Porting *hypre* to Heterogeneous Computer Architectures: Strategies and Experiences⋆

Robert D. Falgout[a], Ruipeng Li[a,*], Björn Sjögreen[a], Lu Wang[b] and Ulrike Meier Yang[a]

[a]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, P.O. Box 808, L-561, Livermore, CA 94551.
[b]Databricks Inc., 160 Spear Street, San Francisco, CA 94105

ARTICLE INFO

Keywords:
Multigrid methods
Iterative methods
Preconditioning techniques
GPU computing
HPC

ABSTRACT

Linear systems are occurring in many applications, and solving them can take a large amount of the total simulation time. The high performance library *hypre* provides a variety of interfaces and linear solvers, including various multigrid methods, that have achieved good scalability on a variety of homogeneous parallel computer architectures. Heterogeneous architectures with nodes that have both CPUs and accelerators provide new challenges, since they require more fine-grained parallelism and reduced data movement between different memories on a single node as well as across nodes. We will discuss our experiences and strategies to port *hypre* to heterogeneous computers with accelerators, including the design of a new memory model, the use of abstractions, the BoxLoop macros in the structured and semi-structured interfaces, and the restructuring of algebraic multigrid (AMG) into modular components. We present numerical experiments comparing CPU and GPU performance for several test problems.

1. Introduction

Linear systems occur in many scientific applications, and their solution can take a substantial amount of the total simulation time. The parallel software library *hypre* [33, 28, 25] provides a variety of conceptual interfaces, linear solvers and preconditioners that are designed to achieve high performance. The interfaces allow users to create a linear system in the way most convenient for them, be it via grids and stencils, finite elements or as a linear-algebraic system. Underneath these interfaces different data structures are created that aim to take advantage of the way the problem has been defined to achieve better performance, when possible. For example, a system expressed via the structured (Struct) interface, in terms of grids and stencils, can be solved using a more efficient solver that takes advantage of the structure. The semi-structured (SStruct) interface is designed to solve mostly structured problems, coming from block-structured, overset, or adaptive mesh refinement grids and allows the use of additional variable types, including edge-centered, and face-centered variables. It also provides a finite element option. Finally, the IJ interface, which takes a linear-algebraic system as input, gives access to more general, albeit more expensive solvers. The multigrid preconditioners and solvers in *hypre* have shown excellent scalability on homogeneous computer architectures [7]. Heterogeneous architectures with nodes that have both CPUs and accelerators provide new challenges, since they require more fine-grained parallelism and reduced communication and data movement, which are significantly more costly than computational operations on these new computer architectures.

To respond to these difficulties *hypre* developers reevaluated their software strategy and started to implement various changes. To accommodate the fact that CPUs and GPUs each have their own memories, a new memory model was developed that specifies the memory in which data are allocated. However, further changes were needed to port the solvers in *hypre* to GPUs. Many *hypre* solvers are algebraic multigrid methods (AMG) [14, 50, 26], which can provide excellent numerical scalability, when they are well designed. AMG differs from geometric multigrid in that it derives the lower levels completely from the linear system at the finest level, $A_1 x = b$, provided by the user. It consists of a setup and a solve phase. During the setup phase, the variables for level $l+1$ are determined by coarsening the graph of A_l, followed by the creation of interpolation P_l, restriction R_l and the coarse grid operator $A_{l+1} = R_l A_l P_l$. Often $R_l = P_l^{\mathsf{T}}$. During the solve phase, the error is smoothed with a few sweeps of a generally simple iterative method such as Jacobi or Gauss-Seidel, and R_l and P_l are used to move between levels. GPUs generally achieve excellent performance on very large problems. This naturally constitutes an issue for multilevel methods, which, on coarser levels, have successively smaller systems that require decreasing numbers of computational operations, but still a large amount of communication. Neverthless, AMG methods are still among the best candidates to achieve good runtimes, even on GPUs, due to their numerical scalability and optimality.

Since there are significant differences between the data structures used from the Struct and IJ interface, different software strategies were developed to port the underlying solvers to GPUs. Since GPUs favor structured operations, *hypre*'s structured solvers, which are based on grids and stencils, were promising candidates for good GPU performance. They were designed with BoxLoops, macros that are used to perform loops across the underlying data structures. This portable design enables the addition of new options, includ-

*Corresponding author
falgout2@llnl.gov (R.D. Falgout); li50@llnl.gov (R. Li); sjogreen2@llnl.gov (B. Sjögreen); yang11@llnl.gov (U.M. Yang)
ORCID(s): 0000-0003-4884-0087 (R.D. Falgout); 0000-0003-2802-5763 (R. Li); 0000-0002-0927-4329 (B. Sjögreen); 0000-0002-6957-0445 (U.M. Yang)

ing new programming models. A different strategy was required to port the unstructured AMG solver to GPUs. We decided on a modular approach, i.e. expressing the algorithms in terms of smaller reusable kernels, wherever possible. Such kernels include matrix and vector operations. This approach reduces the amount of code that needs to be rewritten when porting to a new architecture. The current kernels have been written in CUDA for Nvidia GPUs, but work is in progress to port them to other programming models, such as HIP and SYCL for AMD and Intel GPUs, using vendor-provided conversion tools. While this approach was fairly straightforward for the solve phase, which can completely be expressed in terms of matrix-vector operations, it required new algorithm design for the setup phase, which now can be expressed mostly in terms of matrix operations.

This paper describes the new memory model in Section 2. Section 3 focuses on the efforts to port the structured interface and solvers to various new programming models. Section 4 discusses the algorithms used to achieve GPU performance in the IJ interface, including the generation of IJ matrices, and various kernels and algorithms used in the setup and solve phase of AMG. Section 5 presents additional GPU-enabled components as well as interfaces and solvers, not all of which are available on GPUs yet, but are expected to be ported soon. Section 6 contains numerical results for various test problems obtained on a machine at LLNL with 2 IBM Power 9s and 4 Nvidia V100s per node. We conclude with future plans in Section 7.

2. Heterogeneous Memory Management

In this section, we talk about the memory management model for the heterogeneous CPU-GPU platform and the execution policy based on the memory model.

2.1. Conceptual and physical memory spaces

The conceptual memory space of *hypre* consists of two user-level memory locations, namely `HYPRE_MEMORY_HOST` and `HYPRE_MEMORY_DEVICE`. The physical memory space has four locations, i.e., `hypre_MEMORY_HOST`, `hypre_MEMORY_HOST_PINNED`, `hypre_MEMORY_DEVICE`, and `hypre_MEMORY_UNIFIED`, which are, respectively, CPU memory, CPU pinned memory, GPU device memory and unified memory, and not exposed to the users. Moreover, the pinned host memory is internally used to transfer data between the device and the host, mostly for the MPI communications issued from the host. The mapping between the conceptual memory space to the physical memory space depends on the configurations for different architectures. In the simplest case for a CPU-only build, both `HYPRE_MEMORY_HOST` and `HYPRE_MEMORY_DEVICE` are mapped to `hypre_MEMORY_HOST`. In Table 1, we present the mappings in three different scenarios, where "CPU-only" is the CPU-only build, "GPU without UVM" is the GPU build without unified memory, and "UVM" is the GPU build with unified memory enabled.

Table 1
Mappings from the conceptual memory space to the physical memory space in 3 different configurations of *hypre*.

	`HYPRE_MEMORY_HOST`	`HYPRE_MEMORY_DEVICE`
CPU-only	`hypre_MEMORY_HOST`	`hypre_MEMORY_HOST`
GPU w/o UVM	`hypre_MEMORY_HOST`	`hypre_MEMORY_DEVICE`
UVM	`hypre_MEMORY_HOST`	`hypre_MEMORY_UNIFIED`

2.2. Memory location and execution policy

For the unstructured objects such as the sparse matrices and vectors, a `memory_location` member is introduced in the class to identify the (conceptual) memory location the object currently resides on. Therefore, appropriate host or device functions can be called based on this information of the input object. We refer to the execution location of a function that depends on the memory location of an object as the *execution policy*. Obviously, the execution policy for `hypre_MEMORY_HOST` and `hypre_MEMORY_DEVICE` has no ambiguities, whereas , since unified memory can be accessed from both the host and device, the policy for `hypre_MEMORY_UNIFIED` can be either `HYPRE_EXEC_HOST` or `HYPRE_EXEC_DEVICE` and is defaulted as the former one.

3. Structured Interface and Solvers

In this section, we discuss the portability strategy for the structured interface and solvers in *hypre*.

3.1. Structured interface

The Struct interface in *hypre* is a way of describing linear systems in terms of structured grids and stencils instead of matrix rows and columns. Systems of this type often arise from discretizations of partial differential equations, so the Struct interface provides a more natural interface for these applications. It is also a means of expressing structure in the system that can be exploited to deliver more efficient solvers and computational kernels. The latter is potentially of great benefit on accelerator-based architectures like GPUs.

The grid is described via a global *index space* of integer tuples in 2D as depicted in Figure 1 (1D, 3D, etc. are analogous). The global indexes allow *hypre* to discern how data is related spatially, and how it is distributed across the parallel machine. The basic component of the grid is a *box*, a collection of cell-centered indices defined by its minimum and maximum indexes (the lower-left and upper-right corners in the figure). Each process owns a unique subset of the grid boxes. A vector has values associated with indexes on the grid. A matrix has rows associated with indexes on the grid, where nonzero entries of each row are related to "neighboring" indexes by way of a *stencil*. For example, the following is a five-point stencil commonly used to discretize diffusion equations:

$$\begin{bmatrix} & (0,1) & \\ (-1,0) & (0,0) & (1,0) \\ & (0,-1) & \end{bmatrix}. \tag{1}$$

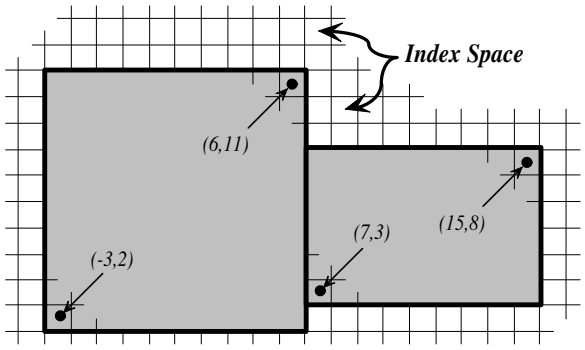


Figure 1: A two-dimensional structured grid is a union of non-intersecting boxes defined on a global index space of integer tuples (i, j).

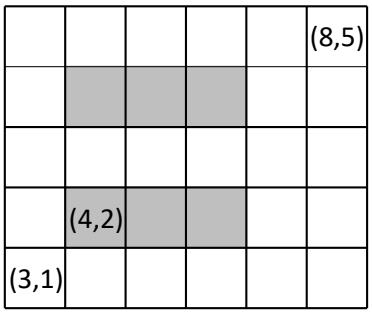


Figure 2: Illustration of a two-dimensional BoxLoop with `loopsize` $= (3, 2)$, `databox` $= (3, 1) \times (8, 5)$, `start` $= (4, 2)$, and `stride` $= (1, 2)$. In this example, the loop generates array indexes 8, 9, 10, 20, 21, and 22, in no predefined order.

With this definition of matrices and vectors, linear algebra operations have regularity that can be exploited for computational efficiency. The core kernels are defined on boxes of data, so *hypre* employs a loop abstraction called a BoxLoop. This abstraction is essentially an iterator that generates indexes into arrays of data. In *hypre*, it is implemented with macros and has the following structure:

```
hypre_BoxLoop1Begin(ndim, loopsize,
                    databox, start, stride, ii);
{
   yp[ii] += alpha * xp[ii];
}
hypre_BoxLoop1End(ii);
```

For a 2D problem, `ndim` is 2, `loopsize` is an integer array of length 2 with the loop size in each dimension, `databox` is the lower and upper corners of a box of data, `start` is the starting index (in the `databox`) for the loop, `stride` is the stride, and `ii` is the index into the data arrays `xp` and `yp`. There are several BoxLoop macros in *hypre*. For example, the `BoxLoop2` macro generates two array indexes for data with two different data-box layouts.

On a CPU-based node, the `hypre_BoxLoop1Begin` macro precomputes various quantities needed for the looping construct and starts the loop. The code between the following braces is arbitrary. The above example illustrates a scaled vector sum or "axpy" on array data `xp` and `yp` that have the same data-box layout. The `hypre_BoxLoop1End` macro increments the array index `ii` and closes the loop. Figure 2 shows an example of the array indexes generated by the BoxLoop for a given set of loop parameters.

This BoxLoop loop abstraction simplifies the process of porting *hypre*'s Struct code to GPUs and enables a variety of different approaches including: OpenMP, CUDA, RAJA, and Kokkos. The implementation of these ports is largely restricted to header file code that defines the various BoxLoop macros. In the following sections, we will desribe the main components of each of the ports.

3.2. OpenMP

OpenMP versions 4.5 and later extend support to GPU-based architectures. Prior to this, *hypre* used OpenMP to provide a multi-threaded shared-memory approach for utilizing on-node parallelism. The original approach placed OpenMP pragmas prior to each BoxLoop in the C source files. Since the syntax of these pragmas changed somewhat for GPU-based architectures, a decision was made to require C99 standard compilers (or later) when using OpenMP and rework the BoxLoop macros to use the `_Pragma` and variadic macro features (the equivalent `__pragma` feature is used for Microsoft C/C++ compilers). This change involved touching most of the Struct code in *hypre*, but was fairly straightforward. Once in place, the port to OpenMP4.5 mainly involved changing the pragma line in the BoxLoop macros. The original CPU pragma was

```
_Pragma("omp parallel for ...")
```

The "..." part of the pragma specifies certain thread-private variables and the parallel-for schedule, which is currently set to `schedule(static)`. The OpenMP4.5 pragma is

```
_Pragma("omp target teams distribute parallel for ...")
```

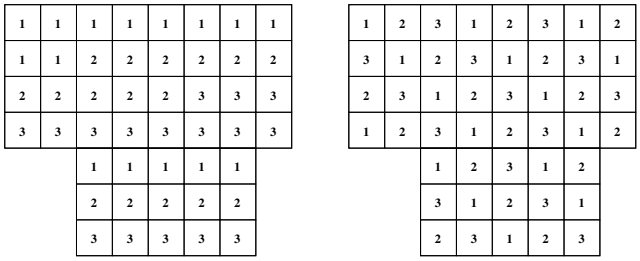
The "..." part of the pragma specifies various OpenMP clauses such as `is_device_ptr()` to indicate pointers to memory allocated on device (e.g., the `xp` and `yp` pointers in the above BoxLoop example).

3.3. CUDA

The CUDA implementation of the BoxLoops relies on the lambda function features of C++11 (and later) to package the loop body into a function that can be launched as a CUDA kernel. Because of the C++11 requirement, a significant effort was required to modify the C code in *hypre* so that it can be compiled with a C++ compiler (although C is essentially a subset of C++, this is not strictly true). The changes needed were straightforward, but tedious, and affected about 300 files in the code. Here is a list of the main changes that needed to be made: changed '`char *`' to '`const char *`' anywhere string literals were expected; added

explicit cast expressions from '`void *`' variables; added argument types to all function declarations; added '`extern "C"`' guards around Fortran interface code and many other places; changed certain variable names like '`new`', '`true`', '`false`' that are reserved words in C++.

A major difference in the parallel BoxLoop implementations on CPUs and GPUs are the ways to partition boxes for different threads. In the CPU implementation, boxes are partitioned into segments of cells that are consecutive in memory for better cache locality, whereas in the GPU implementation, boxes are partitioned in a "cell-wise interleaved" way, i.e., thread 1 accesses cell 1, thread 2 accesses cell 2, and so on, so that adjacent threads can access adjacent cells in memory to increase the DRAM bandwidth for better memory coalescing. See Figure 3 for the illustrations of these two types of approaches for partitionings two boxes.



(a) partition boxes on CPUs (b) partition boxes on GPUs

Figure 3: The partitioning of two boxes on CPUs and GPUs. The numbers in the cells are thread indices.

3.4. RAJA

RAJA [9] is a collection of C++ software abstractions, including looping abstractions, designed to support performance portability across high-performance computing platforms, including GPU-based systems. Like with CUDA, the RAJA implementation of the BoxLoops also requires C++11 (or later) and the use of lamba functions to package the loop body into a function. The lambda function is then passed into the `RAJA::forall` loop execution method.

3.5. Kokkos

Kokkos [17] implements a programming model in C++ for writing performance portable applications. Similar to RAJA, it also provides parallel looping abstractions. The BoxLoop implementation for Kokkos in *hypre* is basically the same as for RAJA, requiring C++11 and lambda functions, but using the `Kokkos::parallel_for` looping construct instead.

3.6. Structured solvers

The *hypre* library provides two structured semicoarsening multigrid solvers, SMG [15] and PFMG [5], which use different relaxation approaches. SMG uses plane smoothing, which leads to a very robust, but potentially expensive algorithm. PFMG, on the other hand, uses pointwise smoothing, making this method highly efficient for suitable problems. Both methods utilize the BoxLoops in the implementation of their setup and solve phases. This software strategy leads to good portability and allows straightforward GPU implementation.

4. Unstructured Interface and Solvers

In this section, we discuss our software strategy and efforts to enable the unstructured interface and linear algebra kernels, preconditioners and solvers to run on GPUs.

4.1. Setup and assembly of IJ Objects

The IJ interface provides access to parallel sparse matrices and vectors in *hypre*, which are distributed by blocks of rows. It is expected that users will provide data in distributed form because this is the only scalable approach for assembling matrices and vectors on thousands or millions of processes. After an IJ matrix has been created and initialized, i.e. the matrix is initially a zero matrix, the matrix coefficients are ready to be modified by repetitively calling the function

```
HYPRE_IJMatrixSetValues(ijmatrix, nrows, ncols,
                        rows, cols, values);
```

where `nrows` is the number of rows to set, `ncols` is an array of size `nrows` that contains the number of values to set in each row. The actual global row and column indices and values are given by `rows`, `cols` and `values`. One can also call the function `HYPRE_IJMatrixAddToValues`, which has the same prototype as above but adds `values` to the matrix instead of setting them. Note that while `AddToValues` can add to values on other processes, `SetValues` will only set values locally, since it is not possible to uniquely determine the order of setting values across processes. If `SetValues` is called from process i to set a matrix coefficient on a different process j, all occurrences of this coefficient on process i will be erased to avoid contributing to this coefficient on process j.

The unassembled matrix coefficients are stashed in an intermediate auxiliary object until `HYPRE_IJMatrixAssemble` is called to generate the final matrix. Although the GPU versions of the IJ routines have the same prototypes as the CPU versions, but expect the input data to be in GPU memory, the underlying data structure and algorithms are significantly different. In the CPU implementation, the data structure for the auxiliary matrix is a dynamic 2-D array, i.e., pointers to pointers, where the matrix coefficients are stored in separate arrays, one for each row, that are pre-allocated with a fixed small size (say, 30), and dynamically resized when the capacity is not enough. Since a matrix coefficient may appear multiple times in the same or different calls of `Set/AddToValues`, a search for the matrix coefficient is first done in the corresponding row of the auxiliary matrix, and the value is updated if it exists or a new coefficient is inserted otherwise.

It is very challenging to have an efficient implementation on GPUs, when using 2-D arrays, especially for large problems. This is caused not only by the large number of dynamic allocations and reallocations but also by the large

number of row searches. A more appropriate data structure and algorithm is obtained by storing all matrix coefficients across different rows in flat 1-D arrays, and postponing the *compression* of the entries that have the same row and column indices in the assembly. These 1-D arrays are preallocated with some arbitrary initial size, such as a small constant times the number of local rows, and reallocated by a prespecified growth factor if there is no more space. Thus, the potential number of memory allocations should be far less than that when using 2-D arrays. The compression in the assembly is performed by a *sort-scan-reduce* approach illustrated in Figure 4, where we consider a simple example of assembling the upper triangular part of the global stiffness matrix from two element matrices and the Dirichlet boundary condition on the edge $(2,3)$ on 1 process. As shown in Figure 4a, the row indices, column indices and numerical values of the unassembled matrix coefficients are stored in arrays `I`, `J` and `A` in the auxiliary matrix. Furthermore, a binary array `S` is used to distinguish the matrix coefficients obtained by `SetValues` from those of `AddToValues`, with values 1 and 0 respectively. Note that we do not need to store this information explicitly in the assembly algorithm on CPUs, since the matrix coefficients are sequentially applied so that the correct order of "set" and "add" is inherently guaranteed. The first step in the assembly is to *sort* the four arrays together using (`I`, `J`) as the tuple keys, in order to bring the coefficients at the same position of the matrix together. Note here that a *stable* sorting algorithm must be chosen to maintain the relative order of the entries with equal keys. In the second step, shown in Figure 4c, a segmented *scan*, i.e., a scan that is broken into distinct segments with the same (`I`, `J`) key, is performed on `S` to keep the last "set" of a matrix coefficient and discard all the previous occurrences of the same matrix coefficient. Clearly, the scan should be exclusive and performed in the reverse order (i.e., from the end to the beginning) and the reduction operation in the scan is to take the maximum. The meaning of binary array `S` has been changed after this step, such that numerical values in `A` that correspond to 1 in `S` need to be zeroed out, denoted by `0` as opposed to the original zeros, whereas those corresponding to 0 are kept unchanged. In the last step of the assembly, a segmented *reduction* is performed to combine the matrix coefficients in `A` that have the same (`I`, `J`) into one sum, and `S` is reduced accordingly by again taking the maximum. Notice that the array `S` defines the correct modifications to the pre-existing values in the previously assembled matrix. After that, the auxiliary matrix is ready to be converted to the final matrix format in *hypre* and then freed.

The example in Figure 4 mimics a common practice in assembling FEM matrices, where contributions from all elements are first added and those from boundary condition are set after. The IJ interface of *hypre* can actually handle more general usage with the same algorithm discussed above. Consider an *add-after-set* situation for entry (2,2) in Figure 5, where the values from the first two "adds" prior to the "set" are zeroed out in the reversed-scan step, while the value from the third "add" after the "set" can be correctly

I	0	0	0	2	2	3	0	0	0	1	1	2	0	1	2	0	2	3
J	0	3	2	2	3	3	0	1	2	1	2	2	2	2	2	3	3	3
A	.5	-.5	0	.5	-.5	1	.5	-.5	0	1	-.5	.5	0	0	1	0	0	1
S	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1

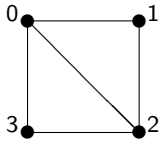
(a) coefficients of element matrices and boundary conditions

I	0	0	0	0	0	0	0	0	1	1	1	2	2	2	2	2	3	3
J	0	0	1	2	2	2	3	3	1	2	2	2	2	2	3	3	3	3
A	.5	.5	-.5	0	0	0	-.5	0	1	-.5	0	.5	.5	1	-.5	0	1	1
S	0	0	0	0	0	1	0	1	0	0	1	0	0	1	0	1	0	1

(b) sort by key

I	0	0	0	0	0	0	0	0	1	1	1	2	2	2	2	2	3	3
J	0	0	1	2	2	2	3	3	1	2	2	2	2	2	3	3	3	3
A	.5	.5	-.5	0	0	0	0	0	1	0	0	0	0	1	0	0	0	1
S	0	0	0	1	1	0	1	0	0	1	0	1	1	0	1	0	1	0

(c) reversed exclusive scan by key



I	0	0	0	0	1	1	2	2	3
J	0	1	2	3	1	2	2	3	3
A	1	-.5	0	0	1	0	1	0	1
S	0	0	1	1	0	1	1	1	1

(d) reduce by key

Figure 4: IJ matrix assembly for FEM stiffness matrix

I	2	2	2	2
J	2	2	2	2
A	.5	.5	1	.5
S	0	0	1	0

(a) sort-by-key

I	2	2	2	2
J	2	2	2	2
A	0	0	1	.5
S	1	1	0	0

(b) reversed scan

I	2
J	2
A	1.5
S	1

(c) reduce

Figure 5: Assemble entry (2,2) with *add-after-set* values

included in the final reduction.

When adding coefficients to other processes, a separate auxiliary matrix is used to store the off-process unassembled coefficients. At the assembly stage, this auxiliary matrix is first compressed by the same aforementioned steps with a slight difference in the scan-step where the segmented scan should be inclusive to invalidate the set to a value belonging to a different process, then transmitted to other processes, and lastly compressed again after being combined with the local auxiliary matrices on the remote processes.

The algorithm used for assembling IJ vectors on GPUs is very similar to the one for IJ matrices and is much simpler, so we omit the details. The parallel sort, scan, and reduction primitives on GPUs are available from the Thrust library [13].

4.2. Sparse linear algebra kernels

Exploiting optimized computational kernels is a common approach to improve performance and increase portability. In this section, we discuss several sparse linear alge-

bra kernels used in the AMG setup and solve phases.

The sparse matrix-vector product (SpMV) is generally the most frequently used kernel in the AMG solve phase. It is applied with the local matrices of coefficient matrix A_l, interpolation matrix P_l, and restriction matrix R_l at each level l. A great amount of work has been done in the literature to increase the performance of SpMV on GPUs, see, e.g., [12, 41, 40, 18, 4, 8] among many others, using different strategies such as designing advanced sparse matrix formats that are more suitable to GPUs, automated performance tuning, improving the load balance, and optimizing the memory bandwidth. In *hypre*, we use the SpMV kernel in the CSR format from the cuSPARSE library. The distributed SpMV with the transpose of a matrix can be computed as the sparse matrix-transpose-vector products (SpMTV). However, since the performance of SpMTV in the CSR format on GPUs is significantly slower than that of SpMV, it can be more efficient to explicitly compute and store the transpose of the matrix, avoid SpMTV and perform SpMV instead.

The performance of a sparse triangular solve (SpTrSV) is usually far lower compared with SpMV due to its inherently sequential nature. This kernel is required in AMG by the Gauss-Seidel relaxation and incomplete LU (ILU) factorization preconditioners. The parallelism in SpTrSV can be discovered by analyzing the directed acyclic graph (DAG) of the triangular matrix and carefully scheduling the order of the nodes to solve. The level-scheduling algorithm was introduced by Anderson and Saad [1], and later by Saltz in [49]. The works in [47, 35] might be the first efforts on GPUs and similar approaches were later adopted in [51, 48]. The element-based scheduling was introduced in [29] for shared-memory machines, and recently implemented on GPUs [38, 43, 37]. SpTrSV kernels based on these two scheduling algorithms are available in the cuSPARSE library. *hypre* can use the cuSPARSE kernels and, alternatively, provides its own implementation of SpTrSV. It was shown in [37] that *hypre*'s implementation can perform better than the cuSPARSE kernels.

The sparse matrix-matrix multiplication (SpGEMM) often represents the most challenging computational kernel in the AMG setup. It is required to construct coarse-grid operators in the Galerkin product in the form of RAP, to generate matrix-matrix based interpolation operators, and to compute S^2, described in Section 4.3. First consider the multiplication of distributed sparse matrices $Q = AP$, which can be decomposed into four local SpGEMMs, i.e.,

$$\begin{aligned} Q_{\text{diag}} &= A_{\text{diag}} P_{\text{diag}} + A_{\text{offd}} P^{\text{ext}}_{\text{diag}}, \\ Q_{\text{offd}} &= A_{\text{diag}} P_{\text{offd}} + A_{\text{offd}} P^{\text{ext}}_{\text{offd}}, \end{aligned} \tag{2}$$

where 'diag' and 'offd' indicate the on-process and the off-process blocks of the local matrix respectively, and P^{ext} is the external rows owned by other processes that are required in the multiplication. However, a more efficient way is to reorganize (2) into one SpGEMM,

$$\begin{bmatrix} Q_{\text{diag}} & Q_{\text{offd}} \end{bmatrix} = \begin{bmatrix} A_{\text{diag}} & A_{\text{offd}} \end{bmatrix} \begin{bmatrix} P_{\text{diag}} & P_{\text{offd}} \\ P^{\text{ext}}_{\text{diag}} & P^{\text{ext}}_{\text{offd}} \end{bmatrix}. \tag{3}$$

Similarly, for RQ where $R = P^{\mathsf{T}}$, the local multiplication is given by

$$\begin{bmatrix} R_{\text{diag}} \\ R_{\text{offd}} \end{bmatrix} \begin{bmatrix} Q_{\text{diag}} & Q_{\text{offd}} \end{bmatrix}, \tag{4}$$

where R_{diag} and R_{offd} are stored as transposes and should be saved for the solve phase. Moreover, the second block row of the product in (4) needs to be transmitted to the neighboring processes and combined with the local result (which is the first block row) on those processes.

Efficient algorithms for SpGEMM and their implementations on GPUs are typically complicated. In a row-based format such as CSR, the objective is essentially to find efficient approaches to accumulate sparse rows, which are often referred to as the *sparse accumulators*. Existing algorithms include expansion-sorting-compression (ESC) methods [44, 21], row-merging methods [39, 30, 11], heap-sort-based methods [39], and hash-based methods [3, 24, 46]. A state-of-the-art SpGEMM kernel in the CSR format is available from the cuSPARSE library, while the implementation in *hypre* based on [24] is generally found to have superior performance. The hash-table based algorithm in *hypre* for multiplying two general sparse matrices $Q = AP$ consists of the following 3 steps:

1. Stochastic estimation: estimate the number of nonzeros in each row of Q. This step is for allocating the hash tables of reasonable sizes used in step 2;
2. Symbolic multiplication: obtain the exact number of nonzeros in each row of Q or a tight upper bound. This step is for allocating adequate memory for Q that is computed in step 3;
3. Numeric multiplication: compute the sparsity pattern and the numerical values of Q.

In the first step, the stochastic reachability-set algorithm by Cohen [19] is used to estimate the structures of matrix products. It represents the product of sparse matrices as a network of layered bipartite graphs, and the desired estimates can be obtained as the output from the network with random samplings. This algorithm is efficient, in linear time in the number of nonzeros of the input matrices, and the row estimations are totally independent, which makes it appealing for GPUs. In the multiplication steps 2 and 3, a hash table is used for each row of the product to accumulate the sparse rows of P. The procedures in these two steps are essentially the same. In step 2, the goal is to only count the number of nonzeros, while the memory has not yet been allocated for Q. In step 3, the actual pattern is determined and the numerical values are accumulated.

To represent the hash table, we use arrays, or *direct address tables*, where the keys and data are stored separately. For the sparse row accumulator, the key is the column index and the data is the matrix value. The accumulated matrix coefficients are stored in the hash table, i.e., *open addressing*, where hash collisions are resolved by *probings* [20]. The CUDA implementation of the *hash-search-insert* operation is shown in Figure 6, where `HashSize` is the capacity of the

```
template <HYPRE_Int ProbType>
static __device__ __forceinline__ HYPRE_Int
hash_search_insert(HYPRE_Int              HashSize,
                   volatile HYPRE_Int     *HashKeys,
                   volatile HYPRE_Complex *HashVals,
                   HYPRE_Int              key,
                   HYPRE_Complex          val,
                   HYPRE_Int              &count) {
  for (HYPRE_Int i = 0; i < HashSize; i++) {
    HYPRE_Int j, k;
    j = HashFunc<ProbType>(HashSize, key, i);
    k = atomicCAS((HYPRE_Int *)(HashKeys+j), -1, key);
    if (k == -1 || k == key) {
      /* slot is empty or has 'key', update value */
      if (k == -1) { count++; }
      atomicAdd((HYPRE_Complex*)(HashVals+j), val);
      return j;
    }
  }
  return -1;
}
```

Figure 6: Parallel Hash-Search-and-Insert operation in CUDA

hash table represented by `HashKeys` and `HashVals`, tuple (`key`, `val`) is a new entry to insert, and `count` keeps the number of elements in the hash table. The values of `HashKeys` and `HashVals` are assumed to be initialized as -1 and 0, respectively. The function returns the slot index `j` where `key` was found or where (`key`, `val`) was inserted, or returns -1 if no slot was found, i.e., the hash table was full. At each step, a probed location is generated in the probe sequence which is determined by the hash function and the probing type. The atomic *compare-and-swap* function, `atomicCAS` in CUDA, is used to find either an empty slot, which means `key` is a new entry, or the slot that has the same key in the hash table. When such a slot can be found, the corresponding value in `HashVals` is modified by `val` using the atomic add function. The purpose of using the atomic functions is to allow simultaneous operations to the same hash table issued by different threads. Once all the matrix coefficients from different rows of P have been inserted, a final row of Q is readily available from the hash table. Furthermore, two hash tables are maintained, one is on the faster shared memory and the other is on the global memory. The global-memory hash table is needed when the shared-memory one overflows.

Finally, the overall SpGEMM algorithm is to independently compute the rows in parallel, where the hash table operations for each row are collectively performed by a warp of threads. A post-processing step might be needed to remove the extra spaces left in the rows where step 2 of the algorithm overestimates the number of nonzeros.

4.3. Algebraic multigrid

Algebraic multigrid consists of a setup and a solve phase. We will focus mostly on the setup phase here, since its port to GPUs is significantly more complex than that of the solve phase, which will be briefly discussed at the end of the section.

The AMG setup phase has the following 4 main steps:

1. Construct strength of connection (SoC) matrices;
2. Use a coarsening algorithm to divide the variables into coarse and fine variables;
3. Build the interpolation and restriction operators for transferring data between the coarse and fine levels;
4. Generate the coarse-level operator via a Galerkin or triple-matrix product.

Our approach to GPU computing is to try to break down and reformulate algorithms into modules each of which can be executed by a call to one of the GPU kernels developed in *hypre* or available from various Nvidia GPU libraries such as Thrust, cuSPARSE and cuRAND. By relying on these kernels, one can completely avoid writing new GPU kernels for these algorithms.

For cases in which no suitable existing kernel is available and sufficient parallelism exists, good performance can be obtained by writing GPU kernels that are parallel by warps instead of by threads. A warp is a group of threads that are executed together on the GPU processing unit, essentially in SIMD mode. The warp size on Nvidia GPUs is currently 32 threads. Nvidia GPUs provide warp shuffle functions that permit very efficient reductions and sharing of data within a warp.

Construct SoC matrix. As an illustration, consider computing the SoC matrix, S, defined by matrix elements a_{ij} of A and assuming $a_{ii} > 0$

$$s_{ij} = \begin{cases} 1 & a_{ij} < \theta \min\limits_{k \neq i} a_{ik} \\ 0 & \text{otherwise} \end{cases},$$

where $0 < \theta < 1$ is a given constant. As a first step $min_{k \neq i} a_{ik}$ is computed for each row i, then the elements of A that satisfy $a_{ij} < \theta \min_{k \neq i} a_{ik}$ are flagged. Algorithm 1 shows a straightforward way to compute s_{ij}. In this algorithm, the arrays of the CSR representation of a sparse matrix A are denoted by `Ai`, `Aj`, and `Ad`, where `Ai[i]` points to the first element of row i in the array of column indices `Aj`, and where `Ad` is the data array.

The algorithm copies the column indices in array `Aj` that are strong connections to `Sj`. The elements of `Sj` can be initialized to -1. A postprocessing step can be used to remove those elements that were not overwritten by the indices of `Aj` indices. In principle, Algorithm 1 could be ported to GPUs by removing the outermost for-loop, and letting i be the thread rank. This loop-oriented approach to GPU porting is reasonable for some applications, e.g., partial differential equation (PDE) solvers where a fixed stencil is applied to an array. The approach would, however, face performance problems if applied to Algorithm 1. First, because there is no sharing of data between the rows, cache utilization would be very poor, and secondly, conditional statements in rows potentially cause divergent execution paths between threads.

A more efficient, warp based, algorithm is outlined in Algorithm 2, where i is set to the warp rank, and t is the thread rank within the warp ($0 \leq t < 32$). Algorithm 2 processes each row independently of other rows, and the threads

Algorithm 1 Strength matrix on CPU

```
for i:=1 to nrows do
  m := maxval
  for k := Ai[i] to Ai[i+1]−1 do
    m := min(m,Ad[k])
  endfor
  for k := Ai[i] to Ai[i+1]−1 do
    if Ad[k] < θm then
      Sj[k] := Aj[k]
    endif
  endfor
endfor
```

within the warp will access elements that are consecutive in memory, leading to better cache utilization.

Algorithm 2 Strength matrix on GPU

```
i := my_warp_id
t := my_thread_id
m := maxval
for k := Ai[i]+t, k := k+32 to Ai[i+1]−1 do
  m := min(m,Ad[k])
endfor
m := allreduce_warp(m)
for k := Ai[i]+t, k := k+32 to Ai[i+1]−1 do
  if Ad[k] < θm then
    Sj[k] := Aj[k]
  endif
endfor
```

PMIS coarsening. While there are various coarsening algorithms implemented in *hypre* on CPUs, it currently provides only one GPU-enabled coarsening, the parallel maximal independent set (PMIS) coarsening, which is based on the highly parallel Luby's algorithm [42] for finding maximal independent sets with a few small modifications. For further details on PMIS, see [23]. The random numbers used in PMIS are generated on GPUs using the cuRAND library.

Interpolation operators. Implementation of direct interpolation [50] on GPUs is straightforward, since the interpolatory set of each F-point i is located in the i-th row of the SoC matrix, and the interpolation formula is simple so that the weights can be computed locally without communication. However, direct interpolation generally leads to inferior convergence. To achieve good convergence one needs to combine PMIS coarsening with a distance-two interpolation formula, such as the extended and extended+i interpolation operators [22]. Their implementation is much more complicated, since they involve distance-2 neighbors and the sparse pattern of P needs to be determined dynamically. The CPU implementation of these algorithms in *hypre* is not suitable for GPUs. To overcome this obstacle, we developed a new class of interpolation operators based on matrix-matrix products with submatrices of A that include only strong connections a_{ij}, i.e. a_{ij} that correspond to nonzero elements s_{ij} in S. The new interpolation operators, referred to as MM-ext, MM-ext+i, and MM-ext+e, are used in *hypre*'s GPU implementation. Most of the computational work to construct the interpolation operators is done efficiently by *hypre*'s GPU-optimized SpGEMM routine. See [36] for details on the MM interpolation operators. It turns out that these new MM interpolation operators also lead to improved times when implemented on CPUs compared to extended+i interpolation, as demonstrated in one of our experiments in Section 6.

Aggressive coarsening. The generation of the coarse grid operator in AMG can lead to large memory and computational complexities. Often, this can be mitigated by coarsening more aggressively. Two different aggressive coarsening strategies are provided in *hypre*, A1- and A2-coarsening. The algorithm implemented follows the description in [50] by Stüben, where the standard SoC matrix, S, is first computed. A coarsening algorithm using S divides the nodes into coarse and fine points, C and F. Coarsening is then applied a second time, but now only to the set of C-points, resulting in a smaller set of final C-points.

The point i is strongly connected to j with a path of length two if there exists a k ($k \neq i, j$) such that i is strongly connected to k and k is strongly connected to j. The strength matrix for A1-coarsening has non-zero elements $s_{ij}^{(A1)} = 1$ if i is strongly connected to j with one or more paths of length one or two. For A2-coarsening, the non-zero elements of the strength matrix, $s_{ij}^{(A2)} = 1$, if i is strongly connected to j with at least two paths of length one or two. The (i, j) element of the squared strength matrix, i.e., $(S^2)_{ij} = \sum_{k=1}^{n} s_{ik} s_{kj}$, is equal to the number of paths of length two from i to j (for $i \neq j$). Hence, the number of paths of length two or length one from i to j is given by the i, j elements of

$$S^{(A)} = S^2 + S.$$

Because the second coarsening phase is only applied to C points, only the sub-block $S_{CC}^{(A)}$ is needed. *hypre* computes the sub-block using the block form

$$S = \begin{pmatrix} S_{FF} & S_{FC} \\ S_{CF} & S_{CC} \end{pmatrix},$$

to rewrite

$$S_{CC}^{(A)} = ((S + I)S)_{CC} = \begin{pmatrix} S_{CF} & S_{CC} + I_{CC} \end{pmatrix} \begin{pmatrix} S_{FC} \\ S_{CC} \end{pmatrix}.$$

The computation is done by the following three steps

1. Extract submatrices
$$S_1 = (S_{CF}\ S_{CC}) \text{ and } S_2 = \begin{pmatrix} S_{FC} \\ S_{CC} \end{pmatrix}.$$
2. Add the identity $(0\ I)$ to S_1.
3. Apply the SpGEMM kernel to compute $S_1 S_2$.

Steps 1 and 2 are relatively inexpensive. The main part of the execution time is once again spent in the sparse matrix-matrix multiplication in step 3.

The strength matrix for A1-coarsening is obtained by setting all non-zero off-diagonal elements in $S_{CC}^{(A)}$ to 1 and the diagonal elements to 0. Similarly, the strength matrix for A2-coarsening is obtained by setting all off-diagonal elements of $S_{CC}^{(A)}$ that are larger than 1 to 1, while eliminating all other elements.

Two-stage interpolation. Aggressive coarsening usually needs to be combined with a long distance interpolation operator, such as two-stage interpolation [53]. The GPU version of *hypre* provides two-stage MM-ext and MM-ext+e interpolation, which use the MM-ext and MM-ext+e interpolation operators mentioned above in both stages of the algorithms. These operators are also formulated as matrix-matrix products of smaller submatrices, which can be computed efficiently on GPUs.

Galerkin products. Galerkin products for coarse-level operators are computed in two multiplications as $R(AP)$ on GPUs. The required SpGEMM kernels have been discussed in Section 4.2. The local matrix concatenation, splitting and merging operations as in (3) and (4) are implemented using the parallel primitives in the Thrust library.

AMG solve on GPUs. Porting the AMG solve phase to GPUs is relatively easier than porting the AMG setup phase, since the main computations can be performed by sparse matrix and vector kernels, such as SpMVs, vector AXPYs and inner products. SpMVs with the interpolation operators P_l are used to move from a coarser level to a finer level, whereas SpMVs with the saved transpose of R_l are used to move to a lower level. Various popular AMG smoothers also consist mainly of SpMVs, which makes them suitable for GPU implementation. Those include l_1-Jacobi, weighted Jacobi and polynomial smoothers [6], which are all available in *hypre*'s GPU implementation.

4.4. Parallel ILU preconditioners

The parallel ILU preconditioners in *hypre* have also been enabled to run on GPUs. There are two types of parallel ILU preconditioners. The first one uses a block Jacobi approach, where the local diagonal block matrix on each process is factored by ILU(0) and the preconditioner is applied by solving the triangular systems in parallel. The ILU(0) factorizations are computed on GPUs using a routine from the cuSPARSE library, and the triangular systems are solved with the SpTrSV kernel. The second preconditioner uses a two-level approach that exploits domain decomposition (DD). The solution at the fine-level points that correspond to the decoupled interior points from DD is obtained using local ILU(0), whereas the solution at the coarse-level points, the domain interface, is obtained via the global Schur complement system. Several efficient approaches were proposed in [52] to iteratively solve the coarse-level system with Krylov subspace methods on GPUs, where the coarse-level operator is not formed explicitly and the main computations in the iterations can be performed in the form of SpMV with local matrix blocks and SpTrSV with the local ILU factors.

5. Additional Interfaces and Solvers

In this section we present additional interfaces, preconditioners and solvers in *hypre* that have either been ported to GPUs or will be targeted soon.

5.1. The semi-structured interface

The semi-structured or SStruct interface [27, 25] is built on top of the Struct and the IJ interface. It is designed for problems that are mostly structured, such as block structured, structured adaptive mesh refinement grids, or overset grids. It also has a finite element option. The interface provides access to more general PDEs than the structured interface, such as systems of PDEs as well as problems using different types of variables, including cell-centered, edge-centered, face-centered or nodal variables. The underlying semi-structured matrix and vector data structures consist of a structured and a generally much smaller unstructured part. Since the interface builds on both the Struct and the IJ interfaces, a large part of this GPU port can leverage the GPU implementation work already done on those components.

The SStruct interface provides access to semi-structured and unstructured solvers. Currently, the only AMG solver that is available in the interface is BoomerAMG, which internally converts structured parts to an unstructured representation and thus is unable to take advantage of the structure. However, we are developing a new semi-structured algebraic multigrid solver that is expected to be more suitable for GPU implementation.

5.2. Krylov solvers

The *hypre* library provides several Krylov solvers, which are implemented in a generic form, so they can be used with structured, semi-structured and unstructured matrices, vectors, and preconditioners. These methods include the conjugate gradient (CG) method, the generalized minimal residual (GMRES) method, and the biconjugate gradient stabilized (BiCGSTAB) method. These algorithms consist mostly of simple matrix and vector operations, such as inner products, AXPYs and SpMVs, which have been ported to GPUs. Consequently, these solvers are fully GPU-enabled when used without preconditioning or with a GPU-enabled preconditioner.

5.3. Maxwell and $H(div)$ solvers

The Auxiliary-space Maxwell Solver (AMS) [34] is a scalable parallel unstructured solver for second order definite and semi-definite Maxwell problems discretized with edge finite elements. Such problems occur in various physics applications, such as electromagnetic simulations. Internally, AMS uses *hypre*'s unstructured AMG solver, BoomerAMG. The Auxiliary-space Divergence Solver (ADS) is a parallel unstructured solver similar to AMS, but is targeting $H(div)$ instead of $H(curl)$ problems. Its usage and options are very

similar to those of AMS, and in general the relationship between ADS and AMS is analogous to that between AMS and AMG. The dependence of these solvers on AMG has facilitated their port to GPUs, since we were able to take advantage of the optimizations in AMG.

5.4. Multigrid reduction

Multigrid reduction (MGR) [16] is a parallel multigrid reduction solver and preconditioner designed to take advantage of user-provided information to solve systems of equations with multiple variable types. It uses two-stage preconditioner strategies and other reduction techniques in a standard multigrid framework. It accepts information about the variables in block form from the user and uses it to define the appropriate splitting into coarse and fine points for the multigrid scheme. The linear system solve proceeds with a relaxation solve on the fine points, followed by a coarse grid correction. The coarse grid solve is handled by scalar AMG (BoomerAMG). MGR provides users with more control over the coarsening process, and can potentially be a starting point for designing multigrid-based physics-based preconditioners. Here also, use of the GPU implementation of AMG will be advantageous.

5.5. pAIR method

The parallel approximate ideal restriction (pAIR) algorithm [31] is an algebraic multigrid method that was developed for highly nonsymmetric matrices, using a special restriction operator and a very lightweight interpolation operator, and has shown to be effective for advection-dominated problems such as the linear transport problems. It can be used as a special option in BoomerAMG. This dependence will enable the use of some GPU optimizations available in AMG when porting the algorithm to GPUs.

6. Numerical Results

In this section, we present various numerical results for different multigrid solvers, comparing CPU and GPU results. The experiments were run on up to 16 nodes of a heterogeneous computer at Lawrence Livermore National Laboratory with 4 Nvidia V100 GPUs and 2 IBM Power 9 CPUs per node. We used *hypre* version 2.22.1, CUDA 10.1.243, the IBM XL C/C++ compiler V16.1.1, and Spectrum MPI Release xl-2021.03.11. We configured the CPU version using `--with-openmp --enable-hopscotch`. The latter option provides improved OpenMP optimizations in the BoomerAMG setup phase. To achieve better performance in the BoomerAMG setup phase on GPUs, it is recommended to use memory pools for GPU memory. There are two options: adding `--enable-device-memory-pool` to the configure line will use a tool for Nvidia GPUs that is included with *hypre*. Another option that can lead to additional improvements is to install the memory management tool Umpire [10] or use an existing installation of it and enable it in *hypre*. We installed Umpire 5.0.1 for the runs using AMG-PCG presented in Figures 7 and 8. For the GPU runs (denoted 'GPU') we used 4 GPUs per node, and for the CPU runs we used either 40 MPI tasks per node (denoted 'CPU40') or 4 MPI tasks with 10 OpenMP threads each (denoted 'CPU').

Table 2
Speedups for GPU runs of PFMG-PCG over CPU runs for the Laplace problem on an $n \times n \times n$ grid

	Speedup GPU/CPU			Speedup GPU/CPU40		
n	Total	Setup	Solve	Total	Setup	Solve
200	0.5	0.7	0.5	0.3	0.5	0.4
400	1.0	0.8	1.1	0.8	0.6	1.0
600	2.3	1.0	2.8	2.1	1.0	2.5
800	4.9	1.1	5.9	4.3	1.3	5.1

We first investigated the performance of the structured solver PFMG [5] used as a preconditioner to conjugate gradient when applied to a 3-dimensional 7-point Laplace problem on an $n \times n \times n$ grid. We used 16 nodes, i.e. 64 (or 640) MPI processes, and varied n from 200, leading to a problem size of 500,000 grid points per node, to 800, a problem sizes of 32 million points per node. PFMG uses semicoarsening, i.e., the problem size is reduced by a factor of two for each level. Consequently, there are 22 multigrid levels for the smallest problem and 28 for the largest problem. We used the fastest version of PFMG for this problem, which enables the non-Galerkin version of PFMG and reduces the number of relaxation steps by skipping levels for relaxation in the solve phase. We also tested the default version, which achieved comparable speedups between CPU and GPU versions, but is not presented here to save space. The top graph in Figure 7 shows the runtimes, including setup, solve and total times. The results show that total times of CPU and GPU runs are about equal for $n = 400$, whereas the CPU40 runs are still slightly faster. For smaller n, all CPU runs are faster, and the GPU runs are faster when $n > 400$. The setup times for PFMG are slower on the GPU up to $n = 600$, however they are overall very small and insignificant compared to the total time. Table 2 presents speedups for setup, solve and total times for GPU over CPU and CPU40 runs. Using 40 MPI tasks per node vs 4 MPI tasks with 10 OpenMP threads leads to faster solve times.

We investigated the performance of BoomerAMG with PCG [32] for the same problem and present the results in the bottom of Figure 7. Here, we used PMIS coarsening, aggressive coarsening on the first level and MM-ext+i interpolation [36] for GPU and ext+i interpolation for CPU on all lower levels. Since aggressive coarsening requires prolongation operators with a longer range, we used multipass interpolation for the CPU version, which leads to the overall fastest CPU times for this problem. For the GPU version, we used two-stage MM-ext+e interpolation with truncation to 2 elements per row for the interpolation matrices in stage one and two, P_1 and P_2, and truncation to 4 elements for the final interpolation matrix $P = P_1 P_2$ [53, 36], since we currently do not have a GPU implementation of multipass interpolation. The results show that AMG-PCG takes overall much longer than PFMG-PCG as expected. The total GPU times are smaller than the CPU times for $n > 300$. The

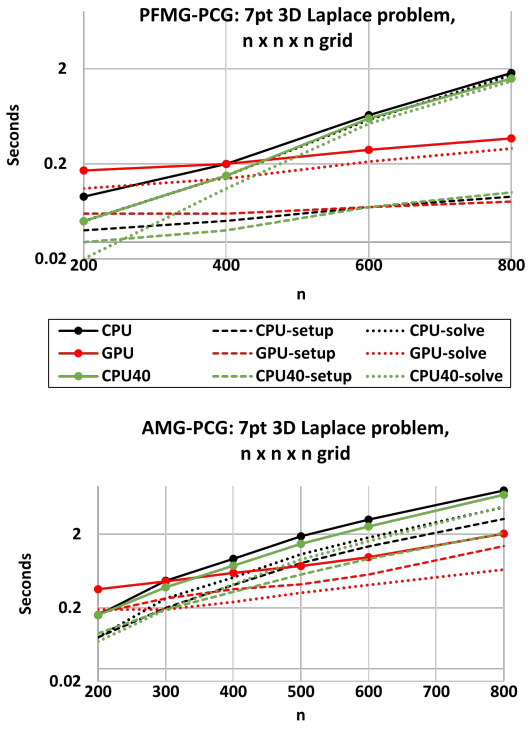


Figure 7: Solving a 7-point 3-D Laplace problem on a grid of size $n \times n \times n$ using PFMG-PCG (top) and AMG-PCG (bottom) on 16 nodes.

Table 3
Speedups for GPU runs of AMG-PCG over CPU runs for the Laplace problem on an $n \times n \times n$ grid

	Speedup GPU/CPU			Speedup GPU/CPU40		
n	Total	Setup	Solve	Total	Setup	Solve
200	0.4	0.5	0.4	0.4	0.5	0.4
300	1.0	0.7	1.4	0.8	0.7	1.0
400	1.6	1.1	2.2	1.3	0.9	1.8
500	2.5	1.9	3.3	2.0	1.4	2.8
600	3.2	2.4	4.4	2.6	1.6	4.0
800	3.9	2.3	7.1	3.4	1.5	7.2

speedups are presented in Table 3. AMG is coarsened to a system smaller than 9, which is then solved by Gaussian elimination. It generates about 8 to 10 levels for the problems considered here. Using 40 MPI tasks per node versus 4 MPI tasks with 10 OpenMP threads leads to better times, particularly for the setup phase.

Since the Laplace problem considered uses a very sparse matrix with only 7 elements per row and GPUs generally perform better on denser matrices, we present results for a diffusion problem with a 27-point stencil in Figure 8. For this problem, we used MM-ext+i interpolation with truncation to 4 elements per row for both CPU and GPU versions and a Jacobi smoother with a weight of 0.85. GPU times are better than CPU times for $n > 200$, and larger speedups than in the case of the 7-point matrix are achieved for setup, solve and total times, see Table 4. Here, 8 to 10 multigrid levels

Table 4
Speedups for GPU runs of AMG-PCG over CPU runs for the 27pt diffusion problem on an $n \times n \times n$ grid

	Speedup GPU/CPU			Speedup GPU/CPU40		
n	Total	Setup	Solve	Total	Setup	Solve
200	0.8	0.9	0.7	0.8	0.9	0.7
300	1.9	1.7	2.2	1.8	1.7	1.8
400	3.1	2.7	3.8	2.9	2.5	3.6
500	4.3	3.9	5.1	4.0	3.7	4.6
600	4.4	3.6	6.5	4.1	3.2	6.1
800	4.6	3.5	7.7	4.4	3.3	7.5

are generated.

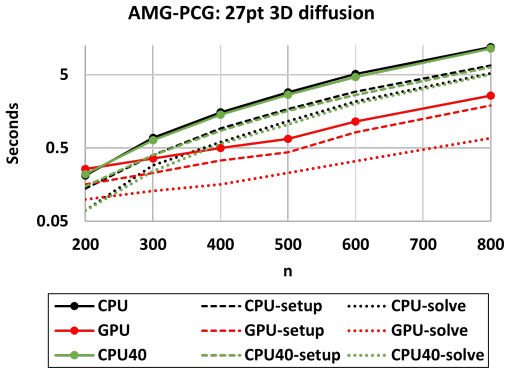


Figure 8: Solving a 27-point 3-D diffusion problem on a grid of size $n \times n \times n$ using AMG-PCG using 16 nodes (64 MPI tasks) .

Figure 9 shows timings for the main components of the AMG setup phase on the individual levels for two 27-point diffusion problems, a large problem of size 64 million and a small problem with 1 million points, using 8 GPUs on 2 nodes. The results show that for this problem, on each level, most of the time is spent in computing the interpolation, followed by the evaluation of the Galerkin product. For the large problem, most of the time is spent on the finest level, and the time on the coarser levels becomes insignificant, whereas for the small problem timings on the coarse levels are significant. Note that an investigation of the setup phase for the Laplace problem using aggressive coarsening on the finest level would look similar, even though, on the GPU, applying MM-ext+i interpolation to a 7-point matrix is much faster than to a 27-point matrix and requires less time than the Galerkin product, however the evaluation of the long range two-stage interpolation is expensive.

Finally, we present a weak scaling study of an unstructured problem which is posed on a crooked pipe as illustrated in Figure 10. The systems are generated with the finite element discretization library MFEM [2, 45] using unstructured hexahedral finite elements and have the following numbers of grid points: 966,609; 7,544,257; 59,604,993. The matrices have on average 27 nonzeroes per row. This problem is very hard to solve for AMG, since a dense layer of highly stretched elements has been added to the neighbor-

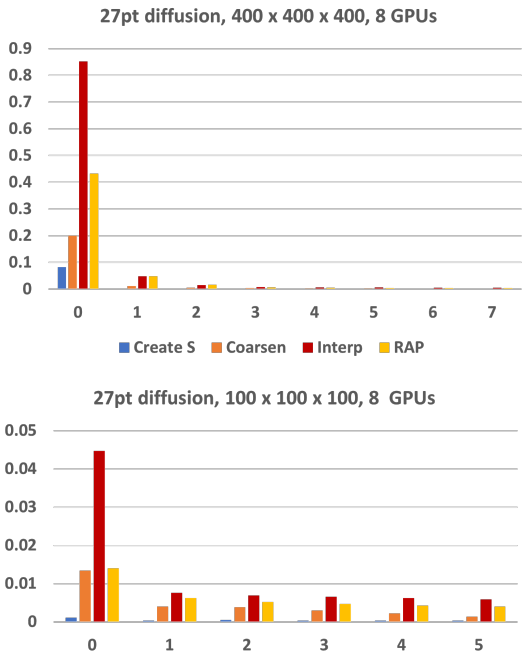


Figure 9: Timings per multigrid levels for setup phase components (creation of the strength matrix, coarsening, generation of the interpolation and the Galerkin product) for a large problem (top) and a small problem (bottom). The y-axis shows execution time in seconds, and the x-axis shows multigrid levels, with 0 denoting the finest level.

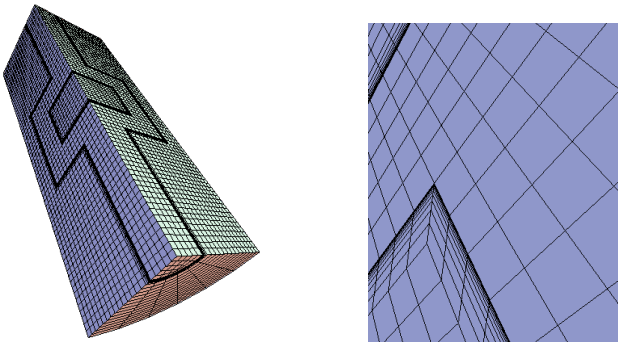


Figure 10: Mesh for the crooked pipe problem.

hood of the material interface as illustrated on the right in Figure 10. It requires many iterations. We used MM-ext+i interpolation for the comparison of both CPU and GPU versions. Here, we used the memory pool routines provided with *hypre*. We also added CPU results using ext+i interpolation [22] to show that designing a more GPU-friendly method can also lead to improved performance on CPUs. The speedup numbers recorded in Figure 11 are however computed using the CPU times generated with MM-ext+i interpolation. Here we used two sweeps of l_1-Jacobi as a smoother. We present comparisons for both a CPU version using 4 MPI tasks per node with 10 OpenMP threads in the upper graph in Figure 11 and a CPU version using flat MPI, i.e., 10 MPI tasks compared to use of one GPU. We achieve the highest speedup for the case with just one MPI task, where we have no communication between MPI processes. Overall speedups increase slightly from 8 to 64 MPI tasks for the OpenMP CPU version, however decrease for the flat MPI version. Note that the use of the new MM-ext+i interpolation does not only improve setup times but also reduces the number of iterations by about 10 to 15 percent for this problem. For further comparisons of convergence and CPU performance for MM-ext+i and ext+i interpolation, see [36].

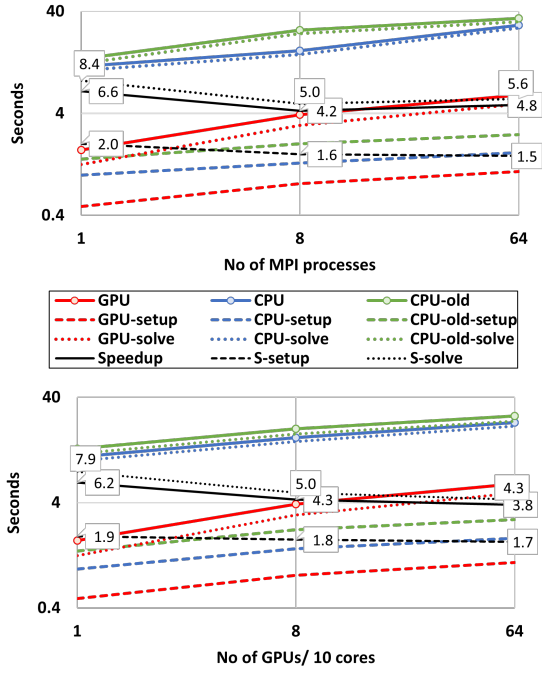


Figure 11: Weak scaling study for a problem on the mesh shown in Figure 10 using 16 nodes comparing CPU and GPU runs using MM-ext+i interpolation for 'CPU', 'GPU' and 'Speedup'. For the 'CPU-old' results, extended+i interpolation was used. In the upper figure, 10 OpenMP threads per MPI task were used for the CPU runs, whereas the lower figure shows flat MPI CPU results, i.e. 10 MPI tasks for the first run, 80 for the second and 640 for the last run.

7. Conclusion

We presented our strategies and efforts to port *hypre*'s structured and unstructured interfaces and multigrid solvers to GPUs. This included the introduction of a new memory model, our efforts to add new programming models to the *hypre* BoxLoop macros and the modularization of the unstructured multigrid solver BoomerAMG. We described several of our GPU kernels, which included matrix-vector and matrix-matrix operations. We also presented additional GPU-enabled components in *hypre*, including ILU preconditioners and Krylov methods, solvers for Maxwell and H(div) problems, as well as other solvers and interfaces not fully ported to GPUs yet. We showed numerical experiments that showed speedups for large problems. However, small problems can generally be solved faster on CPUs.

While we made good progress in porting *hypre* to GPUs, we are far from being finished. In order to further improve the performance on GPUs, we will add multipass interpolation operators based on matrix operations and additional smoothers, including polynomial smoothers. We will also focus on GPU implementations of the SStruct interface and specialized solvers in *hypre*, such as pAIR and MGR. Finally, we will port GPU-enabled portions in *hypre* to additional heterogeneous architectures, including AMD and Intel GPUs, by converting CUDA routines to HIP and DPC++ via vendor tools and adding performance improvements based on profiling.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. This work was supported by the Exascale Computing Project, a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

References

[1] Anderson, E., Saad, Y., 1989. Solving sparse triangular linear systems on parallel computers. Int. J. High Speed Comput. 1, 73–95. URL: http://dx.doi.org/10.1142/S0129053389000056, doi:10.1142/S0129053389000056.

[2] Anderson, R., Andrej, J., Barker, A., Bramwell, J., Camier, J.S., Dobrev, J.C.V., Dudouit, Y., Fisher, A., Kolev, T., Pazner, W., Stowell, M., Tomov, V., Akkerman, I., Dahm, J., Medina, D., Zampini, S., 2020. MFEM: A modular finite element library. Computers & Mathematics with Applications doi:10.1016/j.camwa.2020.06.009.

[3] Anh, P.N.Q., Fan, R., Wen, Y., 2016. Balanced hashing and efficient gpu sparse general matrix-matrix multiplication, in: Proceedings of the 2016 International Conference on Supercomputing, Association for Computing Machinery, New York, NY, USA. URL: https://doi.org/10.1145/2925426.2926273, doi:10.1145/2925426.2926273.

[4] Ashari, A., Sedaghati, N., Eisenlohr, J., Parthasarathy, S., Sadayappan, P., 2014. Fast sparse matrix-vector multiplication on GPUs for graph applications, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE Press, Piscataway, NJ, USA. pp. 781–792. URL: https://doi.org/10.1109/SC.2014.69, doi:10.1109/SC.2014.69.

[5] Ashby, S., Falgout, R., 1996. A parallel multigrid preconditioned conjugate gradient algorithm for groundwater flow simulations. Nuclear Science and Engineering 124, 145–159.

[6] Baker, A.H., Falgout, R.D., Kolev, T.V., Yang, U.M., 2011. Multigrid smoothers for ultraparallel computing. SIAM Journal on Scientific Computing 33, 2864–2887. URL: https://doi.org/10.1137/100798806, doi:10.1137/100798806, arXiv:https://doi.org/10.1137/100798806.

[7] Baker, A.H., Falgout, R.D., Kolev, T.V., Yang, U.M., 2012. Scaling Hypre's Multigrid Solvers to 100,000 Cores. Springer London, London. pp. 261–279. URL: https://doi.org/10.1007/978-1-4471-2437-5_13, doi:10.1007/978-1-4471-2437-5_13.

[8] Baskaran, M.M., Bordawekar, R., 2008. Optimizing sparse matrix-vector multiplication on GPUs using compile-time and run-time strategies. IBM Reserach Report, RC24704 (W0812-047) .

[9] Beckingsale, D., Hornung, R., Scogland, T., Vargas, A., 2019. Performance portable c++ programming with raja, in: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, Association for Computing Machinery, New York, NY, USA. p. 455–456. URL: https://doi.org/10.1145/3293883.3302577, doi:10.1145/3293883.3302577.

[10] Beckingsale, D.A., McFadden, M.J., Dahm, J.P.S., Pankajakshan, R., Hornung, R.D., 2020. Umpire: Application-focused management and coordination of complex hierarchical memory. IBM Journal of Research and Development 64, 00:1–00:10. doi:10.1147/JRD.2019.2954403.

[11] Bell, N., Dalton, S., Olson, L.N., 2012. Exposing fine-grained parallelism in algebraic multigrid methods. SIAM Journal on Scientific Computing 34, C123–C152. URL: https://doi.org/10.1137/110838844, doi:10.1137/110838844, arXiv:https://doi.org/10.1137/110838844.

[12] Bell, N., Garland, M., 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, ACM, New York, NY, USA. pp. 18:1–18:11. doi:10.1145/1654059.1654078.

[13] Bell, N., Hoberock, J., 2012. Chapter 26 - thrust: A productivity-oriented library for cuda, in: mei W. Hwu, W. (Ed.), GPU Computing Gems Jade Edition. Morgan Kaufmann, Boston. Applications of GPU Computing Series, pp. 359 – 371. URL: http://www.sciencedirect.com/science/article/pii/B9780123859631000265, doi:https://doi.org/10.1016/B978-0-12-385963-1.00026-5.

[14] Brandt, A., McCormick, S., Ruge, J., 1984. in: Evans (Ed.), Sparsity and Its Applications. Cambridge University Press, Cambridge. chapter Algebraic multigrid (AMG) for sparse matrix equations.

[15] Brown, P.N., Falgout, R.D., Jones, J.E., 2000. Semicoarsening multigrid on distributed memory machines. SIAM J. Sci. Comput 21, 1823–1834.

[16] Bui, Q., Osei-Kuffuor, D., Castelletto, N., White, J., 2020. A scalable multigrid reduction framework for multiphase poromechanics. SIAM J. Sci. Comput. 42, B379–B396.

[17] Carter Edwards, H., Trott, C.R., Sunderland, D., 2014. Kokkos. J. Parallel Distrib. Comput. 74, 3202–3216. URL: https://doi.org/10.1016/j.jpdc.2014.07.003, doi:10.1016/j.jpdc.2014.07.003.

[18] Choi, J.W., Singh, A., Vuduc, R.W., 2010. Model-driven autotuning of sparse matrix-vector multiply on GPUs. SIGPLAN Not. 45, 115–126. URL: http://doi.acm.org/10.1145/1837853.1693471, doi:10.1145/1837853.1693471.

[19] Cohen, E., 1998. Structure prediction and computation of sparse matrix products. Journal of Combinatorial Optimization 2, 307–332. URL: https://doi.org/10.1023/A:1009716300509, doi:10.1023/A:1009716300509.

[20] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C., 2001. Introduction to Algorithms. 2nd ed., McGraw-Hill Higher Education.

[21] Dalton, S., Olson, L., Bell, N., 2015. Optimizing sparse matrix—matrix multiplication for the gpu. ACM Trans. Math. Softw. 41. URL: https://doi.org/10.1145/2699470, doi:10.1145/2699470.

[22] De Sterck, H., Falgout, R.D., Nolting, J.W., Yang, U.M., 2008. Distance-two interpolation for parallel algebraic multigrid. Numerical Linear Algebra with Applications 15, 115–139.

[23] De Sterck, i., Yang, U., Heys, i., 2006. Reducing complexity in parallel algebraic multigrid preconditioners. SIAM J. on Matrix Analysis and Applications 27, 1019–1039.

[24] Deveci, M., Trott, C., Rajamanickam, S., 2018. Multithreaded sparse matrix-matrix multiplication for many-core and gpu architectures. Parallel Computing 78, 33 – 46. URL: http://www.sciencedirect.com/science/article/pii/S0167819118301923, doi:https://doi.org/10.1016/j.parco.2018.06.009.

[25] Falgout, R., Jones, J., Yang, U.M., 2006. in: Bruaset, A., Tveito, A. (Eds.), Numerical Solutions of Partial Differential Equations on Parallel Computers. Springer-Verlag. Lecture Notes in Computational Science and Engineering. chapter The Design and Implementation of hypre, a Library of Parallel High Performance Preconditioners, pp. 267–294.

[26] Falgout, R.D., 2006. An introduction to algebraic multigrid. Computing in Science Engineering 8, 24–33.

[27] Falgout, R.D., Jones, J.E., Yang, U.M., 2005. Pursuing scalability for hypre's conceptual interfaces. ACM Trans. Math. Softw. 31, 326–350.

[28] Falgout, R.D., Yang, U.M., 2002. hypre: A library of high per-

formance preconditioners, in: Sloot, P.M.A., Hoekstra, A.G., Tan, C.J.K., Dongarra, J.J. (Eds.), Computational Science — ICCS 2002, Springer Berlin Heidelberg, Berlin, Heidelberg. pp. 632–641.

[29] George, A., Heath, M.T., Liu, J., Ng, E., 1986. Solution of sparse positive definite systems on a shared-memory multiprocessor. International Journal of Parallel Programming 15, 309–325. URL: https://doi.org/10.1007/BF01407878, doi:10.1007/BF01407878.

[30] Gremse, F., Höfter, A., Schwen, L.O., Kiessling, F., Naumann, U., 2015. Gpu-accelerated sparse matrix-matrix multiplication by iterative row merging. SIAM Journal on Scientific Computing 37, C54–C71. URL: https://doi.org/10.1137/130948811, doi:10.1137/130948811, arXiv:https://doi.org/10.1137/130948811.

[31] Hanophy, J., Southwoorth, B., Li, R., Manteuffel, T., Morel, J., 2020. Parallel approximate ideal restriction multigrid for solving the S_N transport equations. Nuclear Science and Engineering 194, 989–1008.

[32] Henson, V.E., Yang, U.M., 2002. Boomeramg: A parallel algebraic multigrid solver and preconditioner. Applied Numerical Mathematics 41, 155 – 177.

[33] hypre, . HYPRE: Scalable Linear Solvers and Multigrid Methods. http://www.llnl.gov/CASC/hypre/.

[34] Kolev, T., Vassilevski, P., 2009. Parallel auxiliary space amg for h(curl) problems. J. Comput. Math. 27, 604–623.

[35] Li, R., Saad, Y., 2013. GPU-accelerated preconditioned iterative linear solvers. The Journal of Supercomputing 63, 443–466. doi:10.1007/s11227-012-0825-3.

[36] Li, R., Sjögreen, B., Yang, U.M., 0. A new class of amg interpolation methods based on matrix-matrix multiplications. SIAM Journal on Scientific Computing 0, S540–S564. URL: https://doi.org/10.1137/20M134931X, doi:10.1137/20M134931X, arXiv:https://doi.org/10.1137/20M134931X.

[37] Li, R., Zhang, C., 2020. Efficient parallel implementations of sparse triangular solves for gpu architectures, in: Biros, G., Yang, U.M. (Eds.), Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing, Society for Industrial and Applied Mathematics, Philadelphia, PA. pp. 106–117. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611976137.10, doi:10.1137/1.9781611976137.10, arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611976137.10.

[38] Liu, W., Li, A., Hogg, J.D., Duff, I.S., Vinter, B., 2016. A Synchronization-Free Algorithm for Parallel Sparse Triangular Solves. Springer International Publishing. pp. 617–630. doi:10.1007/978-3-319-43659-3_45.

[39] Liu, W., Vinter, B., 2014. An efficient GPU general sparse matrix-matrix multiplication for irregular data, in: Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium, IEEE Computer Society, Washington, DC, USA. pp. 370–381. URL: http://dx.doi.org/10.1109/IPDPS.2014.47, doi:10.1109/IPDPS.2014.47.

[40] Liu, W., Vinter, B., 2015a. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication, in: Proceedings of the 29th ACM International Conference on Supercomputing, ACM, New York, NY, USA. pp. 339–350. URL: http://doi.acm.org/10.1145/2751205.2751209, doi:10.1145/2751205.2751209.

[41] Liu, W., Vinter, B., 2015b. Speculative segmented sum for sparse matrix-vector multiplication on heterogeneous processors. Parallel Computing 49, 179 – 193. URL: http://www.sciencedirect.com/science/article/pii/S0167819115000770, doi:http://dx.doi.org/10.1016/j.parco.2015.04.004.

[42] Luby, M., 1986. A simple parallel algorithm for the maximal independent set problem. SIAM Journal on Computing 15, 1036–1053.

[43] Marichal, R., Dufrechou, E., Ezzatti, P., 2020. Towards a lightweight method to predict the performance of sparse triangular solvers on heterogeneous hardware platforms, in: Crespo-Mariño, J.L., Meneses-Rojas, E. (Eds.), High Performance Computing, Springer International Publishing, Cham. pp. 109–121.

[44] Matam, K., Indarapu, S.R.K.B., Kothapalli, K., 2012. Sparse matrix-matrix multiplication on modern architectures, in: 2012 19th International Conference on High Performance Computing, pp. 1–10. doi:10.1109/HiPC.2012.6507483.

[45] mfem, . MFEM: Modular finite element methods [Software]. mfem.org. doi:10.11578/dc.20171025.1248.

[46] Nagasaka, Y., Nukada, A., Matsuoka, S., 2017. High-performance and memory-saving sparse general matrix-matrix multiplication for nvidia pascal gpu, in: 2017 46th International Conference on Parallel Processing (ICPP), pp. 101–110. doi:10.1109/ICPP.2017.19.

[47] Naumov, M., 2011. Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU. NVIDIA Corp., Westford, MA, USA, Tech. Rep. NVR-2011 1.

[48] Picciau, A., Inggs, G.E., Wickerson, J., Kerrigan, E.C., Constantinides, G.A., 2016. Balancing locality and concurrency: Solving sparse triangular systems on GPUs, in: 2016 IEEE 23rd International Conference on High Performance Computing (HiPC), pp. 183–192. doi:10.1109/HiPC.2016.030.

[49] Saltz, J.H., 1990. Aggregation methods for solving sparse triangular systems on multiprocessors. SIAM Journal on Scientific and Statistical Computing 11, 123–144. URL: https://doi.org/10.1137/0911008, doi:10.1137/0911008, arXiv:https://doi.org/10.1137/0911008.

[50] Stüben, K., 2000. in: Trottenberg, U., Schuller, A. (Eds.), Multigrid. Academic Press, Inc., USA. chapter Algebraic multigrid (AMG): an introduction with applications.

[51] Suchoski, B., Severn, C., Shantharam, M., Raghavan, P., 2012. Adapting sparse triangular solution to GPUs, in: 2012 41st International Conference on Parallel Processing Workshops, pp. 140–148. doi:10.1109/ICPPW.2012.23.

[52] Xu, T., Li, R., Osei-Kuffuor, D., 2020. A two-level gpu-accelerated incomplete lu preconditioner for general sparse linear systems. Numerical Linear Algebra with Applications Submitted.

[53] Yang, U.M., 2010. On long-range interpolation operators for aggressive coarsening. Numerical Linear Algebra with Applications 17, 453–472.