SIAM J. SCI. COMPUT.
Vol. 34, No. 4, pp. 123–152

EXPOSING FINE-GRAINED PARALLELISM IN ALGEBRAIC MULTIGRID METHODS[*]

NATHAN BELL[†], STEVEN DALTON[‡], AND LUKE N. OLSON[‡]

Abstract. Algebraic multigrid methods for large, sparse linear systems are a necessity in many computational simulations, yet parallel algorithms for such solvers are generally decomposed into coarse-grained tasks suitable for distributed computers with traditional processing cores. However, accelerating multigrid methods on massively parallel throughput-oriented processors, such as graphics processing units, demands algorithms with abundant *fine-grained* parallelism. In this paper, we develop a parallel algebraic multigrid method which exposes substantial fine-grained parallelism in both the construction of the multigrid hierarchy as well as the cycling or solve stage. Our algorithms are expressed in terms of scalable parallel primitives that are efficiently implemented on the GPU. The resulting solver achieves an average speedup of 1.8× in the setup phase and 5.7× in the cycling phase when compared to a representative CPU implementation.

Key words. algebraic multigrid, parallel, sparse, graphics processing units, iterative

AMS subject classifications. 65-04, 68-04, 65F08, 65F50, 68W10

DOI. 10.1137/110838844

1. Introduction.

Throughput-oriented processors, such as graphics processing units (GPUs), are becoming an integral part of many high-performance computing systems. In contrast to traditional CPU architectures, which are optimized for completing scalar tasks with minimal latency, modern GPUs are tailored for parallel workloads that emphasize total task throughput [16]. Therefore, harnessing the computational resources of such processors requires programmers to decompose algorithms into thousands or tens of thousands of separate, fine-grained threads of execution. Unfortunately, the parallelism exposed by previous approaches to the algebraic multigrid (AMG) is too coarse-grained for direct implementation on GPUs.

AMG methods solve large, sparse linear systems $Ax = b$ by constructing a hierarchy of grid levels directly from the matrix A. In this paper, we study the components that comprise the two distinct phases in AMG—the setup and solve phases—and demonstrate how they can be decomposed into scalable parallel primitives.

Parallel approaches to multigrid are plentiful. AMG methods have been successfully parallelized on distributed-memory CPU clusters using MPI [12, 10] and more recently with a combination of MPI and OpenMP [2], to better utilize multicore CPU nodes. While such techniques have demonstrated scalability to large numbers of processors, they are not immediately applicable to the GPU. In particular, effective use of GPUs requires substantial *fine-grained* parallelism at all stages of the computation. In contrast, the parallelism exposed by existing methods for distributed-memory clusters of traditional cores is comparably coarse-grained and cannot be scaled down

[*]Submitted to the journal's Software and High-Performance Computing section June 28, 2011; accepted for publication (in revised form) April 3, 2012; published electronically July 5, 2012.
http://www.siam.org/journals/sisc/34-4/83884.html

[†]NVIDIA Research, 2701 San Tomas Expressway, Santa Clara, CA 95050 (nbell@nvidia.com, http://www.wnbell.com).
[‡]Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801 (dalton6@illinois.edu, lukeo@illinois.edu, http://www.cs.illinois.edu/homes/lukeo).

to arbitrarily small subdomains. Indeed, coarse-grained parallelization strategies are qualitatively different from fine-grained strategies.

For example, it is possible to construct a successful parallel coarse-grid selection algorithm by partitioning a large problem into subdomains and applying an effective, serial heuristic to select coarse-grid nodes on the interior of each subdomain, followed by a less-effective but parallel heuristic to the interfaces between subdomains [22]. An implicit assumption in this strategy is that the interiors of the partitions (collectively) contain the vast majority of the entire domain; otherwise the serial heuristic has little impact on the output. Although this method is scalable to arbitrarily fine-grained parallelism in principle, the result is qualitatively different. In contrast, the methods we develop do not rely on partitioning and expose parallelism to the finest granularity—i.e., one thread per matrix row or one thread per nonzero entry.

Geometric multigrid methods were the first to be parallelized on GPUs [19, 9, 34]. These "GPGPU" approaches, which preceded the introduction of the CUDA and OpenCL programming interfaces, programmed the GPU through existing graphics application programming interfaces such as OpenGL and Direct3d. Subsequent works demonstrated GPU-accelerated geometric multigrid for image manipulation [25] and CFD [13] problems. Previous works have implemented the cycling stage of the AMG on GPUs [18, 21]; however, hierarchy construction remained on the CPU. A parallel aggregation scheme is described in [35] that is similar to ours based on maximal independent sets, while in [1] the effectiveness of parallel smoothers based on sparse matrix-vector products is demonstrated. Although these works were implemented for distributed CPU clusters, they are amenable to fine-grained parallelism as well.

In section 1.2, we review the components of the setup and solve phases of AMG, noting the principal challenge in targeting GPU acceleration. Our approach to the GPU is to describe the components of multigrid in terms of parallel primitives, which we define in section 2. In section 3 we detail our specific approach to exposing fine-grained parallelism in the components of the setup phase, and in section 4 we highlight the value of the sparse matrix-vector product in the computation. In section 5 we discuss several performance results of our method on the GPU in comparison to the efficiency we observe on a standard CPU. Finally, in the appendix we provide additional details of a parallel aggregation method.

1.1. Background.

Multigrid methods precondition large, sparse linear systems of equations and in recent years have become a robust approach for a wide range of problems. One reason for this increase in utility is the trend toward more *algebraic* approaches. In the classical, *geometric* form of multigrid, the performance relies largely on specialized smoothers and a hierarchy of grids and interpolation operators that are predefined through the geometry and physics of the problem. In contrast, AMG methods attempt to automatically construct a hierarchy of grids and intergrid transfer operators without explicit knowledge of the underlying problem—i.e., directly from the linear system of equations [32, 37].

In the remainder of this section, we outline the basic components of AMG in an *aggregation* context [37] and highlight the necessary sparse matrix computations used in the process. We restrict our attention to aggregation methods because of the flexibility in the construction; however, our development also extends to classical AMG methods based on coarse-fine splittings [32].

1.2. Components of AMG.

The performance of AMG relies on a compatible collection of relaxation operators, coarse grid operators, and interpolation operators as well as the efficient construction of these operations. In this section we outline the components of aggregation-based AMG that we consider for construction on the GPU.

Aggregation-based AMG requires an a priori knowledge or prediction of the near-nullspace that represent the low-energy error. For an $n \times n$ symmetric, positive-definite matrix problem $Ax = b$, these m modes are denoted by the $n \times m$ column matrix B. Generally, the number of near-nullspace modes, m, is a small, problem-dependent constant. For example, the scalar Poisson problem requires only a single near-nullspace mode, while six rigid body modes are needed to solve three-dimensional elasticity problems. We also denote the $n \times n$ problem as the *fine* level and label the indices $\Omega_0 = \{0, \dots, n-1\}$ as the fine *grid*. From A, b, and B, the components of the solver are defined through a setup phase and include grids Ω_k, interpolation operators P_k, restriction operators R_k, relaxation error propagation operators, and coarse representations of the matrix operator A_k, all for each level k. We denote index M as the maximum level—e.g., $M = 1$ is a two-grid method.

1.2.1. Setup phase. We follow a setup phase that is outlined in Algorithm 1. The setup assumes input of a sparse matrix, A, and user-supplied vectors, B, which may represent low eigenmodes of the problem; here, we consider the constant vector, a common default for input. The following sections detail each of the successive routines in the setup phase: `strength`, `aggregate`, `tentative`, `prolongator`, and the triple matrix Galerkin product. One of the goals of this paper is to systematically consider the sparse matrix operations in lines 1–6.

ALGORITHM 1. AMG Setup: `setup`.

```
parameters: $A$, $B$
return: $A_0, \dots, A_M$, $P_0, \dots, P_{M-1}$

$A_0 \leftarrow A$, $B_0 \leftarrow B$
for $k = 0, \dots, M$
1    $C_k \leftarrow$ strength($A_k$)                         {strength-of-connection}
2    $Agg_k \leftarrow$ aggregate($C_k$)                      {construct coarse aggregates}
3    $T_k,\ B_{k+1} \leftarrow$ tentative($Agg_k, B_k$)       {form tentative interpolation}
4    $P_k \leftarrow$ prolongator($A_k, T_k$)                 {improve interpolation}

5    $R_k \leftarrow P_k^T$                                   {transpose}
6    $A_{k+1} \leftarrow R_k A_k P_k$                         {coarse matrix, triple-matrix product}
```

1.2.2. Strength-of-connection. A vertex i in the fine matrix graph that strongly influences or strongly depends on a neighboring vertex j typically has a large, relative edge weight. As a result, the traditional approach for aggregation schemes is to identify two points i and j as strongly connected if they satisfy

$$|A(i,j)| > \theta\sqrt{|A(i,i)A(j,j)|}. \tag{1.1}$$

This concise statement yields a connectivity graph represented by sparse matrix C_k. Algorithm 2 describes the complete strength-of-connection algorithm for the COO matrix format. A parallel implementation of this algorithm is discussed in section 3.1.

Algorithm 2. Strength of connection: `strength`.

```
    parameters:   $A_k \equiv (I, J, V)$,   COO sparse matrix
    return:   $C_k \equiv (\hat{I}, \hat{J}, \hat{V})$,   COO sparse matrix

    $\mathcal{M} = \{0, \ldots, nnz(A) - 1\}$
    $D \leftarrow 0$
1   for $n \in \mathcal{M}$                                         {extract diagonal}
        if $I_n = J_n$
            $D(I_n) \leftarrow V_n$
2   for $n \in \mathcal{M}$                                         {check strength}
        if $|V_n| > \theta\sqrt{|D(I_n)| \cdot |D(J_n)|}$
            $(\hat{I}_{\hat{n}}, \hat{J}_{\hat{n}}, \hat{V}_{\hat{n}}) \leftarrow (I_n, J_n, V_n)$
```

1.2.3. Aggregation. An aggregate or grouping of nodes is defined by a root node i and its neighborhood—i.e., all points j, for which $C(i, j) \neq 0$, where C is a strength matrix. The standard (serial) aggregation procedure consists of two phases:

1. For each node i, if i and each of its strongly connected neighbors are not yet aggregated, then form a new aggregate consisting of i and its neighbors.
2. For each remaining unaggregated node i, sweep i into an adjacent aggregate.

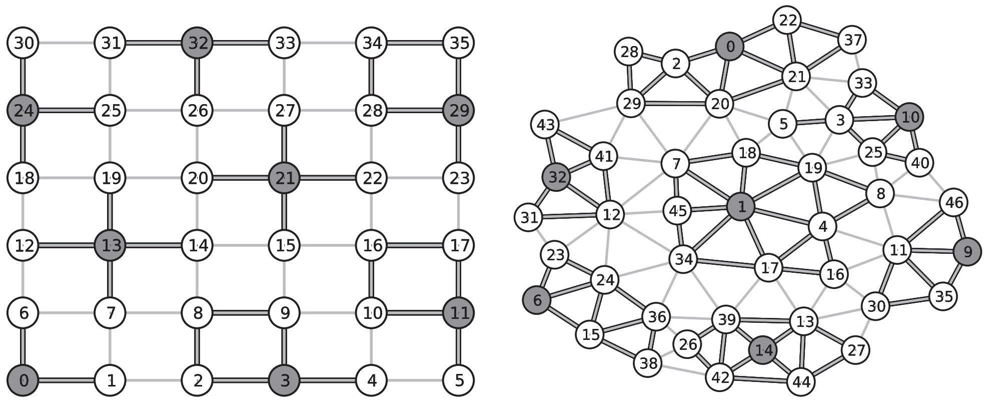
The first phase of the algorithm visits each node and attempts to create disjoint aggregates from the node and its 1-ring neighbors. It is important to note that the first phase is a greedy approach and is therefore sensitive to the order in which the nodes are visited. We revisit this artifact in section 3.2, where we devise a parallel aggregation scheme that mimics the standard sequential algorithm up to a reordering of the nodes.

Nodes that are not aggregated in the first phase are incorporated into an adjacent aggregate in the second phase. By definition, each unaggregated node must have at least one aggregated neighbor (otherwise it could be the root of a new aggregate) so all nodes are aggregated after the second phase. When an unaggregated node is adjacent to two or more existing aggregates, an arbitrary choice is made. Alternatively, an aggregate with the largest or smallest index or the aggregate with the fewest members, etc., could be selected. Figure 1.1 illustrates a typical aggregation pattern for structured and unstructured meshes.

The aggregation process results in a sparse matrix Agg, which encodes the aggregates using the following scheme:

$$Agg(i, j) = \begin{cases} 1 & \text{if the } i\text{th node is contained in the } j\text{th aggregate,} \\ 0 & \text{otherwise.} \end{cases} \tag{1.2}$$

1.2.4. Construction of the tentative prolongator. From an aggregation, represented by Agg, and a set of coarse-grid candidate vectors, represented by B, a tentative interpolation operator is defined. The tentative interpolation operator or prolongator matrix T is constructed such that each row corresponds to a grid point and each column corresponds to an aggregate. When there is only one candidate vector, the sparsity pattern of the tentative prolongator is exactly the same as Agg. Specifically, with the sparsity pattern induced by Agg_k, the numerical entries of T_k



(a) Structured mesh aggregates (b) Unstructured mesh aggregates

FIG. 1.1. *Example of a mesh (gray) and aggregates (outlined in black). Nodes are labeled with the order in which they are visited by the sequential aggregation algorithm and the root nodes, selected in the first phase of the algorithm, are colored in gray. Nodes that are adjacent to a root node, such as nodes* 1 *and* 6 *in Figure* 1.1(a) *are aggregated in phase* 1*. Nodes that are not adjacent to a root node, such as nodes* 8*,* 16*, and* 34 *in Figure* 1.1(a) *are aggregated in second phase.*

are defined by the conditions

$$B_k = T_k B_{k+1}, \qquad T_k^T T_k = I, \tag{1.3}$$

which imply that (1) the near-nullspace candidates lie in the range of T_k and (2) columns of T_k are orthonormal. For instance, in an example with five fine-grid nodes (with two coarse aggregates), we have matrices

$$B_k = \begin{bmatrix} B_{1,1} \\ B_{2,1} \\ B_{3,1} \\ B_{4,1} \\ B_{5,1} \end{bmatrix}, \qquad T_k = \begin{bmatrix} B_{1,1}/C_1 & \\ B_{2,1}/C_1 & \\ & B_{3,1}/C_2 \\ & B_{4,1}/C_2 \\ & B_{5,1}/C_2 \end{bmatrix}, \qquad B_{k+1} = \begin{bmatrix} C_1 \\ C_2 \end{bmatrix} \tag{1.4}$$

that satisfy the interpolation ($B_k = T_k B_{k+1}$) and orthonormality conditions ($T_k^T T_k = I$), using the scaling factors $C_1 = ||[B_{1,1}, B_{2,1}]||$ and $C_2 = ||[B_{3,1}, B_{4,1}, B_{5,1}]||$.

Although we only consider the case of a single candidate vector in this paper, for completeness we note that when B_k contains $m > 1$ low-energy vectors, the tentative prolongator takes on a block structure. An important component of the setup phase is to express these operations efficiently on the GPU, which we detail in section 3.4.

1.2.5. Prolongator smoothing. The tentative prolongation operator is a direct attempt to enforce the range of interpolation to coincide with the (user-provided) near-nullspace modes. This has limitations however, since the modes may not accurately represent the "true" near-nullspace and since the interpolation is still only local and thus limited in accuracy. One approach to improving the properties of interpolation is to smooth the columns of the tentative prolongation operator. With weighted Jacobi smoothing, for example, the operation computes a new sparse matrix P whose columns,

$$P(:, j) = (I - \omega D^{-1} A) T(:, j), \tag{1.5}$$

are the result of applying one-iteration of relaxation to each column of the tentative prolongator. In practice, we compute all columns of P in a single operation using a specialized sparse matrix-matrix multiplication algorithm.

1.2.6. Galerkin product. The construction of the sparse Galerkin product $A_{k+1} = R_k A_k P_k$ in line 6 of Algorithm 1 is typically implemented with two separate sparse matrix-matrix multiplies—i.e., $(RA)P$ or $R(AP)$. Either way, the first product is of the form $[n \times n] * [n \times n_c]$ (or the transpose), while the second product is of the form $[n_c \times n] * [n \times n_c]$.

Efficient sequential sparse matrix-matrix multiplication algorithms are described in [20, 3]. In these methods the compressed sparse row (CSR) format is used, which provides $\mathcal{O}(1)$ indexing of the matrix rows. As a result, the restriction matrix $R_k = P_k^T$ is formed explicitly in CSR format before the Galerkin product is computed.

While the sequential algorithms for sparse matrix-matrix multiplication are efficient, they rely on a large amount of (per thread) temporary storage and are therefore not suitable for fine-grained parallelism. Specifically, to compute the sparse product $C = A * B$, the sequential methods use $O(N)$ additional storage, where N is the number of columns in C. In contrast, our approach to sparse matrix-matrix multiplication, detailed in section 3.3, is formulated in terms of highly scalable parallel primitives with no such limitations. Indeed, our formulation exploits parallelism at the level of individual matrix entries.

1.3. Spectral radius. For the smoothers such as weighted Jacobi or Chebyshev, a calculation of the spectral radius of a matrix—i.e., the eigenvalue of the matrix with maximum modulus—is often needed in order to yield effective smoothing properties. These smoothers are central to the relaxation scheme in the cycling phase and the prolongation in the setup phase, so we consider the computational impact of these calculations. Here, we use an approximation of the spectral radius of $D^{-1}A$, where D is matrix of the diagonal of A.

To produce accurate estimates of the spectral radius we use an Arnoldi iteration which reduces A to an upper Hessenberg matrix H by similarity transformations. The eigenvalues of the small fixed-size dense matrix H are then computed directly. In section 5, we see that computing the spectral radius has a nontrivial role in the total cost of the setup phase.

1.4. Cycling phase. The multigrid cycling or solve phase is detailed in Algorithm 3. Several computations are executed at each level in the algorithm, but as we see in lines 2–6, the operations are largely sparse matrix-vector multiplications (SpMV). Consequently, on a per-level basis, we observe a strong relationship between the performance of the SpMV and the computations in lines 2–6. For example, the smoothing sweeps on lines 2 and 6 are both implemented as affine operations such as $x_k \leftarrow x_k - \omega D^{-1}(Ax_k - b)$ in the case of weighted Jacobi. This is a highly parallelized `AXPY` operation as well as a SpMV. This is also the case for the residual on line 3 and the restriction and interpolation operations on lines 4 and 5. Finally, we coarsen to only a few points so that the coarse solve on line 1 is efficiently handled by either relaxation or an arbitrary direct solver.

While the computations in the solve phase are straightforward at a high level, they rely heavily on the ability to perform *unstructured* SpMV operations efficiently. Indeed, even though the fine-level matrix may exhibit a well-defined structure, the unstructured nature of the aggregation process produces coarse-level matrices with less structure. In section 5 we examine the cost of the solve phase in more detail.

Algorithm 3. AMG Solve: `solve`.

```
parameters: $A_k$, $R_k$, $P_k$, $x_k$, $b_k$
return:   $x_k$,   solution vector

if $k = M$
1   solve $A_k x_k = b_k$
else
2   $x_k \leftarrow$ presmooth$(A_k, x_k, b_k, \mu_1)$        {smooth $\mu_1$ times on $A_k x_k = b_k$}
3   $r_k \leftarrow b_k - A_k x_k$                             {compute residual}
4   $r_{k+1} \leftarrow R_k r_r$                               {restrict residual}
    $e_{k+1} \leftarrow$ solve$(A_{k+1}, R_{k+1}, P_{k+1}, e_{k+1}, r_{k+1})$   {coarse-grid solve}
5   $e_k \leftarrow P_k e_{k+1}$                               {interpolate solution}
    $x_k \leftarrow x_k + e_k$                                 {correct solution}
6   $x_k \leftarrow$ postsmooth$(A_k, x_k, b_k, \mu_2)$       {smooth $\mu_2$ times on $A_k x_k = b_k$}
```

2. Parallel primitives.

Our method for exposing fine-grained parallelism in AMG leverages (data) *parallel primitives* [8, 33]. We use the term primitives to refer to a collection of fundamental algorithms that emerge in numerous contexts, such as reduction, parallel prefix-sum (or scan), and sorting. In short, these primitives are to general-purpose computations what BLAS [26] is to computations in linear algebra.

Given the broad scope of their usage, special emphasis has been placed on the performance of primitives and very highly optimized implementations are readily available for the GPU [33, 29, 28]. The efficiency of our solver, and hence the underlying parallel primitives, is demonstrated in section 5.

Our AMG solver is implemented almost exclusively with the parallel primitives provided by the Thrust library [23]. In the remainder of this section we identify a few of the most important Thrust algorithms and illustrate their usage. For ease of exposition we omit some of the precise usage details; however, the essential characteristics are preserved.

2.1. Reduction.

A critical component is that of simplifying an array to a single value, or a *reduction*. In Thrust, the `reduce` algorithm reduces a range of numbers to a single value by successively summing values together:

$$\texttt{reduce}([3, 4, 1, 5, 2]) \rightarrow 15.$$

The same algorithm can be used to determine the maximum entry by specifying `maximum` for the reduction operator:

$$\texttt{reduce}([3, 4, 1, 5, 2], \texttt{maximum}) \rightarrow 5.$$

In general, any function that is both commutative and associative is a valid reduction operator.

Reductions involving two vectors are implemented with the `inner_product` algorithm. As the name suggests, by default the algorithm computes the mathematical inner product (or dot product),

$$\texttt{inner\_product}([3, 4, 1, 5], [0, 1, 2, 3]) \rightarrow 21.$$

As with `reduce`, the `inner_product` algorithm supports different reduction and elementwise vector operations in place of addition and multiplication, respectively. For example,

$$\texttt{inner\_product}([1,2,3],[1,2,3],\texttt{logical\_and},\texttt{equals}) \rightarrow \texttt{true},$$
$$\texttt{inner\_product}([1,2,3],[1,0,3],\texttt{logical\_and},\texttt{equals}) \rightarrow \texttt{false},$$

tests two vectors for equality.

2.2. Scan. Similarly, the parallel prefix-sum, or *scan*, primitive computes the cumulative sum of an array and is a fundamental component of common algorithms such as stream compaction. In Thrust, the `inclusive_scan` algorithm computes the "inclusive" variant of the scan primitive,

$$\texttt{inclusive\_scan}([3,4,1,5,2]) \rightarrow [3,7,8,13,15],$$

while the `exclusive_scan` algorithm computes the "exclusive" variant,

$$\texttt{exclusive\_scan}([3,4,1,5,2],10) \rightarrow [10,13,17,18,23],$$

which incorporates a user-specified starting value and excludes the final sum. As with reduction, the scan algorithms accept other binary operations such as `maximum`,

$$\texttt{inclusive\_scan}([3,4,1,5,2],\texttt{maximum}) \rightarrow [3,4,4,5,5],$$

provided that the operator is associative.

2.3. Transformations. We also encounter *transformations* on arrays, or elementwise map operations, which are implemented with the `transform` algorithm. Unary transformations apply a unary functor to each element of an input array and write the output to another array. For example, transforming an array with the `negate` functor,

$$\texttt{transform}([3,4,1],\texttt{negate}) \rightarrow [-3,-4,-1],$$

implements vector negation. In the same way, binary transformations apply a binary functor to pairs of elements in two input arrays and write the output to another array. For example, transforming a pair of vectors with the `plus` functor,

$$\texttt{transform}([3,4,1],[4,5,7],\texttt{plus}) \rightarrow [7,9,8],$$

implements vector addition.

2.4. Gathering and scattering. Related to transformation are the `gather` and `scatter` algorithms,

$$\texttt{gather}([3,0,2],[11,12,13,14]) \rightarrow [14,11,13],$$
$$\texttt{scatter}([3,0,2],[11,12,13],[*,*,*,*]) \rightarrow [12,*,13,11],$$

which copy values based on an index map ($[3,0,2]$ in the examples). Here, the placeholder $*$ represents elements of the output range that are not changed by the algorithm. Whenever an algorithm requires an indirect load or store operator—e.g., $v = X[map[i]]$ or $Y[map[i]] = v$—a `gather` and `scatter` is typically required. Predicated versions of the functions,

$$\texttt{gather\_if}([3,0,2],[true,false,true],[11,12,13,14],[*,*,*]) \rightarrow [14,*,13],$$
$$\texttt{scatter\_if}([3,0,2],[true,false,true],[11,12,13],[*,*,*,*]) \rightarrow [*,*,13,11],$$

copy values conditionally.

2.5. Stream compaction. It is often necessary to filter elements of an array based on a given condition or *predicate*. This process, known as stream compaction [33], is implemented with Thrust by combining the `copy_if` algorithm with a predicate functor and is invaluable in expressing certain components of AMG such as strength-of-connection. For example,

$$\texttt{copy\_if}([3,4,1,5,2],\texttt{is\_even}) \rightarrow [4,2]$$

copies the even elements of an array to another array, preserving their original order. Here `is_even` is a unary functor that returns `true` if the argument is even and `false` otherwise.

The first variant of `copy_if` applies the predicate functor to the values of the input sequence and copies a subset of them to the output range. A second variant applies the predicate to a separate sequence and copies a subset of the corresponding values to the output. For example,

$$\texttt{copy\_if}([0,1,2,3,4],[3,4,1,5,2],\texttt{is\_even}) \rightarrow [1,4]$$

outputs the positions of even elements, as opposed to the values of even elements.

2.6. Segmented reduction. The `reduce` algorithm reduces an entire array, which is typically large, down to a single value. In practice it is often necessary to compute a reduction for many different arrays at once; however, using `reduce` for each one is impractical since the array size is generally too small to expose sufficient fine-grained parallelism. For this reason Thrust provides a `reduce_by_key` algorithm that implements *segmented* reduction. In segmented reduction, an array of keys determines which values belong to a segment. For example, the key array $[0,0,1,1,1,2]$ defines three segments of lengths two, three, and one, respectively. When the `reduce_by_key` algorithm encounters a set of adjacent keys that are equivalent, it reduces the corresponding values together and writes the key and the reduced value to separate output arrays. For example,

$$\texttt{reduce\_by\_key}([0,0,1,1,1,2],[10,20,30,40,50,60]) \rightarrow [0,1,2],[30,120,60].$$

Note that the key and value sequences are stored in separate arrays. This "structure of arrays" representation is generally more computationally efficient than the alternative "array of structures" representation, where keys and values are interleaved in memory.

2.7. Sorting. Sorting is a valuable primitive whenever disordered data must be binned or easily indexed. This is helpful in many of our transformations in the AMG setup phase. By default, the `sort` algorithm sorts data in ascending order,

$$\texttt{sort}([3,4,1,5,2]) \rightarrow [1,2,3,4,5],$$

which is equivalent to specifying that elements should be compared using the standard `less` comparison functor.

Thrust also provides the `sort_by_key` algorithm for sorting (logical) key-value pairs. In the key-value sort, the keys are sorted as usual, while the value associated to each key is moved to the corresponding position in the values array,

$$\texttt{sort\_by\_key}([3,4,1,5,2],[10,20,30,40,50]) \rightarrow [1,2,3,4,5],[30,50,10,20,40].$$

Stable variants of the sorting primitives, `stable_sort` and `stable_sort_by_key`, are guaranteed to preserve the relative ordering of equivalent keys.

3. Parallel hierarchy construction.

In this section we describe an implementation of the multigrid setup phase (cf. Algorithm 1) using parallel primitives to expose the fine-grained parallelism required for GPU acceleration. Our primary contributions are parallel algorithms for aggregation and sparse matrix-matrix multiplication. The complete source code for the method presented here is available in the open-source Cusp library [7].

The methods described in this section are designed for the *coordinate* (COO) sparse matrix format. The COO format comprises three arrays `I`, `J`, and `V`, which store the row indices, column indices, and values, respectively, of the matrix entries. We further assume that the COO entries are sorted by ascending row index. Although the COO representation is not generally the most efficient sparse matrix storage format, it is simple and easily manipulated. Furthermore, Cusp provides efficient conversion methods between COO and other popular matrix formats such as CSR, ELLPACK (ELL), and hybrid (HYB).

3.1. Strength of connection.

The symmetric strength-of-connection algorithm (cf. section 1.2.2) is straightforward to implement using parallel primitives. Given a matrix A in coordinate format we first extract the matrix diagonal by comparing the row index array, `I`, to the column index array, `J`,

$$\begin{aligned}
\texttt{D} &= [0,0,0,0],\\
\texttt{is\_diagonal} &= \texttt{transform}(\texttt{I},\texttt{J},\texttt{equals}),\\
&= [\texttt{true},\texttt{false},\texttt{false},\texttt{true},\texttt{false},\texttt{false},\texttt{true},\texttt{false},\texttt{false},\texttt{true}],\\
\texttt{D} &= \texttt{scatter\_if}(\texttt{V},\texttt{I},\texttt{is\_diagonal},\texttt{D}),\\
&= [2,2,2,2],
\end{aligned}$$

and scattering the corresponding entries in the values array, `V`, when they are equal.

The second loop in Algorithm 2 is implemented with stream compaction. First we obtain arrays containing `D[i]` and `D[j]` for each index in `I` and `J`, respectively,

$$\begin{aligned}
\texttt{Di} &= \texttt{gather}(\texttt{I},\texttt{D}),\\
&= [2,2,2,2,2,2,2,2,2,2],\\
\texttt{Dj} &= \texttt{gather}(\texttt{J},\texttt{D}),\\
&= [2,2,2,2,2,2,2,2,2,2],
\end{aligned}$$

from which the strength-of-connection threshold—i.e., $\theta\sqrt{|D(I_n)|\cdot|D(J_n)|}$ in Algorithm 2—of each entry is computed,

$$\texttt{threshold} = \texttt{transform}(\texttt{Di},\texttt{Dj},\texttt{soc\_threshold(theta)}),$$

using a specially defined functor `soc_threshold`. Next, each entry in the values array, `V`, is tested against the corresponding threshold to determine which entries are strong,

$$\begin{aligned}
\texttt{is\_strong} &= \texttt{transform}(\texttt{V},\texttt{threshold},\texttt{greater}),\\
&= [\texttt{true},\texttt{true},\texttt{true},\texttt{true},\texttt{true},\texttt{true},\texttt{true},\texttt{true},\texttt{true},\texttt{true}].
\end{aligned}$$

Finally, the matrix entries corresponding to strong connections are determined using stream compaction to form a coordinate representation of the strength-of-connection matrix, namely,

$$\begin{aligned}\texttt{Ci} &= \texttt{copy\_if}(\texttt{I}, \texttt{is\_strong}, \texttt{identity}),\\ \texttt{Cj} &= \texttt{copy\_if}(\texttt{J}, \texttt{is\_strong}, \texttt{identity}),\\ \texttt{Cv} &= \texttt{copy\_if}(\texttt{V}, \texttt{is\_strong}, \texttt{identity}).\end{aligned}$$

In practice we do not construct arrays such as `Di` and `threshold` explicitly in memory. Instead, the `gather` and `transform` operations are *fused* with subsequent algorithms to conserve memory bandwidth using Thrust's `permutation_iterator` and `transform_iterator`. Similarly, the three calls to `copy_if` are combined into a single stream compaction operation using a `zip_iterator`. The translation from the explicit version of the algorithm (shown here) and the more efficient, fused version (not shown) is only a mechanical transformation. We refer to the Cusp source code [7] for additional detail.

3.2. Aggregation. The sequential aggregation method (cf. section 1.2.3) is designed to produce aggregates with a particular structure. Unfortunately, the greedy nature of the algorithm introduces sequential dependencies that prevent a direct parallelization of the algorithm's first phase. In this section we describe a fine-grained parallel analogue of the sequential method based on a generalized *maximal independent set* algorithm which produces aggregates with the same essential properties. A similar parallel aggregation strategy is described in [35], albeit with a different implementation.

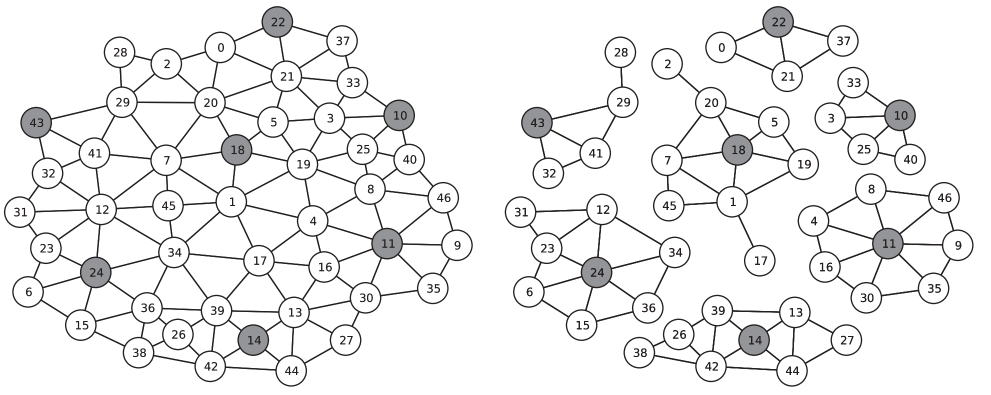
There are two observations regarding the aggregation depicted in Figure 1.1 that lead to the description of our aggregation method. First, no two root nodes of the aggregates are within a distance of two edges apart. Second, if any unaggregated node is separated from all existing root nodes by more than two edges, then it is free to become the root of a new aggregate. Together, these conditions define a collection of root nodes that are a distance-2 maximal independent set, which we denote MIS(2) and formalize in Definition 3.1. The first property ensures independence of the nodes—i.e., that the minimum distance between any pair of root nodes exceeds a given threshold. The second property ensures maximality—i.e., no additional node can be added to the set without violating the property of independence. The standard definition of a maximal independent set, which we denote MIS(1), is consistent with this definition except with a distance of one. We defer a complete description of the generalized maximal independent set algorithm to the appendix. For the remainder of this section, we assume that a valid MIS(2) is computed efficiently in parallel.

DEFINITION 3.1 (MIS(k)). *Given a graph $G = (V, E)$, let $V_{root} \subset V$ be a set of root nodes, and denote by $d_G(\cdot, \cdot)$ the distance or shortest path between two vertices in the graph. Then, V_{root} is a maximal independent set of distance k or* MIS(k) *if the following hold:*

1. *(independence) Given any two vertices $u, v \in V_{root}$, then $d_G(u, v) > k$.*
2. *(maximality) There does not exist $u \in V \setminus V_{root}$ such that $d_G(u, v) > k$ for all $v \in V_{root}$.*

Given a valid MIS(2) the construction of aggregates is straightforward. Assume that the MIS(2) is specified by an array of values in $\{0, 1\}$, where a value of 1 indicates that the corresponding node is a member of the independent set and a value of 0 indicates otherwise. We first enumerate the MIS(2) nodes with an `exclusive_scan` operation, giving each set node a unique index in $[0, N-1]$, where N is the number of nodes in the set. For example, on a linear graph with 10 nodes, a MIS(2) with 4 set nodes,

$$\texttt{mis} = [1, 0, 0, 1, 0, 0, 1, 0, 0, 1],$$



(a) MIS(2) (b) Aggregates

FIG. 3.1. *Parallel aggregation begins with a* MIS(2) *set of nodes, in gray, and represents the root of an aggregate—e.g., node* 18. *As in the sequential method, nodes adjacent to a root node are incorporated into the root node's aggregate in the first phase. In the second phase, unaggregated nodes join an adjacent aggregate—e.g., nodes* 12, 19, *and* 24 *for root node* 18.

is enumerated as

$$\begin{aligned}\texttt{enum} &= \texttt{exclusive\_scan}(\texttt{mis}, 0),\\ &= [0, 1, 1, 1, 2, 2, 2, 3, 3, 3].\end{aligned}$$

Since the ith node in the independent set serves as the root of the ith aggregate, the only remaining task is to propagate the root indices outward.

The root indices are communicated to their neighbors with two operations resembling an SpMV, $y = Ax$. Conceptually, each unaggregated node looks at neighbors for the index of an aggregate. In the first step, all neighbors of a root node receive the root node's aggregate index—i.e., the value resulting from the `exclusive_scan` operation. In the second step, the remaining unaggregated nodes receive the aggregate indices of their neighbors, at least one of which must belong to an aggregate. As before, in the presence of multiple neighboring aggregates a selection is made arbitrarily. The two sparse matrix-vector operations are analogous to the first and second phases of the sequential algorithm, respectively; cf. section 1.2.3. Our implementation of the parallel aggregate propagation step closely follows the existing techniques for SpMV [5].

Figure 3.1 depicts a MIS(2) for a regular mesh and the corresponding aggregates rooted at each independent set node. Although the root nodes are selected by a randomized process (see the appendix), the resulting aggregates are qualitatively similar to those chosen by the sequential algorithm in Figure 1.1(a). Indeed, with the appropriate permutation of graph nodes, the sequential aggregation algorithm selects the same root nodes as the randomized MIS(2) method.

3.3. Sparse matrix-matrix multiplication. Efficiently computing the product $C = AB$ of sparse matrices A and B is challenging, especially in parallel. The central difficulty is that for irregular matrices, the structure of the output matrix C has a complex and unpredictable dependency on the input matrices A and B.

The sequential sparse matrix-matrix multiplication algorithm mentioned in section 1.2.6 does not admit immediate, fine-grained parallelization. Specifically, for matrices A and B of size $[k{\times}m]$ and $[m{\times}n]$ the method requires $O(n)$ temporary storage to determine the entries of each sparse row in the output. As a result, a straightforward parallelization of the sequential scheme requires $O(n)$ storage per thread, which is untenable when one is seeking to develop tens of thousands of independent threads of execution. While it is possible to construct variations of the sequential method with lower per-thread storage requirements, any method that operates on the granularity of matrix rows—i.e., distributing matrix rows over threads—requires a nontrivial amount of per-thread state and suffers load imbalances for certain input. As a result, we have designed an algorithm for sparse matrix-matrix multiplication based on sorting, segmented reduction, and other operations which are well-suited to fine-grained parallelism as discussed in section 2.

As an example, we demonstrate our algorithm for computing $\mathtt{C} = \mathtt{A} * \mathtt{B}$, where

$$\mathtt{A} = \begin{bmatrix} 5 & 10 & 0 \\ 15 & 0 & 20 \end{bmatrix} \quad \text{and} \quad \mathtt{B} = \begin{bmatrix} 25 & 0 & 30 \\ 0 & 35 & 40 \\ 45 & 0 & 50 \end{bmatrix} \tag{3.1}$$

have four and six nonzero entries, respectively. The matrices are stored in coordinate format as

$$\mathtt{A} = \begin{bmatrix} (0,0,\ \ 5) \\ (0,1,10) \\ (1,0,15) \\ (1,2,20) \end{bmatrix} \quad \text{and} \quad \mathtt{B} = \begin{bmatrix} (0,0,25) \\ (0,2,30) \\ (1,1,35) \\ (1,2,40) \\ (2,0,45) \\ (2,2,50) \end{bmatrix}, \tag{3.2}$$

where each (i,j,v) tuple of elements represents the row index, column index, and value of the matrix entry. We note that the (i,j,v) tuples are only a logical construction used to explain the algorithm. In practice the coordinate format is stored in a "structure of arrays" layout with three separate arrays.

To expose fine-grained parallelism, our parallel sparse matrix-matrix multiplication algorithm proceeds in three stages:

1. *expansion* of $\mathtt{A} * \mathtt{B}$ into a intermediate coordinate format $\mathtt{T}$,
2. *sorting* of $\mathtt{T}$ by row and column indices to form $\hat{\mathtt{T}}$,
3. *compression* of $\hat{\mathtt{T}}$ by summing duplicate values for each matrix entry.

In the first stage, $\mathtt{T}$ is formed by multiplying each entry $\mathtt{A}(i,j)$ of the left matrix with every entry in row j of $\mathtt{B}$, $\mathtt{B}(j,k)$ for all k. Here, the intermediate format is

$$\mathtt{T} = \begin{bmatrix} (0,0,\ \ 125) \\ (0,2,\ \ 150) \\ (0,1,\ \ 350) \\ (0,2,\ \ 400) \\ (1,0,\ \ 375) \\ (1,2,\ \ 450) \\ (1,0,\ \ 900) \\ (1,2,1000) \end{bmatrix}. \tag{3.3}$$

The expansion stage is implemented in parallel using gather, scatter, and prefix-sum operations. We refer to the Cusp source code [7] for further detail.

The result of the expansion phase is an intermediate coordinate format with possible duplicate entries that is sorted by row index but not by column index. The second stage of the sparse matrix-matrix multiplication algorithm sorts the entries of T into lexicographical order. For example, sorting the entries of T by the (i, j) coordinate yields

$$\hat{\mathtt{T}} = \begin{bmatrix} (0,0,\ \ 125) \\ (0,1,\ \ 350) \\ (0,2,\ \ 150) \\ (0,2,\ \ 400) \\ (1,0,\ \ 375) \\ (1,0,\ \ 900) \\ (1,2,\ \ 450) \\ (1,2,1000) \end{bmatrix}, \tag{3.4}$$

from which the duplicate entries are easily identified. The lexicographical reordering is efficiently implemented with two invocations of Thrust's `stable_sort_by_key` algorithm.

The final stage of the algorithm compresses duplicate coordinate entries while summing their corresponding values. Since $\hat{\mathtt{T}}$ is sorted by row and column, the duplicate entries are stored contiguously and are compressed with a single application of the `reduce_by_key` algorithm. In our example $\hat{\mathtt{T}}$ becomes

$$\mathtt{C} = \begin{bmatrix} (0,0,\ \ 125) \\ (0,1,\ \ 350) \\ (0,2,\ \ 550) \\ (1,0,1275) \\ (1,2,1450) \end{bmatrix}, \tag{3.5}$$

where the duplicate entries have been combined. The compressed result is now a valid, duplicate-free coordinate representation of the matrix

$$\mathtt{A} * \mathtt{B} = \begin{bmatrix} 125 & 350 & 550 \\ 1275 & 0 & 1450 \end{bmatrix}.$$

All three stages of our sparse matrix-matrix multiplication algorithm expose ample fine-grained parallelism. Indeed, even modest problem sizes generate enough independent threads of execution to fully saturate the GPU. Furthermore, because we have completely flattened the computation into efficient data-parallel algorithms—i.e., **`gather`**, **`scatter`**, **`scan`**, **`stable_sort_by_key`**—our implementation is insensitive to the (inherent) irregularity of sparse matrix-matrix multiplication. As a result, even pathological inputs do not create substantial imbalances in the work distribution among threads.

Another benefit is that our parallel sparse matrix-matrix algorithm has the same computational complexity as the sequential method, which is $O(nnz(\mathtt{T}))$, the number of entries in our intermediate format. Therefore our method is "work-efficient" [8], since (1) the complexity of the sequential method is proportional to the size of the intermediate format T and (2) the work involved at each stage of the algorithm is linear in the size of T. The latter claim is valid since Thrust employs work-efficient implementations of parallel primitives such as `scan`, `reduce_by_key` and `stable_sort_by_key`.

One practical limitation of the method as described above is that the memory required to store the intermediate format is potentially large. For instance, if A and B are both square $n \times n$ matrices with exactly K entries per row, then $O(nK^2)$ bytes of memory are needed to store T. Since the input matrices are generally large themselves ($O(nK)$ bytes) it is not always possible to store a K-times larger intermediate result in memory. In the limit, if A and B are dense matrices (stored in sparse format), then $O(n^3)$ storage is required. As a result, our implementation allocates a workspace of bounded size and decomposes the matrix-matrix multiplication $\mathtt{C} = \mathtt{A} * \mathtt{B}$ into several smaller operations of the form $\mathtt{C(slice,:)} = \mathtt{A(slice,:)}*\mathtt{B}$, where *slice* is a subset of the rows of A. The final result is obtained by simply concatenating the coordinate representations of all the partial results together $\mathtt{C} = [\mathtt{C(slice\_0,:)}, \mathtt{C(slice\_1,:)}, \ldots]$. In practice this subslicing technique introduces little overhead because the workspace is still large enough to fully utilize the device. We refer to the Cusp source code [7] for additional details.

3.4. Prolongation and restriction. The tentative prolongation, prolongation smoothing, and restriction construction steps of Algorithm 1 (lines 3, 4, and 5), are also expressed with parallel primitives. The tentative prolongation operation is constructed by gathering the appropriate entries from the near-nullspace vectors stored in B according to the sparsity pattern defined by Agg. Then, the columns are normalized, first by transposing the matrix, which has the effect of sorting the matrix entries by column index, and then by computing the norm of each column using the reduce_by_key algorithm. Specifically, the transpose of a coordinate format matrix such as

$$\begin{aligned} \mathtt{I} &= \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 \end{bmatrix}, \\ \mathtt{J} &= \begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 0 \end{bmatrix}, \\ \mathtt{V} &= \begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 2 \end{bmatrix} \end{aligned}$$

is computed by reordering the column indices of the matrix and applying the same permutation to the rows and values

$$\begin{aligned} \mathtt{TransI}, \mathtt{Permutation} &= \mathtt{stable\_sort\_by\_key}(\mathtt{J}, [0,1,2,3,4]) \\ &= \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 3 & 5 & 1 & 2 & 4 \end{bmatrix}, \\ \mathtt{TransJ} &= \mathtt{gather}(\mathtt{Permutation}, \mathtt{I}) \\ &= \begin{bmatrix} 0 & 3 & 5 & 1 & 2 & 4 \end{bmatrix}, \\ \mathtt{TransV} &= \mathtt{gather}(\mathtt{Permutation}, \mathtt{V}) \\ &= \begin{bmatrix} 0 & 1 & 2 & 0 & 1 & 2 \end{bmatrix}. \end{aligned}$$

Then the squares of the transposed values array are calculated, followed by row sums,

$$\begin{aligned} \mathtt{Squares} &= \mathtt{tranform}(\mathtt{TransV}, \mathtt{TransV}, \mathtt{multiplies}) \\ &= \begin{bmatrix} 0 & 1 & 4 & 0 & 1 & 4 \end{bmatrix}, \\ \mathtt{Rows}, \mathtt{Sums} &= \mathtt{reduce\_by\_key}(\mathtt{TransI}, \mathtt{Squares}, \mathtt{multiplies}) \\ &= \begin{bmatrix} 0 & 1 \end{bmatrix}, \begin{bmatrix} 9 & 9 \end{bmatrix}, \end{aligned}$$

which correspond to column sums of the original matrix.

Next, the final prolongator P is obtained by smoothing the columns of T according to the formula in section 1.2.5. Here, we apply a specialized form of the general sparse matrix-matrix multiplication scheme described in section 3.3. Specifically, we exploit the special structure of the tentative prolongator, whose rows contain at most one nonzero entry, when computing the expression AT.

Table 4.1
Sparse matrix conversion times (ms) for an unstructured mesh with $1M$ *vertices and* $8M$ *edges.*

From\To	COO	CSR	ELL	HYB
COO	5.09	6.35	20.10	23.55
CSR	7.80	4.03	21.61	24.84
ELL	17.02	18.26	5.90	22.32
HYB	63.27	69.40	83.08	4.12

Finally, the transpose of the prolongation operator is calculated explicitly as $R = P^T$, and the Galerkin triple-matrix product $A_{k+1} = R_k(A_kP_k)$ is computed with two separate sparse matrix-matrix multiplies. As mentioned above, the transpose operation is fast, particularly for the COO format.

4. Parallel multigrid cycling.

After the multigrid hierarchy has been constructed using the techniques in section 3, the cycling of Algorithm 3 proceeds. In this section we describe the components of the multigrid cycling and how they are parallelized on the GPU.

4.1. Vector operations.

In Algorithm 3, the residual vector computation and the coarse grid correction steps require vector-vector subtraction and addition, respectively. While these operations could be fused with the preceding sparse matrix-vector products for potential efficiency, or could be carried out with `DAXPY` in CUBLAS [14], we have implemented equivalent routines with Thrust's `transform` algorithm for simplicity and to keep Cusp self-contained. Similarly, the vector norms (`DNRM2`) and inner products (`DDOT`) that arise in multigrid cycling have been implemented with `reduce` and `inner_product` in Thrust, respectively.

4.2. SpMV.

SpMV, which involves (potentially) irregular data structures and memory access patterns, is more challenging to implement than the aforementioned vector operations. Nevertheless, efficient techniques exist for matrices with a wide variety of sparsity patterns [11, 9, 38, 15, 39, 4, 5]. Our implementations of SpMV are described in [5, 6]. In Algorithm 3 SpMV is used to compute the residual, to restrict the fine-level residual to the coarse grid, to interpolate the coarse-level solution onto the finer grid, and, in many cases, to implement the pre- and postsmoother.

In section 3 we describe a method for constructing the AMG hierarchy in parallel using the COO sparse matrix format. The COO format is simple to construct and manipulate and therefore is well-suited for the computations that arise in the setup phase. However, the COO format is generally not the most efficient for the SpMV operations [6]. Fortunately, once the hierarchy is constructed it remains unchanged during the cycling phase. As a result, it is beneficial to convert the sparse matrices stored throughout the hierarchy to an alternative format that achieves higher SpMV performance.

In Cusp, conversions between COO and other sparse matrix formats, such as CSR, DIA, ELL, and HYB [6], are inexpensive, as shown in Table 4.1. Here we see that the conversion from COO to CSR is trivial, while to more structured formats such as ELL and HYB is of minimal expense. (Note that the conversion to itself represents a straight copy.) When reporting performance figures in section 5 we include the COO to HYB conversion time in the setup phase.

4.3. Smoothing.

Gauss–Seidel relaxation is a popular multigrid smoother with several attractive properties. For instance, the method requires only $O(1)$ temporary storage and converges for any symmetric, positive-definite matrix. Unfortunately, the standard Gauss–Seidel scheme does not admit an efficient parallel implementation. Jacobi relaxation is a simple and highly parallel alternative to Gauss–Seidel, albeit without the same computational properties. Whereas Gauss–Seidel updates each unknown immediately, Jacobi's method updates all unknowns in parallel and therefore requires a temporary vector. Additionally, a weighting or damping parameter ω must be incorporated into Jacobi's method to ensure convergence. The weighted Jacobi method is written in matrix form as $I - \frac{\omega}{\rho(D^{-1}A)} D^{-1}A$, where D is a matrix containing the diagonal elements of A. Since the expression is simply a vector operation and an SpMV, Jacobi's method exposes substantial fine-grained parallelism.

We note that SpMV is the main workhorse for several other highly parallel smoothers. Indeed, so-called polynomial smoothers

$$x \leftarrow x + P(A)r, \tag{4.1}$$

where $P(A)$ is a polynomial in matrix A, are almost entirely implemented with sparse matrix-vector products. We refer to [1] for a comprehensive treatment of parallel smoothers and their associated trade-offs.

5. Evaluation.

In this section we examine the performance of a GPU implementation of the proposed method. We investigate both the setup phase of Algorithm 1 and the solve phase of Algorithm 3 and find tangible speedups in each.

5.1. Test platforms.

The specifications of our test system are listed in Table 5.1. Our system is configured with CUDA v4.0 [30] and Thrust v1.4 [23]. As discussed in section 2, Thrust provides many highly optimized GPU parallel algorithms for reduction, sorting, etc. The entire setup phase, and most of the cycling phase, of our GPU method is implemented with Thrust. As a basis for comparison, we also consider the Intel Math Kernel Library (MKL) version 10.3. MKL provides sparse matrix-matrix multiplication as well as sparse BLAS subroutines such as sparse matrix-vector multiplication.

The Trilinos Project provides a smoothed aggregation-based AMG preconditioner for solving large, sparse linear systems in the ML package [17]. In our comparison, we use Trilinos version 10.6 and specifically ML version 5.0 for the solver. The ML results are presented in order to provide context for the performance of our solver in comparison to a well-known software package.

5.2. Model problem.

Table 5.2 describes the sparse matrices considered in our performance evaluation. We present results on both structured and unstructured grids derived from a tessellation of the unit square, unit cube, and horseshoe. The unstructured meshes are constructed with no inherent structure and the nature of the

Table 5.1
Specifications of the test platform.

Testbed	
GPU	NVIDIA Tesla C2050
CPU	Intel Core i7 950
CLOCK	3.07 GHz
OS	Ubuntu 10.10
Host Compiler	GCC 4.4.5
Device Compiler	NVCC 4.0

Table 5.2
Details of the model problem. Here h represents an average mesh diameter for the tessellation.

Matrix	Unknowns	Nonzeros
1a. 2D FD, 5-point	1,048,576	5,238,784
1b. 2D FE, 9-point	1,048,576	9,424,900
2a. 3D FD, 7-point	1,030,301	7,150,901
2b. 3D FE, 27-point	1,030,301	27,270,901
3a. 2D FE, $h \approx 0.03$	550,387	3,847,375
3b. 2D FE, $h \approx 0.02$	1,182,309	8,268,165
3c. 2D FE, $h \approx 0.015$	2,185,401	15,287,137
4. 3D FE, $h \approx 0.15$	1,088,958	17,095,986
5a. 2D FE, horseshoe	853,761	5,969,153
5b. 2D FE, square	832,081	5,817,905

tessellation is reflected in the sparse matrix, where block and banded patterns are not easily deduced, as they are in the structured case. We consider this case here, since many preconditioners rely on the structured nature of the problem, whereas one advantage of AMG is its relative indifference to matrix structure. Additionally, we also study anisotropic rotated diffusion with an angle of rotation of $\pi/6$ and a diffusion coefficient of 10^{-3} for two meshes, 5a and 5b in Table 5.2, using $\theta = 0.08$ for the symmetric strength-of-connection measure.

The problem we consider is a two- and three-dimensional Poisson problem with Dirichlet boundary conditions. Since AMG is known to perform well on such problems, this choice allows us to focus directly on the efficiency of the parallel implementation, rather than on the merits of AMG as a preconditioner.

5.3. Component performance. Table 5.3 reports timings for sparse matrix-vector multiplication, sparse matrix transposition, and sparse matrix-matrix multiplication using MKL and Cusp for the CPU and GPU results, respectively. All computations use double precision (64-bit) floating point arithmetic. The timings in Tables 5.4 and 5.3 are reported in milliseconds, averaged over 100 operations.

Table 5.4 demonstrates the performance of the level 1 BLAS functions `DDOT` and `DAXPY`, which are applied frequently during the cycling phase of the solver. These algorithms execute very few arithmetic operations per memory access and are therefore limited by the available memory bandwidth. For the largest input size the GPU achieves a maximum speedup of 7.8× in `DDOT`. On the smallest input size, the GPU results in a more modest speedup of 3.3× in `DDOT`, which is attributed to fixed costs in our implementation. In contrast, the `DAXPY` speedup is relatively constant across input sizes and simply reflects the relative memory bandwidth of the two processors.

SpMV operations are used heavily in the cycling phase of the solver (cf. Algorithm 3) and to a lesser extent in the setup phase (cf. Algorithm 1) as well. Table 5.3 reports SpMV timings using the HYB format on the GPU and CSR format on the CPU. As with level 1 BLAS operations, the SpMV operation performs few arithmetic operations per memory access and is memory bound. However, SpMV performance is *also* sensitive to the sparsity structure of the matrix, as it impacts the locality of memory accesses and creates potential variation in the amount of work per thread. Therefore, SpMV does not always saturate memory bandwidth to the same degree as algorithms with simpler memory access patterns such as `DAXPY`. (See [5, 6] for a more detailed study of GPU SpMV performance.) Across the cases considered, the GPU achieves an average speedup of 5.9× in comparison to the CPU. Figure 5.1 demon-

TABLE 5.3
SpMV time, transpose time, Galerkin product time, speedups on the GPU in bold.

	SpMV			Transpose			RAP		
Matrix	CPU	GPU		CPU	GPU		CPU	GPU	
1a. 2D FD	6.5	0.8	**8.2**	27.3	9.7	**2.8**	245.2	186.8	**1.3**
1b. 2D FE	9.5	1.3	**7.2**	25.8	10.1	**2.6**	345.3	286.4	**1.2**
2a. 3D FD	8.0	1.0	**8.1**	39.5	14.5	**2.7**	664.9	432.9	**1.5**
2b. 3D FE	23.0	3.4	**6.7**	47.6	16.7	**2.9**	1314.7	1289.7	**1.0**
3a. 2D FE	4.2	0.9	**4.8**	15.0	5.8	**2.6**	255.4	125.3	**2.0**
3b. 2D FE	8.9	1.7	**5.3**	33.7	11.2	**3.0**	548.3	270.5	**2.0**
3c. 2D FE	16.8	3.2	**5.3**	68.7	20.1	**3.4**	1037.0	497.1	**2.1**
4. 3D FE	16.3	6.5	**2.5**	52.6	17.7	**3.0**	1704.8	1042.1	**1.6**
5a. 2D FE	8.6	1.4	**6.3**	29.0	10.4	**2.8**	493.5	285.4	**1.7**
5b. 2D FE	10.1	1.9	**5.2**	38.8	10.0	**3.9**	588.2	242.6	**2.4**

TABLE 5.4
`DDOT` and `DAXPY` bandwidth (GB/s).

	`DDOT`			`DAXPY`		
Size	CPU	GPU	Speedup	CPU	GPU	Speedup
1M	17.00	55.42	**3.3**	17.69	120.55	**6.8**
2M	17.59	78.15	**4.4**	18.05	121.32	**6.7**
4M	17.70	97.33	**5.5**	18.29	121.50	**6.6**
8M	17.96	110.44	**6.2**	18.27	121.28	**6.6**
16M	17.61	118.76	**6.7**	18.11	120.79	**6.7**
32M	15.81	123.48	**7.8**	17.78	120.25	**6.8**

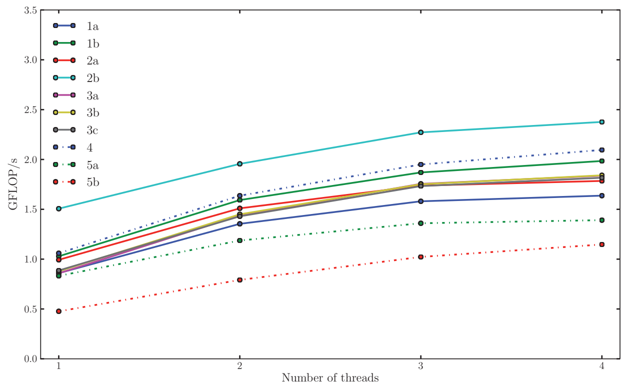


FIG. 5.1. *MKL SpMV GFlops per matrix for a varying number of threads.*

strates that although the CPU SpMV performance does benefit from multithreading; the marginal speedup diminishes rapidly as the available memory bandwidth is saturated.

A sparse matrix transpose operation is used at each level of the setup phase (cf. Algorithm 1) to obtain the restriction operator $R_k = P_k^T$ from the prolongation operator P_k. Our parallel algorithm for transpose is $3.0\times$ faster than MKL on average. We note that the MKL transpose does not benefit from multithreading, suggesting the that underlying implementation is sequential. Indeed, our own sequential implementation of sparse matrix transpose offers equivalent performance.

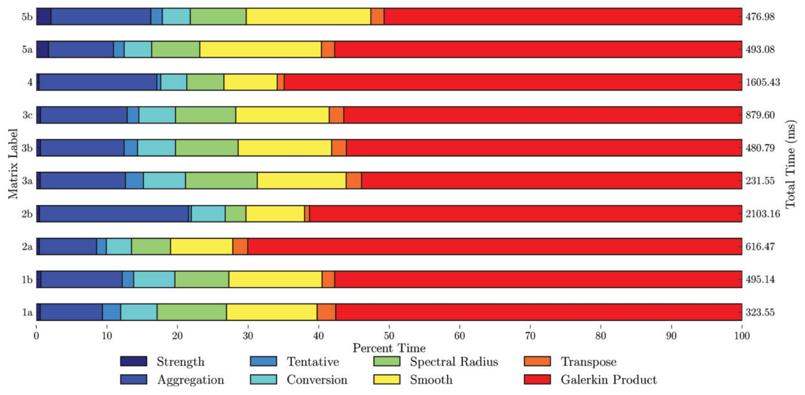


FIG. 5.2. *Relative time on the GPU of each setup phase component on the finest grid level.*

The Galerkin product results represent the cost of constructing the coarse-grid matrix by performing two sparse matrix-matrix multiplication operations. Whereas the cost of other components, such as BLAS algorithms, is directly proportional to the input size, the cost of sparse matrix-matrix multiplication is dependent on the specific sparsity patterns of the two matrices and can differ between inputs of the same size. Our sparse matrix-matrix multiplication algorithm achieves better performance than MKL in all cases considered, with an average speedup of 1.7×. As with the transpose, MKL's sparse matrix-matrix multiplication functionality extracts no observable benefit from multithreading and offers inferior performance to our own sequential implementation based on the SMMP method [3]. As a result, the CPU performance results in the remainder of this section only use MKL for sparse matrix-vector multiplication.

5.4. Setup phase performance. Section 5.3 demonstrates that GPU implementations of the essential sparse matrix operations achieve tangible speedups over a competent CPU implementations. In this section we analyze the performance of the whole multigrid setup phase (cf. Algorithm 1).

Figure 5.2 shows a breakdown of the parallel setup phase into the individual components. The figure identifies several intensive calculations in the algorithm, namely, aggregation and the Galerkin product. As anticipated, the ability to express the strength-of-connection, tentative prolongator construction, and prolongator smoothing functions in terms of parallel primitives leads to a efficient execution on the GPU. Moreover, the figure also shows that matrix conversion is a relatively low cost, enabling the use of SpMV-optimized matrix formats in the subsequent cycling phase. Moreover, the conversion to HYB format is independent of the other operations and may be performed at any time prior to the solve phase. Since matrix conversions are not performed in place, there are two versions of `A` in memory and all operations utilize the COO format as input.

The Galerkin product, which is implemented with two sparse matrix-matrix multiplications $\mathtt{R} * (\mathtt{A} * \mathtt{P})$, is the most expensive step in all cases considered. The relative cost of the Galerkin product is slightly higher in the three-dimensional matrix examples (2a, 2b, and 4) compared to the two-dimensional cases.

In Table 5.5, as a baseline we include timings from the ML package of Trilinos on the same hardware; this validates that our CPU approach is in fact a competitive implementation. ML is configured using the default parameters in the setup phase

Table 5.5
AMG setup time for all components (ms).

Matrix	CPU	GPU	Speedup	Trilinos-ML
1a. 2D FD	868	490	1.8	2040
1b. 2D FE	1062	611	1.7	2298
2a. 3D FD	1624	914	1.8	2906
2b. 3D FE	2849	2212	1.3	4420
3a. 2D FE	638	323	2.0	1324
3b. 2D FE	1436	619	2.3	2785
3c. 2D FE	2811	1047	2.7	5236
4. 3D FE	3092	1811	1.7	4967
5a. 2D FE	1653	1464	1.1	–
5b. 2D FE	1583	904	1.8	–

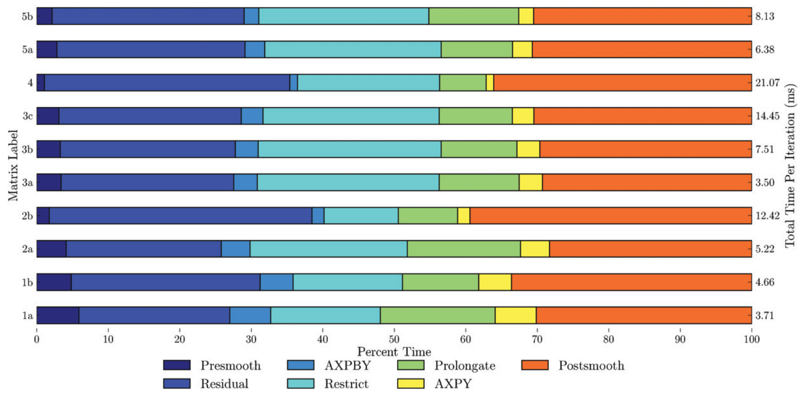


Fig. 5.3. *GPU V-cycle time breakdown on the finest grid level.*

and a Jacobi smoother in the solve phase, as opposed to the default symmetric Gauss–Seidel smoother. We use a damping factor of $\frac{5}{7}$ in the three-dimensional problems as recommended in [17] for the Jacobi smoother. The GPU implementation is faster in all 10 examples, with an average speedup of 1.8× over the CPU reference.

5.5. Solve phase performance. In this section we present the results of our AMG preconditioned CG solver for each matrix in our test suite. The AMG preconditioner uses a single iteration of weighted Jacobi at the of pre- and postsmoothing steps of the V-cycle. The Trilinos data was collected using the default uncoupled aggregation method with a tolerance of 10^{-12} and a maximum of 500 iterations.

Figure 5.3 shows that the main cost of the cycling is in restriction, prolongation, postsmoothing, and the computation of the residual, which are all sparse matrix-vector multiplication operations. Moreover, presmoothing is less costly than postsmoothing because we exploit the fact that x is initially zero and replace the smoother $x \leftarrow x + D^{-1}r$, with the equivalent expression $x \leftarrow D^{-1}b$, with a savings of one sparse matrix-vector multiply. Furthermore, Figure 5.4 shows that the entire AMG cycling is on the order of several matrix vector products, $\mathtt{A} * \mathtt{p}$.

Table 5.6 presents the results of the solve phase performance of our CPU and GPU implementations in addition to results from Trilinos ML. The performance of ML is predictably poor because the solve phase is executed on a single core; therefore in each iteration the SpMV operation, optimized for distributed execution, incurs considerable overhead. The increase in the number of iterations required by the

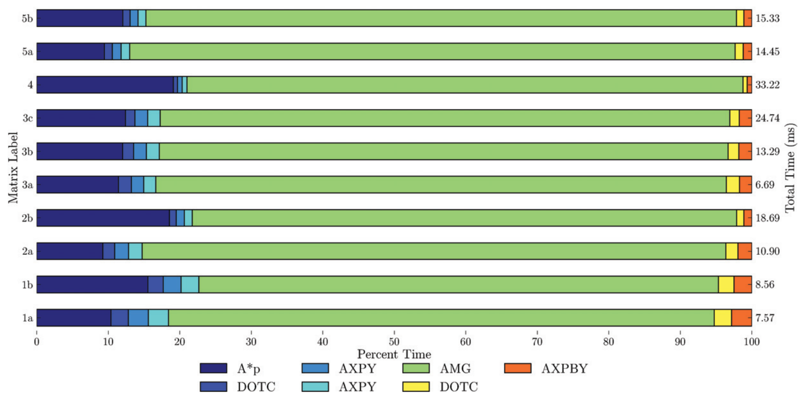


FIG. 5.4. *Total preconditioned solver time breakdown time on the GPU.*

TABLE 5.6
AMG-PCG GPU solve time (ms), number of solver iterations, and per-iteration speedup.

Matrix	CPU		GPU		Speedup	ML	
	time	it.	time	it.	(per it.)	time	it.
1a. 2D FD	1225	20	427	51	**7.6**	14190	33
1b. 2D FE	1082	16	450	46	**7.6**	10590	22
2a. 3D FD	1755	23	294	27	**6.7**	14800	31
2b. 3D FE	1679	14	483	24	**5.9**	13840	20
3a. 2D FE	1535	42	286	49	**5.3**	14020	53
3b. 2D FE	3726	47	634	54	**5.8**	34410	68
3c. 2D FE	7787	53	1426	65	**5.8**	44530	65
4. 3D FE	4343	43	1499	50	**3.0**	28380	47
5a. 2D FE	27697	397	4092	333	**4.8**	–	–
5b. 2D FE	25716	369	4556	309	**4.5**	–	–

GPU reflects the fact that on structured problems, the sequential aggregation method fortuitously selects square-shaped aggregates, while the parallel aggregation method based on MIS(2) creates irregularly shaped aggregates. Despite the increase in the number of iterations performed on the device, the increased SpMV performance using the HYB format allows the GPU to outperform the CPU in all cases.

The larger number of iterations required to solve the anisotropic problems—i.e., 5a 2D FE and 5b 2D FE—on both the CPU and the GPU reflects a weakness of the simple strength-of-connection measure considered in this paper. Since our focus is on exposing fine-grained parallelism in a well-known AMG method, we remark that more robust strength-of-connection schemes should be employed to improve convergence for anisotropic problems [31].

In contrast with the results reported in Table 5.6, where two independent hierarchies generated on the CPU and GPU are used in the cycling phase of the solver, in Figure 5.5 the same multigrid hierarchy is used on both the CPU and the GPU to isolate the performance of the cycling phase per level. The results demonstrate that the GPU is noticeably faster than the CPU on the first two or three grid levels, which represents the vast majority of the computation in a multigrid V-cycle. The coarsest grid levels are processed more rapidly by the CPU, owing to communication latency and the fact that the amount of parallelism on the smaller grids is insufficient to saturate the GPU. However, since the absolute time spent on such grids is small,

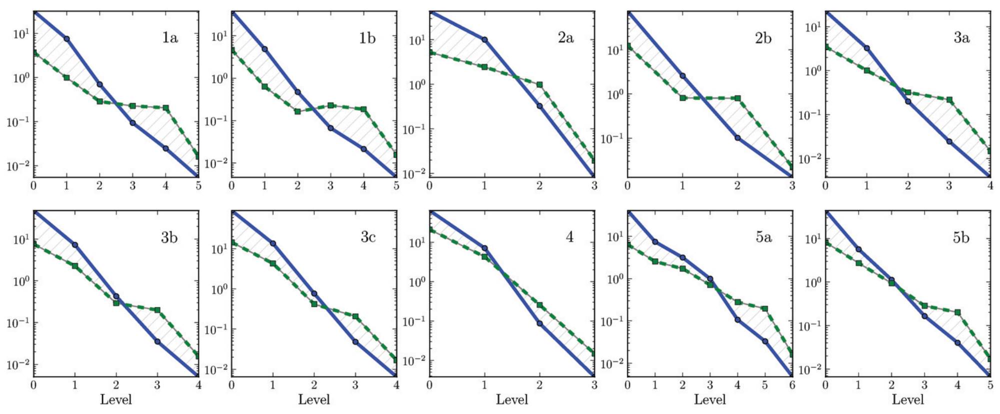
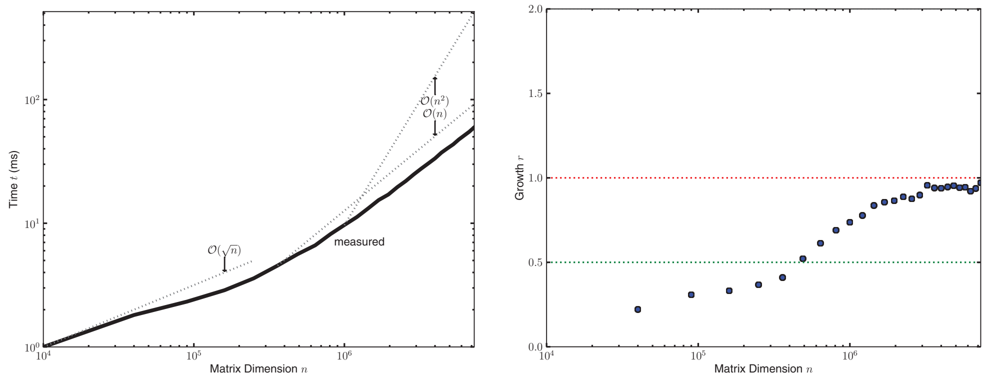


FIG. 5.5. *CPU (blue) vs. GPU (green) solve phase time (ms) per level.*



(a) Measured setup time in comparison to $\mathcal{O}(\sqrt{n})$, $\mathcal{O}(n)$, $\mathcal{O}(n^2)$.

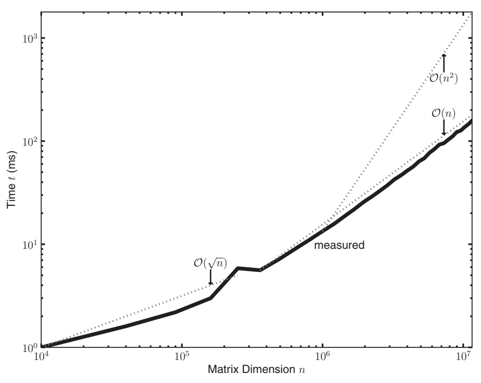
(b) Approximate rate of growth r in $\mathcal{O}(n^r)$.

FIG. 5.6. *Two-dimensional structured setup scalability test.*

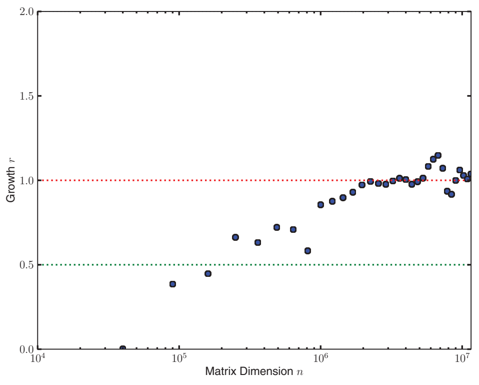
the GPU cycle is faster overall. However, the same is not necessarily true of other multigrid cycles, such as the F- or W-cycle, which visit coarse grids more frequently than fine grids.

5.6. Scalability. One important feature of a successful multigrid method is the ability to scale linearly with matrix size n. Consequently, we expect our algorithm to exhibit this scaling in observed setup and solve times. Here we consider example matrix 1a in Table 5.6—i.e., the two-dimensional structured Poisson problem—where AMG is known to scale linearly. The problem size is scaled equally in each coordinate direction, and we measure the wall clock time of both the setup and the solve phase.

The results are highlighted in Figures 5.6 and 5.7. As shown in Figures 5.6(a) and 5.7(a), we observe sublinear scaling before the GPU is saturated and the expected linear scaling after the problem size is sufficiently large. The fortuitous sublinear growth is consistent with other algorithms that utilize primitives on the GPU; e.g., see [28]. Indeed, as depicted in Figures 5.6(b) and 5.7(b), if we measure the growth rate r of time dependence $\mathcal{O}(n^r)$ over a window of five samples, the scaling settles at a linear relationship.



(a) Measured solve time in comparison to $\mathcal{O}(\sqrt{n})$, $\mathcal{O}(n)$, $\mathcal{O}(n^2)$.



(b) Approximate rate of growth r in $\mathcal{O}(n^r)$.

FIG. 5.7. *Two-dimensional structured solve scalability test.*

6. Conclusions. We have demonstrated the first implementation of AMG that exposes fine-grained parallelism at all stages of the setup and cycling phases. In particular, we have described highly parallel methods for sparse matrix-matrix multiplication and aggregation. Furthermore, we have shown that parallel primitives are a viable substrate to describing complex sparse matrix operations and targeting the GPU.

Building upon earlier work in SpMV, we have demonstrated meaningful speedup in both the setup and cycling phases of the AMG solver on structured and unstructured problems. Whereas CPU implementations of major components such as SpMV, sparse matrix-matrix multiplication, and sparse matrix transposition derive little or no benefit from multithreading, our implementations leverage scalable parallel primitives.

Appendix. Distance-*k* maximal independent sets. In this section we describe an efficient parallel algorithm for computing distance-k maximal independent sets, denoted MIS(k) and defined in Definition 3.1. We begin with a discussion of the standard distance-1 maximal independent set—i.e., MIS(1)—and then detail the generalization to arbitrary distances. Our primary interest is in computing a MIS(2) to be used in the parallel aggregation scheme discussed in section 3.2.

Computing a distance-1 maximal independent set is straightforward in a serial setting, as shown by Algorithm 4. The algorithm is greedy and iterates over nodes, labeling some as MIS nodes and their neighbors as non-MIS nodes. Specifically, all nodes are initially candidates for membership in the maximal independent set s and labeled (with value 0) as undecided. When a candidate node is encountered it is labeled (with value 1) as a member of the MIS, and all candidate neighbors of the MIS node are labeled (with value -1) as being removed from the MIS. Upon completion, the candidate set is empty and all nodes are labeled with either a -1 or 1.

Computing maximal independent sets in parallel is challenging, but several methods exist. With $k = 1$, our parallel version in Algorithm 5 can be considered a variant of Luby's method [27], which has been employed in many codes such as ParMETIS [24]. A common characteristic of such schemes is the use of randomization to select independent set nodes in parallel.

As with the serial method, all nodes are initially labeled (with a 0) as a candidate for membership in the MIS s. Additionally, each node is assigned a random value in

ALGORITHM 4. MIS_serial.

```
parameters:   $A$,   $n \times n$ sparse matrix
return:   $s$,   independent set

$I = \{0, \dots, n-1\}$                                {initial candidate index set}
$s \leftarrow 0$                                       {initialize to undecided}
for $i \in I$                                          {for each candidate}
    if $s_i = 0$                                       {if unmarked...}
        $s_i = 1$                                      {add to candidate set}
        for $j$ such that $A_{ij} \neq 0$
            $s_j = -1$                                 {remove neighbors from candidate set}
$s = \{i : s_i = 1\}$                                  {return a list of MIS nodes}
```

ALGORITHM 5. MIS_parallel.

```
parameters: $A$, $n \times n$ sparse matrix; $k$, edge distance
return:   $s$,   independent set

$I = \{0, \dots, n-1\}$
$s \leftarrow 0$                                       {initialize state as undecided}
$v \leftarrow$random                                   {initialize value}
while $\{i \in I : s_i = 0\} \neq \emptyset$
    for $i \in I$                                      {for each node in parallel}
        $T_i \leftarrow (s_i, v_i, i)$                 {set tuple (state,value,index)}
1   for $r = 1, \dots, k$                              {propagate distance $k$}
        for $i \in I$                                  {for each node in parallel}
            $t \leftarrow T_i$
            for $j$ such that $A_{ij} \neq 0$
                $t \leftarrow \max(t, T_j)$            {maximal tuple among neighbors}
            $\hat{T}_i \leftarrow t$
        $T = \hat{T}$
2   for $i \in I$                                      {for each node in parallel}
        $(s_{\max}, v_{\max}, i_{\max}) \leftarrow T_i$
        if $s_i = 0$                                   {if unmarked...}
            if $i_{\max} = i$                          {if maximal...}
                $s_i \leftarrow 1$                     {add to set}
            else if $s_{\max} = 1$                     {otherwise...}
                $s_i \leftarrow -1$                    {remove from set}
$s = \{i : s_i = 1\}$                                  {return a list of MIS nodes}
```

the array v. The purpose of the random value is to create disorder in the selection of nodes, allowing many nodes to be added to the independent set at once. Specifically, the random values represent the precedence in which nodes are considered for membership in the independent set. In the serial method this precedence is implicit in the graph ordering. Figure A.1 illustrates a two-dimensional graph with random values values drawn from integers in $[0, n = 36)$.

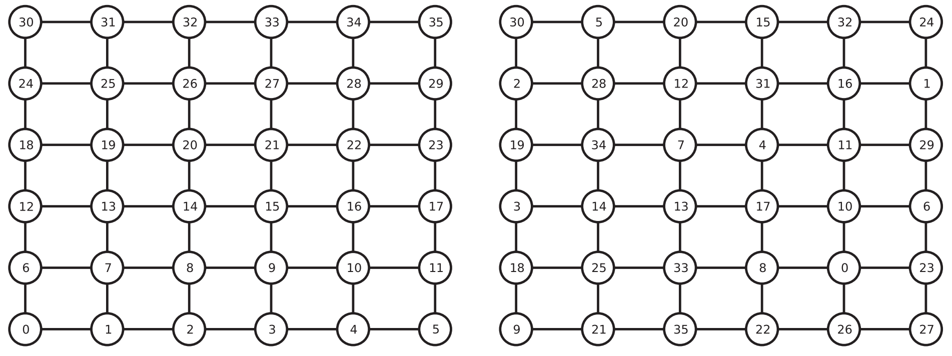

(a) Natural enumeration. (b) Random enumeration.

Fig. A.1. *A structured graph with a natural order of nodes and a randomized enumeration.*

The algorithm iterates until all nodes have been labeled with a -1 or 1, classifying them as a non-MIS node or a MIS node, respectively. In each iteration, an array T of 3-tuples is created, tying together the node state—i.e., -1, 0, or 1—the node's random value, and the linear index of the node. In a second phase, the nodes compute, in parallel, the maximum tuple among the neighbors. Given two tuples $t_i = (s_i, v_i, i)$ and $t_j = (s_j, v_j, j)$ the maximum is determined by a lexicographical ordering of the tuples. This ordering ensures that MIS nodes have a priority over candidate nodes and that candidate nodes have priority over non-MIS nodes. In the third phase, the candidate node states are updated based on the results of the second phase. Candidate nodes that are the local maximum—i.e., $I_{\max} = i$—are added to the independent set, while those with an existing MIS neighbor—i.e., $I_{\max} \neq i$ and $s_{\max} = 1$—are removed from candidacy. Since it is impossible for two neighboring nodes to be local maxima, the selected nodes are independent by construction. Furthermore, the correctness of the algorithm does not depend on the random values. Indeed, if all the random values are 0, the algorithm degenerates into the serial algorithm since the third component of the tuple, the node index, establishes precedence among neighbors with the same random value. In each iteration of the algorithm at least one candidate node's state is changed, so termination is ensured. Figure A.2 illustrates the classification of nodes during six iterations of the parallel algorithms.

Note that the parallelism in Algorithm 5 is fine-grained, exposing one thread of independent execution per node within each parallel for loop. The first and third phases are simple elementwise vector operations which are carried out with the `transform` algorithm in our implementation. The second stage, which propagates the maximal tuples, is implemented with a generalization of SpMV. In either case there is an implicit global synchronization among threads upon exiting the parallel for loop over nodes.

Generalizing the parallel distance-1 maximal independent set algorithm to compute MIS(k) for arbitrary k can be accomplished in several ways. One solution is to compute the kth power of A as an explicit matrix $A^K = A * A * \ldots A$ using sparse matrix-matrix multiplication, and then to apply the standard distance-1 maximal independent set algorithm. However, computing A^k explicitly is computationally expensive and the storage for A^k grows rapidly with k. An alternative approach, which is generally superior, uses $k > 1$ in Algorithm 5 so that maximum tuples are propagated k times using A, which has the same effect as one step of A^k. Line 1

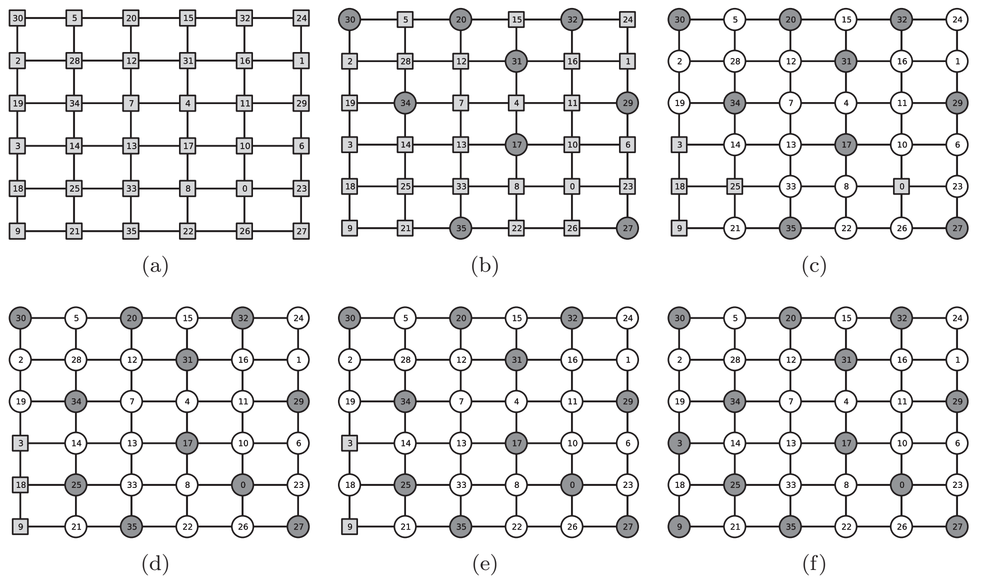


FIG. A.2. *Parallel MIS construction for $k = 1$.*

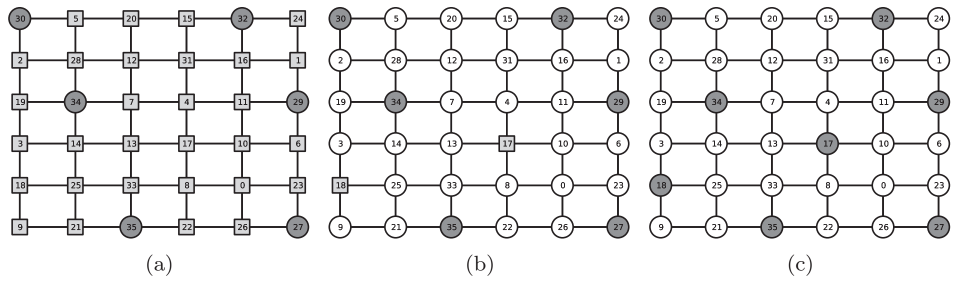


FIG. A.3. *Parallel MIS construction for $k = 2$.*

of Algorithm 5 illustrates this scheme with an additional optimization in the form of a second state-update pass. The second update pass on line 2 exploits the fact that many nodes can be immediately classified based on the results of the first pass without another iteration. Specifically, nodes whose maximum neighbor was classified as an independent set node—i.e., $s_{n_{\max}} = 0$—can be safely removed from candidacy. This optimization generally reduces the number of outer iterations by anticipating and efficiently applying the effect of the *next* iteration. An example of distance-2 MIS is depicted in Figure A.3.

In our implementation of Algorithm 5 the random values are produced by a hash function, $v_i = \texttt{hash}(i)$, where `hash` is a simple integer hash function. Although not a source of high-quality random numbers, the resulting values are adequate for our purpose. More sophisticated hash-based random number generators are discussed in [36, 40].

Figure A.4 shows the results of an empirical test of our MIS(2)-based aggregation algorithm. We compare the convergence rate of a solver constructed with the standard serial aggregation algorithm against the convergence rate of a solver constructed with our parallel aggregation algorithm when applied to a small two-dimensional Poisson

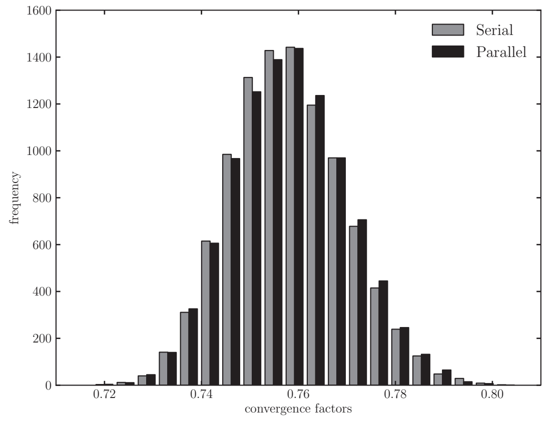


Fig. A.4. *Distribution of convergence factors for serial and parallel aggregation methods.*

problem. In each trial the rows and columns of the matrix are permuted randomly using a high-quality pseudorandom number generator, and a two-level hierarchy is constructed from using the serial and parallel aggregation schemes on the permuted matrix. Randomly permuting the matrix has the effect of randomizing the order in which the sequential aggregation algorithm visits nodes.

The close agreement of the two distributions offers empirical evidence that our hash-based randomization method is adequate for the purpose of computing aggregates in parallel. Indeed, the average (mean) convergence rates of the serial and parallel methods are 0.7582 and 0.7580, respectively.

REFERENCES

[1] M. Adams, M. Brezina, J. Hu, and R. Tuminaro, *Parallel multigrid smoothing: Polynomial versus Gauss-Seidel*, J. Comput. Phys., 188 (2003), pp. 593–610.

[2] A. H. Baker, T. Gamblin, M. Schulz, and U. M. Yang, *Challenges of scaling algebraic multigrid across modern multicore architectures*, in Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium, 2011.

[3] R. E. Bank and C. C. Douglas, *Sparse matrix multiplication package (SMMP)*, Adv. Comput. Math., 1 (1993), pp. 127–137.

[4] M. M. Baskaran and R. Bordawekar, *Optimizing Sparse Matrix-Vector Multiplication on GPUs*, Research report RC24704, IBM, 2009.

[5] N. Bell and M. Garland, *Efficient Sparse Matrix-Vector Multiplication on CUDA*, Technical report NVR-2008-004, NVIDIA Corporation, 2008.

[6] N. Bell and M. Garland, *Implementing sparse matrix-vector multiplication on throughput-oriented processors*, in Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, New York, ACM, 2009, pp. 18:1–18:11.

[7] N. Bell and M. Garland, *CUSP: Generic Parallel Algorithms for Sparse Matrix and Graph Computations*, http://code.google.com/p/cusp-library (2009).

[8] G. E. Blelloch, *Vector Models for Data-Parallel Computing*, MIT Press, Cambridge, MA, 1990.

[9] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder, *Sparse matrix solvers on the GPU: Conjugate gradients and multigrid*, ACM Trans. Graph., 22 (2003), pp. 917–924.

[10] E. Chow, R. D. Falgout, J. J. Hu, R. S. Tuminaro, and U. M. Yang, *A survey of parallelization techniques for multigrid solvers*, Parallel Processing for Scientific Computing, SIAM, Philadelphia, 2006, pp. 179–202.

[11] M. Christen, O. Schenk, and H. Burkhar, *General-purpose sparse matrix building blocks using the NVIDIA CUDA technology platform*, in Proceedings of the First Workshop on General Purpose Processing on Graphics Processing Units, Northeastern University, Boston, MA, 2007.

[12] A. J. Cleary, R. D. Falgout, V. E. Henson, J. E. Jones, T. A. Manteuffel, S. F. McCormick, G. N. Miranda, and J. W. Ruge, *Robustness and scalability of algebraic multigrid*, SIAM J. Sci. Comput., 21 (2000), pp. 1886–1908.

[13] J. M. Cohen and M. J. Molemaker, *A fast double precision CFD code using CUDA*, in Proceedings of the 21st International Conference on Parallel Computational Fluid Dynamics, 2009.

[14] *Cublas Library*, Version 3.1, NVIDIA Corporation, 2010, http://developer.nvidia.com/cublas.

[15] E. F. DAzevedo, M. R. Fahey, and R. T. Mills, *Vectorized sparse matrix multiply for compressed row storage format*, in Proceedings of the International Conference on Computational Science, Springer, New York, 2005, pp. 99–106.

[16] M. Garland and D. B. Kirk, *Understanding throughput-oriented architectures*, Commun. ACM, 53 (2010), pp. 58–66.

[17] M. W. Gee, C. M. Siefert, J. J. Hu, R. S. Tuminaro, and M. G. Sala, *ML* 5.0 *Smoothed Aggregation User's Guide*, Technical report SAND2006-2649, Sandia National Laboratories, Livermore, CA, 2006.

[18] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. S. McCormick, H. Wobker, C. Becker, and S. Turek, *Using GPUs to improve multigrid solver performance on a cluster*, Int. J. Comput. Sci. Engrg., 4 (2008), pp. 36–55.

[19] N. Goodnight, G. Lewin, D. Luebke, and K. Skadron, *A multigrid solver for boundary value problems using programmable graphics hardware*, in HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, Aire-la-Ville, Switzerland, Eurographics Association, 2003, pp. 102–111.

[20] F. G. Gustavson, *Two fast algorithms for sparse matrices: Multiplication and permuted transposition*, ACM Trans. Math. Softw., 4 (1978), pp. 250–269.

[21] G. Haase, M. Liebmann, G. Plank, and C. Douglas, *Parallel algebraic multigrid on general purpose GPUS*, in Proceedings of the 3rd Austrian Grid Symposium, J. Volkert et al., ed., 2010, pp. 28–37.

[22] V. E. Henson and U. M. Yang, *Boomerang: A parallel algebraic multigrid solver and preconditioner*, Appl. Numer. Math., 41 (2002), pp. 155–177.

[23] J. Hoberock and N. Bell, *Thrust: A Parallel Template Library, Version* 1.4.0, 2011, http://thrust/github.com.

[24] G. Karypis and V. Kumar, *Parallel multilevel k-way partitioning scheme for irregular graphs*, SIAM Rev., 41 (1999), pp. 278–300.

[25] M. Kazhdan and H. Hoppe, *Streaming multigrid for gradient-domain operations on large images*, in Proceedings of SIGGRAPH '08, New York, ACM, 2008, pp. 21:1–21:10.

[26] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, *Basic linear algebra subprograms for Fortran usage*, ACM Trans. Math. Software, 5 (1979), pp. 308–323.

[27] M. Luby, *A simple parallel algorithm for the maximal independent set problem*, SIAM J. Comput., 15 (1986), pp. 1036–1055.

[28] D. G. Merrill and A. S. Grimshaw, *Revisiting Sorting for GPGNU Stream Architectures*, Technical report CS2010-03, Department of Computer Science, University of Virginia, Charlottesville, VA, 2010.

[29] D. Merrill and A. Grimshaw, *Parallel Scan for Stream Architectures*, Technical report CS2009-14, Department of Computer Science, University of Virginia, Charlottesville, VA, 2009.

[30] *NVIDIA CUDA Programming Guide*, version 4.0, NVIDIA Corporation, 2011, http://developer.nvidia.com/cuda.

[31] L. N. Olson, J. Schroder, and R. S. Tuminaro, *A new perspective on strength measures in algebraic multigrid*, Numer. Linear Algebra Appl., 17 (2010), pp. 713–733.

[32] J. W. Ruge and K. Stüben, *Algebraic multigrid*, in Multigrid Methods, Front. Appl. Math. 3, SIAM, Philadelphia, 1987, pp. 73–130.

[33] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, *Scan primitives for GPU computing*, in Proceedings of Graphics Hardware 2007, ACM, 2007, pp. 97–106.

[34] M. Stürmer, H. Köstler, and U. Rüde, *How to optimize geometric multigrid methods on GPUS*, in Proceedings of the 15th Copper Mountain Conference on Multigrid Methods, 2011.

[35] R. S. Tuminaro and C. Tong, *Parallel smoothed aggregation multigrid : Aggregation strategies on massively parallel machines*, in Proceedings of the Supercomp Conference, 2000, p. 5.

[36] S. Tzeng and L.-Y. Wei, *Parallel white noise generation on a GPU via cryptographic hash*, in Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games, ACM, 2008, pp. 79–87.

[37] P. Vaněk, J. Mandel, and M. Brezina, *Algebraic Multigrid by Smoothed Aggregation for Second and Fourth Order Elliptic Problems*, Computing, 56 (1996), pp. 179–196.

[38] R. W. Vuduc and H.-J. Moon, *Fast sparse matrix-vector multiplication by exploiting variable block structure*, in Proceedings of the High Performance Computing and Communications: First International Conference, HPCC 2005, Sorrento, Italy, 2005.

[39] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, *Optimization of sparse matrix-vector multiplication on emerging multicore platforms*, in Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, 2007.

[40] F. Zafar, M. Olano, and A. Curtis, *GPU random numbers via the tiny encryption algorithm*, in Proceedings of the Conference on High Performance Graphics, Eurographics Association, 2010, pp. 133–141.