AMGX: A LIBRARY FOR GPU ACCELERATED ALGEBRAIC MULTIGRID AND PRECONDITIONED ITERATIVE METHODS

M. NAUMOV†, M. ARSAEV‡, P. CASTONGUAY†, J. COHEN†, J. DEMOUTH§, J. EATON¶, S. LAYTON†, N. MARKOVSKYI‡, I. REGULY†, N. SAKHARNYKH†, V. SELAPPAN†, AND R. STRZODKA†

Abstract. The solution of large sparse linear systems arises in many applications, such as computational fluid dynamics and oil reservoir simulation. In realistic cases the matrices are often so large that they require large scale distributed parallel computing to obtain the solution of interest in a reasonable time. In this paper we discuss the design and implementation of the AmgX library, which provides drop-in GPU acceleration of distributed algebraic multigrid (AMG) and preconditioned iterative methods. The AmgX library implements both classical and aggregation-based AMG methods with different selector and interpolation strategies, along with a variety of smoothers and preconditioners, including block-Jacobi, Gauss–Seidel, and incomplete-LU factorization. The library contains many of the standard and flexible preconditioned Krylov subspace iterative methods, which can be combined with any of the available multigrid methods or simpler preconditioners. The parallelism in the aggregation scheme exploits parallel graph matching techniques, while the smoothers and preconditioners often rely on parallel graph coloring algorithms. The AMG algorithm implemented in the AmgX library achieves 2–5× speedup on a single GPU against a competitive implementation on the CPU. As will be shown in the numerical experiments section, both setup and solve phases scale well across multiple nodes, sustaining this performance advantage.

Key words. AMG, aggregation, classical, preconditioned iterative methods, ILU, graph matching, graph coloring, level-scheduling, MPI, CUDA, GPU, CFD, reservoir simulation

AMS subject classifications. 15, 35, 65, 68, 76

DOI. 10.1137/140980260

1. Introduction. The design and manufacturing of cars, boats, and aircraft as well as efficient extraction of oil and gas often rely on accurate simulations performed by large software packages such as ANSYS® Fluent [5] and Schlumberger® Eclipse/Intersect [50, 51], respectively. At the core of these packages lies the solution of large sparse linear systems, which often takes more than 60% of the total simulation time.

For example, we are usually interested in solving a set of $k$ linear systems

$$A_i x_i = f_i \quad \text{for } i = 1, \ldots, k,$$

where the coefficient matrix $A_i \in \mathbb{R}^{n \times n}$, solution $x_i$, and right-hand side $f_i \in \mathbb{R}^{n \times 1}$. These systems are often constructed as part of the nonlinear process for $k$ number of steps, where either the coefficient matrix changes at every iteration or the matrix does not change, but the right-hand side and the solution do change from one step to the next.

*Received by the editors August 1, 2014; accepted for publication (in revised form) July 20, 2015; published electronically October 29, 2015.

†Nvidia, 2701 San Tomas Expressway, Santa Clara, CA 95050 (mnaumov@nvidia.com, patricecastonguay@gmail.com, jocohen@nvidia.com, slayton@nvidia.com, regulyistvan@gmail.com, nsakharnykh@nvidia.com, vsellappan@nvidia.com, strzodka@mpi-inf.mpg.de).
‡Nvidia, 12 Dvintsev St., Moscow 127018, Russia (marsaev@nvidia.com, nmarkovskiy@nvidia.com).
§Nvidia, Le Colisée, 12 avenue de l’Arche, Courbevoie 92400, France (jdemouth@nvidia.com).
¶Nvidia, 11001 Lakeline Blvd., Austin, TX 78717 (jeaton@nvidia.com).
In realistic cases, these systems are often so large that they do not fit into the memory of a single machine, and large clusters must be used to find the solution. In this setting direct solution of these linear systems becomes prohibitively expensive from the memory standpoint and also does not scale well with the increased number of nodes. Therefore, multigrid and iterative methods that can attain a less accurate solution much faster are often preferred.

The implementation of algebraic multigrid (AMG) has been considered by many authors [11, 55]. In particular, the unsmoothed aggregation based on local heuristics has been studied in [37, 40, 45], and the smoothed aggregation has been proposed in [58, 57], while the classical approach based on global heuristics has been considered in [25, 20]. Finally, the implementation of AMG on different parallel CPU-based computer platforms has been considered in [2, 19, 56]. We will build upon many of these existing techniques when developing the AMG on distributed platforms with multiple GPUs.

The early attempts of utilizing the GPU for the AMG focused simply on off-loading the matrix-vector multiplication during the solve phase of the algorithm [30, 38]. Since then, Bell, Dalton, and Olson have shown how to expose fine-grained parallelism within a single node in the setup and solve phases of the smoothed aggregation-based algorithm using Thrust library primitives [13]. Finally, additional work on the setup and solve phases with K-cycle on a single node has been done in [12].

The AmgX library can be seen almost as a super set of the above algorithms. It implements both unsmoothed aggregation- and classical-based AMG methods with different selector and interpolation strategies. It also contains many of the standard preconditioned Krylov subspace iterative methods [7, 48, 44, 59], with a variety of smoothers and preconditioners, including block-Jacobi, Gauss–Seidel, and incomplete-LU (ILU) [49]. A highlight of the library is the full configurability of a solver hierarchy with arbitrary depth, in which the outer solver uses inner solvers as preconditioners, which themselves can also be preconditioned by other methods. This allows the user to quickly experiment with a variety of inner-outer schemes often discussed in the scientific literature [53].

In contrast to earlier approaches, the AmgX library also takes advantage of multiple GPUs and allows handling of very large sparse linear systems that fit into the aggregate memory of all GPUs present in the system. Within a single node, in the setup phase we rely on existing parallel graph matching techniques [35, 52, 4], while the smoothers and preconditioners take advantage of parallel graph coloring algorithms [1, 3, 39, 43]. In a distributed environment, the AmgX library relies on graph partitioning algorithms [36, 15] and uses techniques based on rings of nearest neighbors to keep track of communication. In this setting, only the required halo elements are communicated across different nodes. The latency of these transfers is hidden by overlapping communication and computation. Moreover, if the global problem becomes too small to fill all active GPUs with work, consolidation onto fewer GPUs is performed, which again allows the library to minimize communication costs while fully taking advantage of computational resources at hand. We will discuss these in more detail in the following sections.

Finally, as will be shown in the numerical experiments section (section 5), the aggregation-based AMG algorithm implemented in AmgX achieves 2–4 × speedup on a single GPU when compared to commercial proprietary ANSYS implementation on the CPU. Also, the classical-based AMG algorithm achieves 2–5 × speedup on a single
GPU when compared to HYPRE on the CPU [26]. Both approaches scale well across multiple nodes, sustaining this performance advantage. We point out that we do not have a license for another competitive commercial package called GAMPACK, which to the best of our knowledge implements the classical-based AMG on multiple nodes, targeting problems arising from the oil reservoir simulations [54]. Therefore, we do not present comparisons with it.

2. Algebraic multigrid (AMG). In general, the AMG algorithm solves the large (fine) linear system by cycling through levels composed of smaller (coarse) linear systems and finding updates that bring one closer to the exact solution. For example, often the update $d = P e$ for the approximate solution of the fine linear system

\begin{equation}
Ax = f
\end{equation}

is computed based on the solution of the coarse linear system

\begin{equation}
Ce = g,
\end{equation}

where the coefficient matrix is obtained using the sparse matrix product $C = RAP$ and the right-hand side $g = Rr$ is obtained by restricting the residual $r = f - Ax$. Here, the prolongation $P$ and restriction $R$ matrices allow us to transition from coarse to fine and fine to coarse levels of the grid, respectively. Different cycling strategies define level transition patterns, such as the popular V-, W- and F-cycles.

Furthermore, the updated solution is often improved by smoothing, normally using a preconditioned Richardson’s iteration of the form

\begin{equation}
x_{j+1} = x_j + \omega M^{-1}(f - Ax_j),
\end{equation}

with different choices of the preconditioner $M$ defining individual smoothers, such as Jacobi, Gauss–Seidel, or ILU, and scalar $\omega$ being the so-called relaxation factor.

We separate the algorithm into two phases. In the setup phase we construct matrices $P$, $R$, and $C$ for the given problem and recurse until the coarsest problem has less than some specified “small enough” number of rows. We choose to let $R = P^T$.

In the solve phase we start with an initial guess and go through a smoothing and correction cycle, transitioning between levels, to find a better solution. For example, a V-cycle with two levels and pre- and postsmoothing is illustrated in Figure 1.

Here, different choices of prolongation matrix $P$ define individual AMG methods.

For example, in unsmoothed aggregation-based AMG, the matrix $P$ is formed such that the fine grid variables are grouped into disjoint coarse aggregates, typically...
of size 2–16 variables each. The formation of $P$ is done in a single step, with the sparsity pattern defined by this grouping and all nonzero values set to 1. In the next section we will discuss how to find such a $P$ using graph matching techniques, and later on we will use size 8 aggregates in our numerical experiments.

On the other hand, in classical-based AMG, the matrix $P$ is formed in two steps. In the coarsening step, we find its sparsity pattern, for example, using the parallel maximal independent set (PMIS) algorithm. In the interpolation step, we find its nonzero values, for example, constructing a distance-one (D1), distance-two (D2), or multipass interpolation [60]. The interpolation step often operates on nonlocal variables and results in a more expensive setup, but a better convergence during solve.

3. Parallel algorithms within a single node (single GPU). There are many algorithms in the AmgX library that are critical to ensuring its performance advantage on the GPU. In this section we discuss three of these algorithms: (i) parallel graph matching used in the aggregation path, (ii) the sparse matrix multiplication by another sparse matrix used in the classical path, and (iii) the ILU factorization often used as a black-box smoother and preconditioner in AMG and iterative methods, respectively. These algorithms stand out because they illustrate the fine-grained parallelization of relatively complex computational patterns and because the general techniques used to exploit them are also applicable to other problems and hardware. Ultimately, the goal of these techniques is to uncover enough parallelism to take full advantage of the memory bandwidth.

3.1. Graph matching—aggregation. In order to solve a linear system with AMG we must first set up a hierarchy of levels (the $P$, $R$, and $C$ matrices mentioned earlier). In this section we focus on algorithms needed to set up this hierarchy in the aggregation-based AMG method. We restrict ourselves to a hierarchy of two levels, which can be trivially generalized to any number of levels by recursion. We call the larger first level fine and the smaller second level coarse.

As an example, let us consider the symmetric coefficient matrix

\[
A = \begin{pmatrix}
  a_{11} & a_{14} & a_{15} & a_{26} \\
  a_{22} & a_{33} & a_{37} & 0 \\
  a_{41} & a_{44} & a_{48} & 0 \\
  0 & a_{55} & a_{59} & 0 \\
  a_{62} & 0 & a_{66} & 0 \\
  0 & a_{73} & a_{77} & 0 \\
  a_{84} & 0 & a_{88} & 0 \\
  a_{94} & a_{95} & a_{99} & 0
\end{pmatrix}
\]

(3.1)

on the fine level. Let us also assume for simplicity that all diagonal elements are positive and all off-diagonal elements are negative. The treatment of nonsymmetric matrices with a mix of negative and positive elements on the diagonal can be found, for instance, in [55].

The coarse matrix can be obtained by aggregating pairs of nodes that are strongly coupled to each other. There are many formulations for “strength of connection,” which define exactly what “strongly coupled” means [55, 26, 20]. In this paper, we say node “$a_{ii}$ is strongly coupled to $a_{jj}$” if

\[
|a_{ij}| \geq \alpha \cdot \max_{k \neq i} |a_{ik}|
\]

(3.2)
for some scalar parameter $\alpha \in [0,1]$. Also, in this case we say that “$a_{jj}$ is the strong neighbor of $a_{ii}$.” Notice that if $\alpha = 1$, then $j$ corresponds to the index of the largest off-diagonal entry in absolute value in the $i$th row. Any ties between multiple strongly coupled nodes can be broken using small random perturbations or a simple heuristic, such as the proximity of the nodes in the sense of the distance $|i-j|.

Once we have selected the strongly coupled nodes, we aggregate them. We remove these nodes from consideration and repeat the process until there are no more nodes to be aggregated. We optionally merge any remaining singletons, that is, force each of them to join a strongly coupled neighbor aggregate. Any singletons that are only weakly coupled will be solved exactly on the fine level (they correspond to rows with a single element on the diagonal, or rows with only very small off-diagonal entries).

Aggregation can be performed in parallel using graph matching as outlined in Algorithm 1. It describes a straightforward one phase handshaking approach, which serves as a basis for a variety of other more complex schemes [16]. In this approach each node “extends a hand” to a strongly coupled neighbor, and if the neighbor reciprocates, the two nodes are aggregated. The process is repeated until there are no more nodes left to be aggregated. Assuming elements $a_{15}, a_{26}, a_{37}, a_{41}, a_{51}, a_{62}, a_{73}, a_{84}, a_{95}$ are the largest and $a_{48}, a_{94}$ are the second largest in the sense of (3.2), a hypothetical diagram of the progress of the algorithm on the adjacency graph corresponding to the coefficient matrix (3.1) can be seen in the three steps in Figures 2, 3, and 4, respectively.

\begin{algorithm}
\caption{Graph matching.}
1: Let $G(V,E)$ be the adjacency graph of the coefficient matrix $A$.
2: Let $S \subseteq V$ be a set of vertices with no connections and $W = V \setminus S$.
3: Let $P = [p_{ij}]$ be the elements of the prolongation matrix $P$.
4: for $k = 1, 2, \ldots$, until $W = \{\emptyset\}$ do
5: \hspace{1em} for $v \in W$ in parallel do \triangleright Setup Hand Shakes
6: \hspace{2em} Find $v$’s strongest neighbor $w = sn(v)$ following (3.2).
7: \hspace{1em} end for
8: \hspace{1em} for $v \in W$ in parallel do \triangleright Aggregate Matching Hand Shakes
9: \hspace{2em} if $sn(w) = v$ then
10: \hspace{3em} Form aggregate $i$ (set $P$ matrix elements $p_{vi} = p_{wi} = 1$).
11: \hspace{3em} Remove them from the next outer iteration, so that $W = W \setminus \{v, w\}$.
12: \hspace{2em} end if
13: \hspace{1em} end for
14: end for
\end{algorithm}

In our example, the resulting prolongation matrix $P$ can be written as

\[
P = \begin{pmatrix}
1 \\
1 \\
1 \\
1 \\
1 \\
1 \\
1 \\
1 \\
1
\end{pmatrix},
\]

while the restriction matrix $R$ is defined as $R = P^T$. 

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.
Finally, the matrix on the coarse level can be computed using the Galerkin product

\[ P^T A P = \begin{pmatrix} (a_{11} + a_{15} + a_{51} + a_{55}) & \cdots & a_{14} & a_{59} \\ \vdots & \ddots & \vdots & \vdots \\ a_{41} & \cdots & (a_{44} + a_{48} + a_{84} + a_{88}) & a_{49} \\ a_{95} & \cdots & a_{94} & a_{99} \end{pmatrix}. \]

In the AmgX implementation, the product (3.4) is computed implicitly, because matrix \( P \) has a very special form. As is shown above, it is a binary matrix with at most one nonzero value per row (zero rows corresponding to diagonal elements in the fine matrix are allowed). This allows for many optimizations, and custom kernels have been developed to speed up the computation. In the classical-based AMG the matrix \( P \) has a more complex form and therefore the Galerkin product requires special attention. Its implementation is the subject of the next section.

### 3.2. Sparse matrix multiplication—classical

In the classical-based AMG, one of the most time consuming operations in the setup phase is the computation of the Galerkin product

\[ C = P^T A P, \]
where we let the restriction matrix $R$ be defined as $R = P^T$. For clarity of presentation we will use an intermediate matrix $Z$ and compute the Galerkin product as two multiplications of a pair of matrices $Z = AP$ and $C = P^T Z$. It is possible to compute the product directly; however, this makes the efficient on-chip data reuse described in the following less effective due to many more intermediate results.

It is important to note that when the product is broken into two steps, the ordering of the steps, in other words, computing $(P^T A)P$ or $P^T (AP)$, is important. Even though the total number of nonzero elements in $P^T A$ and $AP$ is the same, in the former case we have fewer rows with a larger number of nonzero elements per row, while in the latter case we have a larger number of rows with fewer elements in them. This is an important distinction because we often use a hash table to store elements of $Z$, where each bucket is associated with a row and has a certain maximum number of elements in it [21]. Resizing the buckets is possible, but is a computationally expensive operation, and therefore limiting the number of elements per row often helps avoid it. For this reason, even though the two approaches are equivalent from the mathematical perspective, in our numerical experiments the latter results in a significantly shorter computation time.

Let us now focus our attention on the multiplication of a pair of two matrices, which has been the subject of many papers [29, 14]. Letting matrix $A = [a_{ij}]$ and matrices $Z = [z_i^T]$ and $P = [p_i^T]$ be split into their elements and rows, we can write for each row of the resulting matrix

\begin{equation}
    z_i^T = \sum_{j=1, a_{ij} \neq 0}^n a_{ij} p_j^T
\end{equation}

for $i = 1, \ldots, n$. An illustration of the sparse matrix-matrix multiplication $Z = AP$ is shown in Figure 5.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{sparse_matrix_multiplication.png}
\caption{Sparse matrix-matrix multiplication $Z = AP$ (color version online).}
\end{figure}

The multiplication is split into two phases. First, we count the number of nonzero elements per row in the resulting matrix $Z$. Then, we can allocate the space needed to store its column indices and values. Second, we find the column indices and compute the values of the result. The two steps are almost identical from the memory access perspective and differ only in the computation of the values at the end.

Let matrix $A$ be stored in the compressed sparse row format, that is, in arrays of row offsets $\mathbf{Ap}$, column indices $\mathbf{A1}$, and values $\mathbf{Av}$, and let $P$ and $Z$ be stored likewise. Then straightforward CUDA pseudo-code implementing these two steps is shown below.
__global__ csrgemm_count_kernel(int n, int *Ap, int *Ai, double *Av, int *Pp, int *Pi, double *Pv, int *Zp, int *Zi, double *Zv) {
    int row, col, j, k;

    //for each row of A in parallel
    for (row = threadIdx.z + blockIdx.z * blockDim.z; row < n; row += blockDim.z * gridDim.z) {
        //for each col of A in parallel
        for (j = Ap[row] + threadIdx.y + blockIdx.y * blockDim.y; j < Ap[row + 1]; j += blockDim.y * gridDim.y) {
            col = Ai[j]; //also, row of P
            //for each col of P in parallel
            for (k = Pp[col] + threadIdx.x + blockIdx.x * blockDim.x; k < Pp[col + 1]; k += blockDim.x * gridDim.x) {
                col_P = Pi[k]; //key
                hashTable[row].insert(col_P);
            }
        }
        Zp[row] = hashTable[row].size();
    }
}

__global__ csrgemm_compute_kernel(int n, int *Ap, int *Ai, double *Av, int *Pp, int *Pi, double *Pv, int *Zp, int *Zi, double *Zv) {
    int row, col, j, k; double val, val_P;

    //for each row of A in parallel
    for (row = threadIdx.z + blockIdx.z * blockDim.z; row < n; row += blockDim.z * gridDim.z) {
        //for each col of A in parallel
        for (j = Ap[row] + threadIdx.y + blockIdx.y * blockDim.y; j < Ap[row + 1]; j += blockDim.y * gridDim.y) {
            col = Ai[j]; //also, row of P
            val = Av[j];
            //for each col of P in parallel
            for (k = Pp[col] + threadIdx.x + blockIdx.x * blockDim.x; k < Pp[col + 1]; k += blockDim.x * gridDim.x) {
                col_P = Pi[k]; //key
                val_P = Pv[k] //value
                //perform union (eliminate duplicate keys)
                hashTable[row].insert_by_key(col_P, val * val_P);
            }
        }
    }
Notice that as discussed earlier, we are using a hash table on lines 20 and 47 in the above pseudo-code to merge the results for each of the rows in matrix $Z$. We are not showing the implementation of this data structure here, but rather point to a detailed description in [21].

In practice many optimizations can be added to the above routines. For example, we can take advantage of the \_popc, \_ballot, and \_any intrinsics widely used in the Fermi architecture to better control threading. We can improve memory access using Kepler \_ldg and \_shfl intrinsics. We note that these are essential to attain high performance on the GPU and are indeed used in the implementation. However, the code structure remains the same, so we avoid cluttering the pseudo-code with them.

Finally, notice that the above kernels use three-dimensional (3D) parallelism, with parallel threads organized in the $x$, $y$, and $z$-dimensions. We take advantage of the \_dim3 data structure to keep the kernel pseudo-code simple and organize the kernel launches in the following fashion:

$$\cdots$$
$$\text{dim3 threads}(tx,ty,tz);$$
$$\text{dim3 blocks}(bx,by,bz);$$
$$\text{//count non-zeroes per row}$$
$$\text{csrgemm\_count\_kernel}<<<\text{blocks,threads}>>>(\cdots);$$
$$\text{//allocate memory}$$
$$\cdots$$
$$\text{csrgemm\_compute\_kernel}<<<\text{blocks,threads}>>>(\cdots);$$
$$\cdots$$

Notice that essentially the only difference between a sequential and the CUDA pseudo-code illustrated in the kernels above are the highlighted keywords, such as \_global and threadIdx/threadDim and blockIdx/blockDim, which indicate that the code is a kernel and allow thread indexing in it. This illustrates the simplicity and versatility of the CUDA programming language.

### 3.3. ILU with zero fill-in—smoothers/preconditioners

Next we focus on the implementation of the ILU factorization, which is one of the most popular black-box smoothers/preconditioners. There are two distinct approaches to expose parallelism in the ILU factorization with zero fill-in. To illustrate them, let us consider the following coefficient matrix:

$$A = \begin{pmatrix} a_{11} & * & * & * & * & * & * & * \\ a_{22} & * & * & * & * & * & * & * \\ a_{33} & * & * & * & * & * & * & * \\ a_{44} & * & * & * & * & * & * & * \\ a_{55} & * & * & * & * & * & * & * \\ a_{66} & * & * & * & * & * & * & * \\ a_{77} & * & * & * & * & * & * & * \\ a_{88} & * & * & * & * & * & * & * \\ a_{99} & & & & & & & * \\ \end{pmatrix}$$

(3.7)
where \(*\) denotes an element that is either present or not in the matrix.

The first approach is often called “level-scheduling” and involves an implicit reordering of the linear system (1.1). In this approach we still solve the original system,

\[(3.8) \quad Ax = f,\]

but we find out which rows are independent, group them into levels, and process all the rows within a single level in parallel \([28, 42]\). In this setting the “levels” represent the data dependencies between groups of rows. Therefore, the next level can be processed only when the previous level has finished.

The directed acyclic graph (DAG) illustrating the data dependencies in the ILU factorization of the matrix in (3.7) is shown in Figure 6. Note that in practice we do not need to construct the data dependency DAG because it is implicit in the structure of the matrix. There is a dependency between node \(i\) and \(j\) for \(i > j\) if there exists an element \(a_{ij} \neq 0\) in the matrix. Therefore, the dependency graph can be easily traversed if, for example, the matrix is stored in compressed sparse column format.

The level-scheduling scheme can be viewed as an implicit reordering of the given matrix. This observation is important because it implies that the rate of convergence of iterative methods will not be affected when the level-scheduling scheme is used. It is important to note that physically shuffling the rows in memory, so that the rows belonging to the same level are next to each other, is irrelevant from the mathematical perspective because by construction the rows being shuffled are independent. From the practical point of view, however, such reshuffling can improve memory coalescing, and in our experiments we see up to 25% performance increase at the cost of extra memory required to perform the shuffle.

Also, notice that the analysis phase of the level-scheduling scheme that discovers parallelism for the ILU factorization and the lower triangular solve is actually the same. This implies that the obtained information can be reused for the incomplete factorization and the subsequent lower triangular solves, while the upper triangular solves require a separate analysis phase. This technique also allows us to parallelize algorithms, such as Gauss–Seidel and successive overrelaxation, which involve sparse triangular solves.

An outline of an analysis phase is shown in Algorithm 2. Notice that in this algorithm the node’s children are visited only if they have no data dependencies on the other nodes. The independent nodes are grouped into levels, which are shown with dashed lines in Figure 6. This information is passed to the numerical factorization.
phase, which can process the nodes belonging to the same level in parallel. Finally, an outline of the numerical factorization phase is shown in Algorithm 3.

Algorithm 2. Symbolic analysis phase.
1: Let \( n \) and \( e \) be the matrix size and level number, respectively.
2: \( e \leftarrow 1 \)
3: repeat ▷ Traverse the Matrix and Find the Levels
4: for \( i \leftarrow 1, n \) do ▷ Find Root Nodes
5: if \( i \) has no data dependencies then
6: Add node \( i \) to the list of root nodes.
7: end if
8: end for
9: for \( i \in \) the list of root nodes do ▷ Process Root Nodes
10: Add node \( i \) to the list of nodes on level \( e \).
11: Remove the data dependency on \( i \) from all other nodes.
12: end for
13: \( e \leftarrow e + 1 \)
14: until all nodes have been processed.

1: Let \( k \) be the number of levels.
2: for \( e \leftarrow 1, k \) do
3: \( \text{list} \leftarrow \) the sorted list of rows in level \( e \).
4: for \( \text{row} \in \text{list} \) in parallel do ▷ Process a Single Level
5: Update elements in the \( \text{row} \).
6: end for
7: Synchronize threads. ▷ Synchronize between Levels
8: end for

The second approach to ILU factorization relies on the graph coloring of the adjacency graph of the coefficient matrix of the linear system at hand [9, 10, 43, 47] and involves an explicit reordering \( Q \), so that we solve the reordered linear system

\[
(Q^T AQ)(Q^T x) = Q^T f.
\]

The main objective of graph coloring is to assign a color to every node in a graph such that no two nodes have the same color and at the same time use as few colors as possible. In practice we often use integers to represent different colors. Suppose that the sparsity pattern of (3.7) is symmetric; if it is not, we can always apply the same algorithm on the graph induced by \( A + A^T \). A sample graph coloring of the adjacency graph of coefficient matrix (3.7) is shown in Figure 7.

Finding a graph coloring that uses the minimum number of colors for a general graph is an NP-hard problem [17]. However, there are many algorithms that can produce heuristic-based colorings that are a good enough approximation of the minimum coloring in a reasonable amount of time [34, 27]. From the preconditioner perspective, fewer colors mean more parallelism, while more colors often imply stronger coupling between the steps. It usually does not hurt the preconditioner to have a few more colors than the theoretical minimum. A sample parallel approximate scheme is illus-
trated in Algorithms 4 and 5, which find an independent set and use it to produce a valid coloring of a graph, respectively.

It is important to point out that there exist many recoloring techniques that improve an existing approximate coloring [33]. These often decrease the number of colors and therefore increase the degree of available parallelism for a given problem. In our preliminary experiments we have seen a speedup of up to 2$x$ resulting from these techniques.

**Algorithm 4.** Independent set.

1. Let $G(V,E)$ be the adjacency graph of the coefficient matrix $A$.
2. Let $S = \{\emptyset\}$ be the independent set.
3. Assign a random number $r(v)$ to each vertex $v \in V$.
4. for $v \in V$ in parallel do \hspace{1cm} $\triangleright$ Find Local Maximum
5. \hspace{1cm} if $r(v) > r(w)$ for all neighbors $w$ of $v$ then
6. \hspace{1cm} \hspace{1cm} Add vertex $v$ to the independent set $S$.
7. \hspace{1cm} end if
8. end for

**Algorithm 5.** Graph coloring.

1. Let $G(V,E)$ be the adjacency graph of the coefficient matrix $A$.
2. Let set of vertices $W = V$.
3. for $k = 1, 2, \ldots$ until $W = \{\emptyset\}$ do \hspace{1cm} $\triangleright$ Color an Independent Set Per Iteration
4. \hspace{1cm} Find in parallel an independent set $S$ of $W$.
5. \hspace{1cm} Assign color $k$ to vertices in $S$.
6. \hspace{1cm} Remove vertices in set $S$ from $W$, so that $W = W \setminus S$
7. end for

Once we have computed a graph coloring, we can use it to compute a permutation such that nodes of the same color are adjacent to one another. In our example, this
permutation \( q^T = [1, 2, 3, 8, 9, 4, 5, 6, 7] \) corresponds to the reordering matrix

\[
Q^T = \begin{pmatrix}
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1
\end{pmatrix}
\]

which results in the reordered coefficient matrix

\[
Q^T AQ = \begin{pmatrix}
a_{11} & a_{14} & a_{15} & a_{16} \\
a_{22} & a_{26} & a_{28} & a_{29} \\
a_{33} & a_{37} & a_{38} & a_{39} \\
a_{44} & a_{45} & a_{46} & a_{47} \\
a_{55} & a_{56} & a_{57} & a_{59} \\
a_{66} & a_{67} & a_{68} & a_{69} \\
a_{77} & a_{78} & a_{79} & a_{79}
\end{pmatrix}
\]

Notice that this matrix has diagonal blocks, which expose the parallelism available in the factorization and subsequent lower and upper triangular solves. The matrix updates during the factorization can now be performed trivially using diagonal scaling, matrix addition, and multiplication. Also, notice that if we look at the reordered matrix \( Q^T AQ \) from the level-scheduling perspective, its data dependency DAG has wider and fewer levels, as shown in Figure 8. This implies that by using an explicit reordering we have extracted more parallelism than what was available in the original linear system.

![Fig. 8. The data dependency DAG of the reordered matrix \( Q^T AQ \).](image-url)

In general, in order to attain the shortest total time, there is a tradeoff between a longer analysis phase that finds a more accurate coloring and a shorter numerical factorization and triangular solve phases that take advantage of the uncovered parallelism. A single analysis is often amortized across multiple solve calls. Therefore, this tradeoff often depends on the number of steps taken by the iterative method for a given problem. It is important to keep this in mind when deciding how much effort should be spent in the analysis phase.
Finally, a good question to address is how the explicit reordering $Q$ affects the convergence of the iterative method. Notice that the reordering matrix $Q$ is an orthogonal matrix ($Q^T Q = QQ^T = I$); therefore the reordered linear system (3.9) has been obtained from the original linear system (3.8) using an orthogonal transformation. Since orthogonal transformation does not change the eigenvalues of a matrix, which govern the convergence of iterative methods, the convergence of the unpreconditioned iterative methods is not affected [32].

However, when preconditioning such as ILU factorization is used, the convergence of the preconditioned iterative methods is affected. It is clear that, depending on the reordering, different fill-in entries of the matrix will be dropped during the incomplete factorization, resulting in a better or worse preconditioner. We are not aware of any general theoretical results about the effects of reordering on convergence, but there are many empirical studies, some of which are listed in [24, 23, 22, 8]. In our experiments, convergence was usually negatively affected by the graph coloring reordering, but this impact was not significant enough to offset gains obtained through the extra parallelism attained by coloring.

Lastly, it is important to point out that physically shuffling the rows in memory, so that the rows belonging to the same color are next to each other, is irrelevant from the mathematical perspective because we have already potentially changed the dependencies between them using coloring. However, just as with level-scheduling, from the practical point of view such reshuffling can improve memory coalescing at the cost of some extra memory required to perform the shuffle.

4. Parallel algorithms across multiple nodes (multiple GPUs). An additional layer of complexity is introduced when we combine GPU acceleration with distributed processing. Now we must prepare the problem by partitioning and packing it so that multiple GPUs across distributed nodes can effectively solve it.

The partitioning should satisfy two requirements: (i) equally balanced partition sizes, and (ii) minimal “graph edge cuts” between partitions. We attempt to satisfy these by applying common graph partitioning utilities, such as Metis [36] or Scotch [15]. As a result we obtain a map that assigns each degree of freedom in the problem to a particular partition. We use this map to assign each row of the matrix and element of the vector to its partition. A trivial partitioning, where rows are assigned contiguously, is illustrated on the left side of Figure 9.

![Fig. 9. A sample graph partitioning and packing of matrix A (color version online).](image)

Once we have partitioned the matrix, we pack the data on each partition. We renumber the column indices, such that (i) we move the diagonal blocks to the left, (ii) we perform a stable sort of the remaining column indices based on the enumeration of the partition to which they belong, and (iii) we shift them so that there are no gaps between consecutive indices. Notice that this often allows us to switch from
64-bit integer indices used for the global problem to 32-bit integer indexing for the local problem. This renumbering also implies an ordering on the vector elements, such that the local ones come first and the halo elements that need to be fetched from other partitions, following the ordering defined by (ii) and (iii), come last.

Finally, we classify the matrix rows into interior, boundary, and halo rows. The interior rows correspond to rows that are not connected to other partitions; in other words, they do not have any column indices outside of the diagonal block. On the other hand, the boundary rows correspond to rows that are connected to other partitions. Lastly, the halo rows of a particular partition are rows from all the other partitions to which the current partition boundary is connected (through column indices outside of the diagonal block). The vector elements are classified in the same fashion, where we let the interior and boundary elements be called local vector elements.

The result of packing for the matrix and vector is shown on the right side of Figure 9, where for the matrix the diagonal blocks and the remaining columns are indicated by the solid and dashed lines, respectively, while for the vector the local and halo elements are indicated by the solid and dashed lines, respectively.

4.1. Sparse matrix-vector multiplication. Let us now discuss two approaches to distributed sparse matrix-vector multiplication, which is one of the most common patterns of computation in iterative methods and AMG.

First, to compute the result we can simply start the exchange of the halo of the vector, then wait for it to arrive, and perform a local matrix-vector multiplication. We can also add “latency-hiding” to this approach by attempting to overlap some of the communication time with computation work. Notice that we have classified our local rows into interior and boundary rows. Therefore, we can start the exchange of the halo of the vector, then, while it is in progress, at the same time start the computation of the interior elements (which are independent from the vector halo by construction), and once both of these tasks are complete we can perform the remaining computation for the boundary elements (which does require the vector halo elements). Therefore, the multiplication is split into interior and boundary steps, as illustrated in Figure 10 for the first partition from Figure 9. This technique can be very effective in hiding communication time when there is enough computation to be done for the interior elements.

Second, we can perform multiple iterations without communication if we exchange larger halos of the vector and the matrix. For example, we may formulate our problem as applying powers of a matrix to a vector \( y = A^k x \), in a fashion similar to that of communication-avoiding Krylov subspace iterative methods [31]. Letting \( k = 2 \) for
the sake of simplicity, if we exchange matrix halo rows and the second degree halo (halo of the halo) of the vector, then we can compute $y = A^2 x$ without additional communication. The approach is illustrated in Figure 11. In this case, in effect we have larger communication in the setup and no communication during the actual computation. Although we have not experimented extensively with this approach, the preliminary results were, unfortunately, not promising, because in our cases the sparsity pattern was such that the overhead of exchanging matrix halos was too large to be competitive.

5. **Numerical experiments.** In this section we study the performance and scalability of the AmgX library on several test problems as a stand-alone package and as part of the ANSYS Fluent 16.0 computational fluid dynamics software package. Some of these problems were obtained from public sources [18], while others were obtained from existing national laboratory [61] and industry benchmarks [5]. In all experiments, we used AmgX 1.2 and CUDA Toolkit CUSPARSE and CUBLAS 5.5 libraries [46].

5.1. As a stand-alone library (on a single node). When used as a stand-alone solver, the AmgX library can be used for a wide variety of abstract problems formulated as linear equations, so long as the equations have some elliptic character, that is, are consistent discretizations of a PDE of elliptic type. In this section we choose several matrices that satisfy this condition from The University of Florida Sparse Matrix Collection [18]. These matrices and their brief descriptions are listed in Table 1.

<table>
<thead>
<tr>
<th>#</th>
<th>Matrix</th>
<th>m,n</th>
<th>nnz</th>
<th>spd</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2cubes-sphere</td>
<td>101,492</td>
<td>1,647,264</td>
<td>yes</td>
<td>electromagnetics</td>
</tr>
<tr>
<td>2</td>
<td>thermal2</td>
<td>1,228,045</td>
<td>8,580,313</td>
<td></td>
<td>thermal simulation</td>
</tr>
<tr>
<td>3</td>
<td>atmosmodd</td>
<td>1,270,432</td>
<td>8,814,880</td>
<td>no</td>
<td>atmospheric model.</td>
</tr>
<tr>
<td>4</td>
<td>atmosmodl</td>
<td>1,489,752</td>
<td>10,319,760</td>
<td>no</td>
<td>atmospheric model.</td>
</tr>
<tr>
<td>5</td>
<td>cage14</td>
<td>1,505,785</td>
<td>27,130,349</td>
<td>yes</td>
<td>biology</td>
</tr>
</tbody>
</table>

First, we compare the performance of the AmgX 1.2 and HYPRE 2.9.0b libraries on a single node. We use a workstation with some of the latest hardware: Intel Xeon E5-2690v2 10-core at 3.0 GHz CPU and Nvidia Tesla K40 GPU. It has CentOS 6.4 OS with GCC 4.6.4 and ICC 14.0.2 compilers.
We use common configuration options for both libraries. In the setup phase we select the classical path V-cycle with aggressive PMIS with multipass interpolation on the first level and standard PMIS with D2 interpolation on the subsequent levels. In the solve phase we use a Jacobi smoother with two pre-, post-, and coarsest-sweeps. Please refer to section 2, the AMG introductory section, for a description of these standard parameters. We let the stopping criteria be based on the maximum number of iterations being less than 300 and relative residual $\|r_i\|_2/\|r_0\|_2 \leq 10^{-8}$, where $r_i = f - Ax_i$ is the residual at the $i$th iteration.

We compare the performance of the AmgX library compiled with GCC to that of the HYPRE library compiled with ICC and GCC compilers. We use the latter as the baseline and plot the attained speedup based on the total (setup + solve) time in Figure 12. First, notice that in our experiments using ICC for HYPRE brought only modest speedups of no more than 10%. Therefore, we will use the more common and freely available GCC compiler from now on. Second, notice that in our experiments AmgX indeed often obtained speedup of $2-5 \times$, with detailed results shown in Table 2.

### Table 2

**Performance of the AmgX and HYPRE libraries on a single node.**

<table>
<thead>
<tr>
<th></th>
<th>AmgX</th>
<th>HYPRE (ICC)</th>
<th>HYPRE (GCC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>Setup</td>
<td>Solve</td>
<td>Setup</td>
</tr>
<tr>
<td>1</td>
<td>0.027</td>
<td>0.035</td>
<td>0.026</td>
</tr>
<tr>
<td>2</td>
<td>0.209</td>
<td>0.234</td>
<td>0.727</td>
</tr>
<tr>
<td>3</td>
<td>0.309</td>
<td>0.279</td>
<td>0.324</td>
</tr>
<tr>
<td>4</td>
<td>0.333</td>
<td>0.322</td>
<td>0.835</td>
</tr>
<tr>
<td>5</td>
<td>0.053</td>
<td>0.118</td>
<td>0.122</td>
</tr>
</tbody>
</table>

### 5.2. As a stand-alone library (on multiple nodes). Second, we compare the performance of AmgX 1.2 and the AMG2013 CORAL benchmark [62], which has been derived directly from the HYPRE 2.9.0b library. We perform the experiments on the Titan supercomputer from the Oak Ridge National Laboratory. This machine has thousands of interconnected hardware nodes with AMD Opteron 6274 16-core at 2.2GHz CPU and Nvidia Tesla K20x GPU (one per node). It uses Cray Linux Environment OS and GCC 4.8.2 compiler.

In order to study weak scaling performance we use the parameters suggested by the CORAL benchmark. As the input matrix we generate a 3D (7 point) discretization of the Poisson equation in a cube domain of dimensions $200 \times 200 \times 200$ (200$^3$) per node and replicate it in the x-, y-, and z-dimensions to cover the selected number of
nodes. As the solver parameters, in the setup phase we use V-cycle with PMIS and D2 interpolation operator in the classical (CLA) path and size 8 aggregates in the aggregation (AGG) path, respectively. In the solve phase we use a Jacobi smoother with two pre-, post-, and coarsest-sweeps. We also use an outer CG solver to speed up convergence. Please refer to section 2 for a description of these standard parameters. We let the stopping criteria be based on the maximum number of iterations being less than 300 and relative residual $||r_i||_2/||r_0||_2 \leq 10^{-8}$.

It is important to point out that in the CORAL benchmark each rank generates its own part of the global matrix corresponding to a local cube domain. Furthermore, since the AmgX library expects one rank to be assigned per GPU (and there is one GPU per node on the machine), for AmgX experiments we will always run a single rank per node, each with $200^3$ cube domain. For the AMG2013 (HYPRE) runs we will experiment with two configurations. The first will mirror the AmgX experiments, by running with a single rank per node, each with $200^3$ cube domain. In this case, we will effectively compare the performance of running with a single rank per node with AmgX utilizing a single GPU and AMG2013 (HYPRE) utilizing a single core. It’s not surprising therefore that in this setting we see speedups of 20–40× with AmgX. The second experiment will effectively partition the $200^3$ cube domain into eight $100^3$ cube domains, and run eight ranks per node, each with $100^3$ cube domain. In this case we will still be solving the same global problem, but we will be comparing AmgX utilizing a single GPU and AMG2013 (HYPRE) utilizing eight cores on a node. Notice that the hardware architecture of a node is such that each two cores share a floating point unit (FPU) and L2 cache; therefore it is fair to launch eight ranks per node, effectively having one rank per FPU/L2 and therefore utilizing the resources of the entire node. In this more challenging setting we see speedups in the range of 2–5× with AmgX.

In order to better understand where the time is spent in the algorithm, we plot the scalability of the setup time, solve time (per iteration), and total time in Figures 13–15. Recall that in weak scaling we increase the problem size and the number of nodes, so that the problem size per node stays constant, and therefore we hope that the time line remains flat.

Notice that in the aggregation (AGG) path the setup phase scales extremely well. This is a reflection of the fact that in its implementation we do not create aggregates across partitions, therefore making the setup a local process. The time per iteration also stays relatively stable, reflecting efficient implementation of the distributed sparse matrix-vector multiplication. However, the number of iterations grows as the number of nodes increases. This happens because the local setup becomes less effective as the global problem size increases. Therefore overall total time shown in Figure 15 is not perfectly flat. In the future we plan to address this issue by investigating the quality of the aggregates created in the setup phase.

In the classical (CLA) path the situation is reversed. The solve phase of the algorithm performs well, with the solve time per iteration being similar to that of the aggregation path and the growth in number of iterations being significantly lower; see Figure 14. This reflects a better setup phase which incorporates global information when creating the coarse matrices. Unfortunately, this also leads to the setup phase time line not being as flat as we would like, as shown in Figure 13. This reflects the complexity of computing distributed Galerkin product $C = RAP$. In the future work we will consider using different data structures for holding local matrices (for example, using a separate square and a leftover matrix) to facilitate this computation.

An interesting observation is that the growth in the solve time per iteration seems to be closely related to the log(# of nodes), which is often proportional to the cost of
Fig. 13. Weak scaling results: AmgX versus HYPRE setup time on hundreds of GPUs (color version online).

Fig. 14. Weak scaling results: AmgX versus HYPRE solve time per iteration on hundreds of GPUs (color version online).

Fig. 15. Weak scaling results: AmgX versus HYPRE total time to solution on hundreds of GPUs (color version online).
distributed reduction operations. In fact we can easily fit a line through the obtained results as shown in Figure 16. In the future we plan to take a closer look at how to mitigate the effect of such operations.

Finally, we illustrate the effect of adding an outer solver, such as GMRES or CG, for this particular problem. We plot the number of iterations taken by the stand-alone AMG, restarted GMRES with restart 10 (GMRES-AMG), and CG preconditioned by AMG (CG-AMG). As shown in Figures 17 and 18, adding an outer solver significantly reduces the number of iterations taken to convergence, without significantly increasing the time taken per iteration. This is in line with our own practical experience that indicates that adding an outer solver is often beneficial to speed up the computation. Although we only report experiments with AmgX, similar benefits were observed with the HYPRE library as well. In fact the AMG2013 benchmark does indeed use an outer CG to speed up the computation.

Finally, the detailed results of the experiments are summarized in Tables 3–5.

5.3. In real-world computational fluid dynamics (CFD). The AmgX library is used as part of the major commercial CFD software package ANSYS Fluent 16.0. It provides acceleration for the solution of large sparse linear systems in shared and distributed settings on multiple GPUs.
Fig. 18. AmgX (CLA) solve time per iteration, with and without an outer solver (color version online).

Table 3
Weak scaling of AMG2013 CORAL benchmark (based on HYPRE).

<table>
<thead>
<tr>
<th># of nodes</th>
<th>Setup time(s)</th>
<th>Solve time(s)</th>
<th>Total time(s)</th>
<th># it.</th>
<th>Setup time(s)</th>
<th>Solve time(s)</th>
<th>Total time(s)</th>
<th># it.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>42.49</td>
<td>43.17</td>
<td>85.67</td>
<td>12</td>
<td>7.011</td>
<td>8.500</td>
<td>15.51</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>62.04</td>
<td>58.85</td>
<td>120.8</td>
<td>13</td>
<td>7.621</td>
<td>9.786</td>
<td>17.40</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>78.60</td>
<td>77.80</td>
<td>156.4</td>
<td>14</td>
<td>8.238</td>
<td>9.815</td>
<td>18.05</td>
<td>15</td>
</tr>
<tr>
<td>8</td>
<td>94.29</td>
<td>83.72</td>
<td>178.0</td>
<td>15</td>
<td>8.805</td>
<td>10.50</td>
<td>19.31</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>110.5</td>
<td>89.37</td>
<td>199.9</td>
<td>16</td>
<td>8.872</td>
<td>11.75</td>
<td>20.63</td>
<td>18</td>
</tr>
<tr>
<td>32</td>
<td>124.5</td>
<td>100.4</td>
<td>224.9</td>
<td>18</td>
<td>9.035</td>
<td>12.39</td>
<td>21.43</td>
<td>19</td>
</tr>
<tr>
<td>64</td>
<td>140.2</td>
<td>105.9</td>
<td>246.2</td>
<td>19</td>
<td>9.264</td>
<td>13.07</td>
<td>22.33</td>
<td>20</td>
</tr>
<tr>
<td>128</td>
<td>139.9</td>
<td>116.7</td>
<td>256.7</td>
<td>21</td>
<td>9.414</td>
<td>13.78</td>
<td>23.19</td>
<td>21</td>
</tr>
<tr>
<td>256</td>
<td>141.2</td>
<td>127.4</td>
<td>268.7</td>
<td>23</td>
<td>9.621</td>
<td>15.31</td>
<td>24.93</td>
<td>23</td>
</tr>
<tr>
<td>512</td>
<td>141.9</td>
<td>148.8</td>
<td>290.7</td>
<td>27</td>
<td>10.11</td>
<td>15.72</td>
<td>25.84</td>
<td>24</td>
</tr>
</tbody>
</table>

Table 4
Weak scaling of AmgX (classical and aggregation AMG with an outer CG).

<table>
<thead>
<tr>
<th># of nodes</th>
<th>Setup time(s)</th>
<th>Solve time(s)</th>
<th>Total time(s)</th>
<th># it.</th>
<th>Setup time(s)</th>
<th>Solve time(s)</th>
<th>Total time(s)</th>
<th># it.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.480</td>
<td>2.234</td>
<td>2.714</td>
<td>39</td>
<td>1.786</td>
<td>2.102</td>
<td>3.888</td>
<td>27</td>
</tr>
<tr>
<td>2</td>
<td>0.564</td>
<td>2.707</td>
<td>3.271</td>
<td>42</td>
<td>1.931</td>
<td>2.306</td>
<td>4.237</td>
<td>27</td>
</tr>
<tr>
<td>4</td>
<td>0.617</td>
<td>3.236</td>
<td>3.854</td>
<td>48</td>
<td>2.013</td>
<td>2.483</td>
<td>4.497</td>
<td>28</td>
</tr>
<tr>
<td>8</td>
<td>0.622</td>
<td>3.804</td>
<td>4.427</td>
<td>55</td>
<td>2.251</td>
<td>2.666</td>
<td>4.918</td>
<td>30</td>
</tr>
<tr>
<td>16</td>
<td>0.630</td>
<td>4.231</td>
<td>4.862</td>
<td>60</td>
<td>2.609</td>
<td>2.988</td>
<td>5.597</td>
<td>31</td>
</tr>
<tr>
<td>32</td>
<td>0.634</td>
<td>4.711</td>
<td>5.346</td>
<td>66</td>
<td>3.051</td>
<td>3.367</td>
<td>6.418</td>
<td>33</td>
</tr>
<tr>
<td>64</td>
<td>0.637</td>
<td>5.771</td>
<td>6.408</td>
<td>78</td>
<td>3.908</td>
<td>3.883</td>
<td>7.792</td>
<td>36</td>
</tr>
<tr>
<td>128</td>
<td>0.633</td>
<td>6.460</td>
<td>7.093</td>
<td>84</td>
<td>5.084</td>
<td>4.622</td>
<td>9.707</td>
<td>37</td>
</tr>
<tr>
<td>256</td>
<td>0.635</td>
<td>7.638</td>
<td>8.273</td>
<td>98</td>
<td>7.600</td>
<td>5.608</td>
<td>13.20</td>
<td>40</td>
</tr>
<tr>
<td>512</td>
<td>0.635</td>
<td>10.00</td>
<td>10.63</td>
<td>125</td>
<td>12.68</td>
<td>6.780</td>
<td>19.46</td>
<td>44</td>
</tr>
</tbody>
</table>

The linear systems arising in ANSYS Fluent simulations come from complex geometries, with several different modelling and meshing techniques. We showcase the performance of the AmgX library on two industrial benchmarks that model internal and external flow. We compare our performance against the commercial proprietary
Table 5
Weak scaling of AmgX (classical AMG with and without an outer GMRES).

<table>
<thead>
<tr>
<th># of nodes</th>
<th>Classical (AMG)</th>
<th>Classical (GMRES-AMG)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>200^3 cube/rank, 1 rank/GPU/node</td>
<td>200^3 cube/node, 1 rank/GPU/node</td>
</tr>
<tr>
<td></td>
<td>Setup time(s)</td>
<td>Solve time(s)</td>
</tr>
<tr>
<td>1</td>
<td>1.899</td>
<td>8.882</td>
</tr>
<tr>
<td>2</td>
<td>1.843</td>
<td>9.863</td>
</tr>
<tr>
<td>4</td>
<td>2.027</td>
<td>11.54</td>
</tr>
<tr>
<td>8</td>
<td>2.243</td>
<td>13.42</td>
</tr>
<tr>
<td>16</td>
<td>2.540</td>
<td>14.66</td>
</tr>
<tr>
<td>64</td>
<td>3.897</td>
<td>21.51</td>
</tr>
<tr>
<td>128</td>
<td>4.931</td>
<td>26.50</td>
</tr>
<tr>
<td>256</td>
<td>7.164</td>
<td>29.29</td>
</tr>
<tr>
<td>512</td>
<td>12.48</td>
<td>41.89</td>
</tr>
</tbody>
</table>

Fig. 19. Real world: AmgX performance on internal flow model (color version online).

AMG solver inside ANSYS Fluent. We note that the latter benchmark gives rise to very large sparse matrices with approximately 450 million unknowns.

The experiments are performed on a Nvidia internal cluster. The internal flow model ran on a single node with dual socket Intel Xeon E5-2687W 8-core at 3.1 GHz CPU and two Tesla K40 GPUs. The external flow model ran on 12 nodes, each with dual socket Intel Xeon E5-2667 6-core at 2.9 GHz and four Tesla K40 GPUs. In both cases CentOS 6.4 OS with GCC 4.6.4 compiler was used. Since we always need the problem to fit into the aggregate memory of the GPUs, in the former case we use two GPUs, and in the latter case we use all four GPUs on each of the available 12 nodes to solve the problem.

Notice that in both of our numerical experiments we obtain a speedup of 2–4× for the AmgX solver and 2–3× for the total simulation time; see Figures 19 and 20. We note that the acceleration impact on the total simulation time is more significant on problems with relatively high percentage of time spent in the linear system solver.
6. Conclusion. The AmgX provides GPU acceleration in a flexible, easy to use C API library. It combines modern AMG methods, iterative schemes, and a variety of preconditioners and smoothers with massively parallel algorithms on the GPU. The library works on shared and distributed computing platforms. It has shown promising stand-alone results and has performed well at scale as part of real-world CFD software packages with all of their complexity. Although the performance is dependent on the particular problem at hand, in our numerical experiments we have often attained acceleration in the range of $2-5\times$ for the same quality numerical results versus modern multicore CPU AMG implementation.

In the future, we plan to expand the set of algorithms provided by the library and improve its performance. Different eigenvalue solvers [6] and block-iterative methods [41] are some of the algorithms under consideration. Also, we plan to work on hybrid algorithms that exploit both CPU and GPU at the same time.

Acknowledgments. The authors would like to acknowledge Felix Abecassis, Alexandre Fender, Christoph Franke, Rohit Gupta, Justin Luitjens, Francesco Rossi, Konstantin Stulov, Lu Wang, Zhenhai Zhu, and many others who have contributed to the development of the AmgX library.

REFERENCES
ALGEBRAIC MULTIGRID AND ITERATIVE METHODS ON GPU


