the Galerkin products (2). Each Galerkin product is computed using two sparse matrix-matrix multiplications (SpGEMM): First, $W_k = A_k \times P_k$ is determined, then $A_{k+1} = R_k \times W_k$ is obtained in a second step.

A fast sparse matrix-matrix multiplication based on row merging has been proposed recently for NVIDIA GPUs by Gremse et al. [19]. We have refined and extended these ideas to also obtain improved sparse matrix-matrix multiplication implementations on INTEL CPUs, INTEL Xeon Phis, AMD GPUs, and NVIDIA GPUs [15], resulting in considerable performance gains compared to the initial implementation used by Bell et al. [9]. The basic ideas are outlined in the following.

We assume the sparse matrices to be stored in the compressed sparse row (CSR) format with array triplets $\{I_A, J_A, V_A\}, \{I_B, J_B, V_B\}, \{I_C, J_C, V_C\}$ for the matrices $A$, $B$, and $C$, respectively. Column indices in each row are assumed to be sorted in ascending order.

Let $a_i, i = 1, \ldots, M$ be the rows of $A$, $b_i, i = 1, \ldots, K$ be the rows of $B$, and $c_i, i = 1, \ldots, N$ be the rows of $C$. Then, each row $c_i$ of $C = AB$ is computed as

$$c_i = \sum_{a_i,j \neq 0} a_{i,j} b_j.$$  

(5)

Even though (5) can be written as the sparse matrix-vector product $c_i = a_i B$, the sparsity of $a_i$ makes a direct reuse of optimized kernels for sparse matrix-vector products inefficient in general, since $c_i$ would become a dense vector. Instead, the computation of the sparse rows $c_i$ requires the scaled summation of sparse rows $b_j$, which we will discuss later.

Because the result rows $c_i$ can be computed independently, a parallelization over all result rows is straightforward. However, the memory locations of the resulting indices and values of $C$ in CSR format is a-priori unknown, since the number of nonzero entries in each $c_i$ is unknown. Consequently, the SpGEMM operation can be formally split into an analysis phase, where the number of nonzero entries in $c_i$ is either computed exactly or only an upper bound is estimated, and a computation phase, where the numerical entries in $c_i$ are computed.

Implementations which first store each $c_i$ in a separately allocated buffer and then transfer the data over to a representation of $C$ in CSR format are inherently memory-inefficient: The memory required for the temporary buffers is at least as large as the memory required for storing $C$ in CSR format and is undesired in practice, because the available system memory is exhausted earlier than necessary. Furthermore, the buffer allocation for each $c_i$ may have significant overhead if $A$ and $B$ are very sparse and cache misses will increase because of reduced data locality. Thus, we require the analysis phase to not only derive upper bounds on the number of nonzeros in each row in $C$, but instead to compute the exact sparsity pattern. Since the determination of the exact sparsity pattern requires some scratchpad memory, the analysis phase can be further split into a preprocessing step for making enough scratchpad memory available to each thread, and the computation of each row length $c_i$ using the scratchpad memory.
Overall, our SpGEMM implementations consist of three stages:

- **Stage 1: Determine Scratchpad Size.** Compute upper bounds for the row lengths in parallel via

  \[
  \text{nnz}(c_i) \leq \min \left( \sum_{a_{i,j} \neq 0} \text{nnz}(b_j), N \right),
  \]  

  (6)

  where \( \text{nnz}(\cdot) \) denotes the number of nonzeros of the argument. Then, allocate scratchpad memory for each thread (or thread group on GPUs) such that all rows \( c_i \) can be processed by the respective thread (or thread group). Note that the scratchpad memory is reused for multiple rows and thus the extra memory is usually negligible. If a global bound \( b_{\text{row}}^B \) on the number of nonzeros for each of the rows in \( B \) is available, the upper bound

  \[
  \text{nnz}(c_i) \leq \min(\text{nnz}(a_i) \times b_{\text{row}}^B, N)
  \]

  can be used instead, which is more pessimistic, yet cheaper to compute because the nonzeros of \( a_i \) do not need to be traversed. Such a cheaper estimate is particularly attractive if \( A \) and \( B \) are extremely sparse, since the computation of (6) may already entail significant overhead.

- **Stage 2: Compute Row Lengths.** With scratchpad memory available for each thread (or thread group on GPUs), the length of each row \( c_i \) can be computed. This is achieved by merging the nonzero indices of the relevant rows \( b_j \) by exploiting available hardware features: On multi-core CPUs we use the AVX2 instruction set to process up to eight rows concurrently, while on the Xeon Phi we use the 512-bit wide AVX vector registers to merge up to 16 rows concurrently. With NVIDIA GPUs we merge up to 32 rows and on AMD GPUs up to 64 rows at the same time. After all rows \( b_j \) are merged, the row lengths are directly stored in the CSR-array \( IC \), which is of the same size as \( IA \) and can be allocated at the very beginning of the SpGEMM kernel.

- **Stage 3: Fill CSR-structure of \( C \).** After all row lengths have been computed, a parallel prefix summation (exclusive scan) is performed in-place in \( IC \) to obtain the correct entries. Then, the buffers \( JC \) and \( VC \) are allocated based on the size information in the last entry in \( IC \). Finally, the computation of the column indices in \( JC \) is similar to the previous step, except that the intermediate results in the scratchpad memory are not discarded, but instead written to their respective locations in \( JC \). While computing the column indices, the values in \( VC \) are computed with the same control flow: Whenever column indices from different rows in \( B \) are merged, the respective values are accumulated.

The similar computations carried out in the second and third stage may seem redundant, but are necessary for minimizing memory footprint and avoiding expensive memory reallocations during the computation of \( C \). Also, the redundant loads only affect the indices, whereas the nonzero values are loaded only once.

### 2.2. Cycle Phase

The constructed AMG preconditioner is applied as a preconditioner to a Krylov method. In each application of the preconditioner, the AMG hierarchy is traversed recursively in order to improve the current approximation to the solution. We consider only so-called \( V \)-cycles in this work, which visits each multigrid level exactly once:
Algorithm 3: AMG V-Cycle function \( V_{\text{Cycle}} \)

**Input:** level \( k \), right hand side \( b_k \), initial guess \( x_k \)

**Output:** updated guess \( x_k \)

1. \( x_k \leftarrow S_k(b_k, A_k, x_k) \) \quad // pre-smoothing
2. \( r_k \leftarrow b_k - A_k x_k \) \quad // compute residual
3. \( r_{k+1} \leftarrow R_k r_k \) \quad // restrict residual to coarse grid
4. if \( k = k_{\text{max}} - 1 \) then
   5. \( x_{k+1} \leftarrow A^{-1}_{k+1} r_{k+1} \) \quad // exact direct solution on coarsest grid
5. else
   6. \( x_{k+1} \leftarrow V_{\text{Cycle}}(k+1, r_{k+1}, 0) \) \quad // apply V-Cycle on coarser grid
   7. \( x_k \leftarrow x_k + P_k x_{k+1} \) \quad // correction from coarse grid
   8. \( x_k \leftarrow S_k(b_k, A_k, x_k) \) \quad // post-smoothing

Here, \( S_k \) denotes a smoother at level \( k \) which is responsible for damping high-frequency errors corresponding to the matrix \( A_k \). Typical choices are Jacobi or damped Jacobi smoothers in a massively parallel setting. Symmetric Gauss-Seidel relaxation is a popular choice for sequential processing, but requires additional techniques such as coloring in order to yield reasonable performance in parallel. Numerical experiments indicate that damped Jacobi is scalable and reasonably efficient, so the additional setup effort for symmetric Gauss-Seidel relaxation does not pay off overall [12].

Other multigrid cycles such as \( W \)-cycles can be used instead of \( V \)-cycles without additional complications. Gandham *et al.* reported the use of \( K \)-cycles, where Krylov iterations are recursively applied on coarse levels, through which asymptotic optimality of an unsmoothed aggregation scheme was retained [12]. The drawback of \( K \)-cycles, however, is that the number of corrections on coarse grids may grow exponentially. This is particularly a concern when using GPU acceleration, because the increased work on coarse grids with smaller numbers of unknowns favors an execution on the CPU.

The computational hot-spot in an outer Krylov method is usually the sparse matrix-vector product. Only in cases where the Krylov space is stored explicitly and repeated orthogonalizations are executed (as is the case with e.g. the generalized minimum residual method), a significant share of time may be spent in vector orthogonalizations at later iterations. Similarly, the main cost of AMG cycles stems from the pre- and post-smoothing steps as well as from the inter-grid transfers through restriction and interpolation. The smoothing operations using a damped Jacobi method are very similar in structure to matrix-vector products. Inter-grid transfers are sparse matrix-vector multiplications by nature. Therefore, fast sparse matrix-vector products are essential for good performance of AMG preconditioners in the solver cycle stage.

### 2.2.1. Sparse Matrix-Vector Products

The importance of efficient matrix-vector products is reflected in a lot of research conducted recently for GPUs [20, 21, 22, 23, 24] as well as Xeon Phi [25]. Different storage formats tailored explicitly to maximize the performance for sparse matrix-vector products have been proposed, arguing that the additional one-time conversion overhead is justified by the performance gains of many subsequent products. With the development of CSR-adaptive by Greathouse and Daga [13], a fast sparse matrix-vector product has become available for the standard CSR format, eliminating the need for any sparse matrix conversions and the memory overheads associated with the conversion. Since the CSR format has already been a crucial ingredient for fast sparse matrix-matrix products discussed earlier, CSR-adaptive enables full solvers to run efficiently using a single, widely used sparse matrix storage format.

The CSR-adaptive kernel for fast matrix-vector products has been developed originally for AMD GPUs. However, we found that the kernel also performs very well on NVIDIA GPUs, because the structured (coalesced) memory accesses are beneficial on both architectures. The key idea is to consider groups of rows for each thread group and load the respective data with fast, regular memory access to shared on-chip memory. Based on the lengths of the rows loaded to shared memory, the thread group can then either work on a full row if the row length is large, or one thread per row computes the result entry if the
FirePro W9100 | 2x Xeon E5-2670 | Xeon Phi 7120P | Tesla K20m
---|---|---|---
Vendor | AMD | INTEL | INTEL | NVIDIA
Mem. BW (GB/sec) | 320 | 136 | 352 | 208
GFLOP/sec (float) | 5238 | 1766 | 2416 | 4106
GFLOP/sec (double) | 2619 | 884 | 1208 | 1173
TDP (Watt) | 275 | 240 | 300 | 244

Table I. Overview of hardware used for the comparison. All values are theoretical peaks. Practical peaks for memory bandwidth and floating point operations per seconds are around 70 to 80 percent of the theoretical peaks, except for the Xeon Phi, where only 50 percent of the theoretical peak bandwidth can be obtained [25].

Row lengths are small. Unstructured memory accesses are effectively shifted from slower global memory to fast shared on-chip memory, thus improving overall performance [13]:

Algorithm 4: CSR-Adaptive Sparse Matrix-Vector Product Kernel Skeleton

```
Input: Row block metadata row_blocks; CSR data arrays I, J, V for A; vector x; threshold parameter 
Output: y = Ax 
1 foreach thread group with index workgroupID do 
2   startRow ← row_blocks[workgroupID] 
3   nextStartRow ← row_blocks[workgroupID + 1] 
4   numRows ← nextStartRow - startRow 
5   // Buffer data from J, V for rows {startRow, ..., nextStartRow-1} in shared memory here 
6   if numRows > threshold then 
7     // Compute result entries using one thread per result entry yi 
8     // i ∈ {startRow, ..., nextStartRow-1} in parallel 
9     yi = ∑r∈[Ii:Ii+1) Vr x Jr 
10    else 
11     // Use all threads for computing each result entry yi 
12     // i ∈ {startRow, ..., nextStartRow-1} sequentially 
13     yi = parallel sum r∈[Ii:Ii+1) (Vr x Jr) 
```

Array subscripts denote the respective entry in the array. Similarly, array ranges are denoted using an index range as subscript. The threshold parameter in Algorithm 4 is usually chosen between 1 and 10. We have chosen a threshold of 1, as this choice enables some code simplifications which reduce code maintenance efforts and at the same time increase the occupancy of the compute units.

3. BENCHMARKS

We compared our implementations of a parallel (non-smooth) aggregation AMG preconditioner and a parallel smoothed aggregation AMG preconditioner as outlined in the previous section on current high-end hardware: A dual-socket system equipped with two INTEL Xeon E5-2670 v3 (Haswell) twelve-core CPUs, an AMD FirePro W9100 workstation GPU, an NVIDIA Tesla K20m GPU, and an INTEL Xeon Phi 7120 (Knights Corner) have been used for the comparison, cf. Table I. Memory error correction was enabled for the NVIDIA Tesla K20m GPU and disabled for AMD FirePro W9100 on the machines we had available for the benchmarks. OpenMP threads have been pinned to cores to avoid undesirable thread migrations. All reported values represent median values from five runs.

We consider linear finite element discretizations of the Poisson equation $\Delta u = 1$ with homogeneous Dirichlet boundary conditions at the $x$ and $y$ boundaries on unstructured simplex meshes of the unit square and unit cube, respectively. This choice allows for a better comparison with earlier work [9, 12] and is from the computational point of view still representative for more complicated problems. Since the resulting system matrices are symmetric positive definite, we use a conjugate gradient (CG) solver for the Krylov solver. For better comparison with other available
A PERFORMANCE COMPARISON OF ALGEBRAIC MULTIGRID PRECONDITIONERS

Figure 2. Comparison of the number of solver iterations (left) and levels in the AMG hierarchy (right) required for solving the two-dimensional Poisson equation when using a classical one-pass Ruge-Stüben AMG preconditioner (One-Pass RS), an aggregation-based AMG preconditioner, and a smoothed aggregation AMG preconditioner. Asymptotic optimality is only achieved with classical one-pass Ruge-Stüben AMG, at the expense of a higher number of levels.

solver alternatives we also include a pipelined unpreconditioned CG solver [14] (or, equivalently, diagonally preconditioned CG) as well as classical AMG implementations using single-threaded one-pass Ruge-Stüben coarsening supplemented with direct interpolation [17]. The number of pre- and post-smoothing steps was chosen such that the smallest time to solution was obtained: One pre- and one post-smoothing step at each level in the multigrid hierarchy are applied for the classical AMG implementation, whereas the aggregation AMG preconditioner as well as the smoothed aggregation AMG preconditioner use two pre- and post-smoothing steps. The coarsest unstructured meshes were generated using Netgen [26] and then subsequently refined uniformly to obtain higher resolution. Note that this simple setting ensures good convergence of the unpreconditioned solvers relative to solvers with AMG preconditioners. Thus, AMG preconditioners are likely to perform better relative to unpreconditioned solvers in practical situations involving e.g. inhomogeneous or anisotropic diffusion.

3.1. Poisson Equation in 2D

A comparison of solver iterations required to reach a relative reduction of the initial residual by a factor $10^{-8}$ and the number of levels in the AMG hierarchy is given in Figure 2. Aggregation-based coarsening is more aggressive than classical coarsening and thus results in a smaller number of levels, but is not asymptotically optimal. Smoothed aggregation coarsening is even more aggressive and requires only four levels, but nevertheless shows better asymptotic behavior: The number of iterations grows only by a factor of two for system sizes ranging over three decades. Looking at only the number of solver iterations required, the unpreconditioned solver is not an attractive choice at all.

Solver setup times, solver cycle times, and total solver execution times (i.e. the sum of the former two) for each of the four hardware platforms are depicted in Figure 3. The sequential one-pass Ruge-Stüben AMG preconditioner takes about one order of magnitude longer to set up than the two parallel AMG preconditioners. Kernel launch overheads on GPUs as well as OpenMP synchronization costs become visible for problem sizes below 100 000 unknowns. The highest overheads are observed for the AMD FirePro W9100, where problem sizes above 1 000 000 are needed for negligible overheads.

The solver cycle times in the center column of Figure 3 reflect the small number of iterations required for the classical one-pass Ruge-Stüben AMG preconditioner. Similarly, the smoothed aggregation AMG preconditioner outperforms the aggregation-based AMG preconditioner because of the smaller number of iterations needed. The better asymptotic behavior of the aggregation-based AMG preconditioner over the unpreconditioned pipelined CG solver only starts to show at very large problem sizes above 1 000 000 unknowns, but the gains are very mild.
Figure 3. Comparison of solver setup times (left), solver cycle times (center), and total solver times (right) of classical one-pass Ruge-Stüben coarsening (One-Pass RS), aggregation-based AMG, and smoothed aggregation AMG for the two-dimensional Poisson equation on each of the four hardware platforms considered. The setup for the sequential classical one-pass coarsening has been run on a single CPU cores for the AMD and NVIDIA GPUs. Overall, performance differences among the AMG methods are fairly small despite of their different algorithmic nature.

Total solver execution times in the right column of Figure 3 show that the smoothed aggregation preconditioner is the best choice for problem sizes above 200,000 on all four hardware platforms. The unpreconditioned solver is a better choice for smaller problem sizes in the benchmark setting considered here. In contrast to the classical one-pass Ruge-Stüben AMG preconditioner, which spends most time in the setup stage, and in contrast to the aggregation-based AMG preconditioner, which spends most time in the solver cycle stage, the smoothed aggregation AMG preconditioner spends about the same amount of time in each of the two stages. This balance is visible on all four hardware platforms.

The results from Figure 3 are collected in Figure 4 for a better comparison of the individual hardware platforms. Overall, the performance differences are relatively small for the largest problem sizes: The dual INTEL Xeon E5-2670 v3 system is on par with the AMD FirePro W9100, while the Tesla K20m is about 50 percent slower. The CPU-based execution is the best choice between 100,000 and 1,000,000 unknowns. A further improvement of execution times on the CPU for problem sizes below 100,000 is possible by selectively disabling OpenMP for small loops. Optimizations of the hardware or the driver stack for small problem sizes are also desirable for the AMD FirePro.
A PERFORMANCE COMPARISON OF ALGEBRAIC MULTIGRID PRECONDITIONERS

Figure 4. Comparison of total solver time (i.e. AMG setup time plus solver cycle time) for solving the two-dimensional Poisson equation using an unpreconditioned iterative solver, and a smoothed aggregation AMG preconditioner on each of the four hardware platforms.

Figure 5. Comparison of the number of solver iterations (left) and levels in the AMG hierarchy (right) required for solving the three-dimensional Poisson equation when using a classical one-pass Ruge-Stüber AMG preconditioner (One-Pass RS), an aggregation-based AMG preconditioner, and a smoothed aggregation AMG preconditioner. Asymptotic optimality is only achieved with classical one-pass Ruge-Stüber AMG, at the expense of a higher number of levels.

W9100 and the INTEL Xeon Phi, which do not take less than 0.1 seconds overall even for very small problem sizes.

3.2. Poisson Equation in 3D

The setup for the three-dimensional Poisson equation is similar to the two-dimensional case. Similar behavior in terms of the number of iterations required for convergence and the number of levels in the multigrid hierarchy is obtained for the aggregation-based AMG preconditioner and the smoothed aggregation AMG preconditioner as for the two-dimensional case: The number of iterations grows faster for the aggregation-based variant, cf. Figure 5. Asymptotic optimality of the classical one-pass Ruge-Stüber AMG preconditioner is not obtained. The reason is that the same threshold for strong connections was used as in the two-dimensional case.

The plots of solver setup time and solver cycle time for each of the preconditioners and hardware platforms considered is given in Figure 6. Overall, the fastest time to solution is obtained when
running a pipelined conjugate gradient solver without preconditioner. This can be explained by the denser system matrices in the three-dimensional case, which results in substantially higher computational effort for computing Galerkin products, which depend quadratically on the number of nonzeros per row. In contrast, the computational work for matrix-vector products, which are the bottleneck for the unpreconditioned solver, only depends linearly on the average number of nonzeros per row. Furthermore, the condition number of the system matrix grows slower with the number of unknowns, hence the number of iterations required for convergence grows slower.

A final comparison of the four different hardware platforms in Figure 7 shows that for the largest system sizes all results are within a factor of three. Among the AMG preconditioners, the dual CPU configuration is faster than the GPUs and the Xeon Phi. This is primarily due to the faster Galerkin products, which benefit from the large caches of the CPUs. The CPU is also the best choice for smaller system sizes, even though our benchmarks included OpenMP synchronization overheads, which can be easily overcome by selecting the onset of OpenMP parallelization for matrix and vector operations appropriately.
4. SUMMARY AND CONCLUSIONS

We extended previous parallel approaches for AMG preconditioners to current high-end hardware from INTEL, NVIDIA, and AMD in order to conduct a fair performance comparison of the different platforms. Our results show that GPUs from NVIDIA and AMD are equally well suited for problem sizes large enough to hide latencies across the PCI-Express bus, yet small enough to fit into GPU RAM. The sweet spot for NVIDIA GPUs is about an order of magnitude larger than for AMD GPUs, because the latter suffer from significantly kernel launch overheads. Although the Xeon Phi is able to provide performance comparable to that of GPUs, it was the slowest platform overall in our benchmarks.

Best overall performance, however, was obtained on the dual-CPU system. This is best explained by the large caches, which provide best performance for sparse matrix-matrix products in the setup stage of AMG preconditioners. Moreover, purely CPU-based systems are able to scale to smaller as well as larger problem sizes than GPUs or coprocessors, because they neither suffer from PCI-Express overhead, nor is the main memory limited to relatively small size.

Another finding of our benchmarks is that a good AMG preconditioner should be balanced: Neither should too much time be spent on the preconditioner setup, nor should almost all time be spent on the solver cycle phase. It is unlikely that such a choice can be fully automated, hence users will need to be aware of the most important parameters to adjust. Also, unpreconditioned or diagonally preconditioned iterative solvers may also be a viable choice for rather well-behaved problems, since they map well to the underlying hardware.

ACKNOWLEDGEMENT

This work has been supported by the Austrian Science Fund (FWF), grants P23296 and P23598, by the European Research Council (ERC) through the grant #247056 MOSILSPIN, and by the VSC Research Center funded by the Austrian Federal Ministry of Science, Research and Economy (bmwfw). The authors are grateful to Joachim Schöberl for providing access to the hardware used for performance comparisons in this work.
REFERENCES


