An Optimized Multicolor Point-Implicit Solver for Unstructured Grid Applications on Graphics Processing Units

Mohammad Zubair,* Eric Nielsen,† Justin Luitjens,‡ and Dana Hammond†
*Department of Computer Science, Old Dominion University, Norfolk, Virginia
†Computational AeroSciences Branch, NASA Langley Research Center, Hampton, Virginia
‡NVIDIA Corporation, Santa Clara, California

***Abstract*—In the field of computational fluid dynamics, the Navier-Stokes equations are often solved using an unstructured-grid approach to accommodate geometric complexity. Implicit solution methodologies for such spatial discretizations generally require frequent solution of large tightly-coupled systems of block-sparse linear equations. The multicolor point-implicit solver used in the current work typically requires a significant fraction of the overall application run time. In this work, an efficient implementation of the solver for graphics processing units is proposed. Several factors present unique challenges to achieving an efficient implementation in this environment. These include the variable amount of parallelism available in different kernel calls, indirect memory access patterns, low arithmetic intensity, and the requirement to support variable block sizes. In this work, the solver is reformulated to use standard sparse and dense Basic Linear Algebra Subprograms (BLAS) functions. However, numerical experiments show that the performance of the BLAS functions available in existing CUDA libraries is suboptimal for matrices representative of those encountered in actual simulations. Instead, optimized versions of these functions are developed. Depending on block size, the new implementations show performance gains of up to 7x over the existing CUDA library functions.**

I. Introduction

FUN3D is a suite of computational fluid dynamics software [1] developed at the NASA Langley Research Center to solve the Navier-Stokes (NS) equations for a broad range of aerodynamics applications across the speed range. The NS equations constitute a complex system of time-dependent nonlinear partial differential equations (PDEs) expressing the conservation of mass, momentum, and energy and are characterized by tightly-coupled multiscale interactions. The system is often closed using auxiliary PDEs governing turbulence quantities. For high-energy flows, traditional perfect gas assumptions are invalid and the system must be further expanded to accommodate finite-rate chemistry models. For these reasons, accurate and efficient simulations of complex aerodynamic flows are challenging and require significant computational resources.

FUN3D uses an implicit time-integration strategy based on a node-based, finite-volume spatial discretization on mixed-element unstructured grids. An approximate nearest-neighbor linearization of the residual equations for each control volume gives rise to a large tightly-coupled system of block-sparse linear equations that must be solved at each physical time step. The block size is determined by the number of governing equations and may range from five to several dozen. Multicolor point-implicit iterations are used to solve the system of linear equations. Although the implementation has been highly optimized for central processing units, computing solutions to these linear systems nonetheless accounts for a significant fraction of the overall runtime in virtually all FUN3D simulations.

In this paper, an effort to port the multicolor point-implicit linear solver used within FUN3D to graphics processing units (GPUs) is described. Many researchers have studied efficient implementations of iterative solvers on GPUs [2], [3]. Most of these solvers have been for sparse matrices, and the focus was primarily on optimization of an underlying sparse matrix-vector product. The sparse matrix-vector product has also been explored separately [4], [5], [6], [7], [8], [9]. To achieve improved performance, implementations have been developed using *cuSPARSE* library functions [10], [11].

To develop an efficient GPU implementation of the multicolor point-implicit solver, functions provided by the *cuSPARSE* and *cuBLAS* [12] libraries have first been considered. The function *cusparseSbsrmv* multiplies a block-sparse matrix with a vector, and the function *cublasStrsmBatched* solves block systems of equations by performing forward and backward substitutions using a previously-factored coefficient matrix. Numerical experiments show that the performance of the library functions is suboptimal for linear systems representative of those encountered in FUN3D simulations.

Optimized implementations of the *cusparseSbsrmv* and *cublasStrsmBatched* functions are proposed. To perform a sparse matrix-vector product, an algorithm that allocates a number of warps, or a group of 32 threads, to process a subset of the blocks in one row of the sparse matrix is proposed. Additional warps are then allocated to perform forward and backward substitutions. Several challenges are encountered, including a variable extent of available parallelism, indirect memory addressing, low arithmetic intensity, and the need to accommodate different block sizes. To address these challenges, particular emphasis is placed on coalesced memory loads, the use of shared memory and pre-fetching, minimal thread divergence within warps, and strategic use of shuffle instructions available on recent hardware.

While FUN3D can achieve coarse-grained parallelism by

using a conventional domain decomposition approach coupled with standard message passing techniques, the scope of the current effort is an optimal solver kernel appropriate for execution on a single device. The extension to node-level parallelism and multiple devices is relegated to future work.

The remainder of the paper is organized as follows. First, a description of the linear system and associated data structures is provided in Section II. Details of the multicolor point-implicit algorithm are presented in Section III. Section IV describes the optimized implementation of the solver for GPU architectures. Computational results are presented in Section V, including comparisons of the initial approach based on existing library functions with an optimized implementation developed in the current work. A summary is presented in Section VI along with a discussion of future work.

II. Linear System Layout and Data Structures

The implicit solution approach used within FUN3D results in linear systems of equations of the form $\mathbf{Ax} = \mathbf{b}$ that must be frequently solved during the course of a simulation. For a spatial mesh containing n grid points, $\mathbf{A}$ represents a sparse $n \times n$ matrix, where each matrix entry is a dense block of scalar coefficients of size $nb \times nb$ based on the linearization of the nonlinear governing equations at each grid point. Each of the n rows and columns containing $nb \times nb$ blocks are referred to as a $brow$ and a $bcol$, respectively. The matrix $\mathbf{A}$ is segregated into two separate matrices,

$$\mathbf{A} \equiv \mathbf{D} + \mathbf{O} \tag{1}$$

where $\mathbf{D}$ and $\mathbf{O}$ represent the diagonal and off-diagonal blocks of $\mathbf{A}$, respectively.

During FUN3D initialization, the grid points are renumbered using a reverse Cuthill-McKee (RCM) algorithm [13]. A similar technique is used for other grid entities such as elements and edges. This approach improves cache locality during stencil-based operations such as flux and Jacobian evaluations and yields a tightly-banded matrix structure for the implicit system of equations. An example of applying this renumbering strategy to a tetrahedral mesh with $n = 983,633$ is shown in Figure 1, where the indicated non-zero off-diagonal entries represent dense blocks of fixed size for a given simulation.

The data layout used to store the matrix $\mathbf{D}$ is straightforward. An array D of the form (nb, nb, n) is used to hold the n blocks of $\mathbf{D}$, where each $nb \times nb$ block is stored in column-major order.* Prior to performing each linear solve, each diagonal block $\mathbf{D}_i$ is decomposed in-place into lower and upper triangular matrices $\mathbf{L}_i$ and $\mathbf{U}_i$, respectively, for $1 \leq i \leq n$. Values are stored using sixty-four bit precision for numerical stability during the solution phase.

The sparse $n \times n$ matrix $\mathbf{O}$ contains nnz non-zero block entries whose $nb \times nb$ blocks are stored using a modified

*Note that the convention used here employs a boldface symbol to represent the matrix and italic notation to indicate the array holding the non-zero values of the matrix.

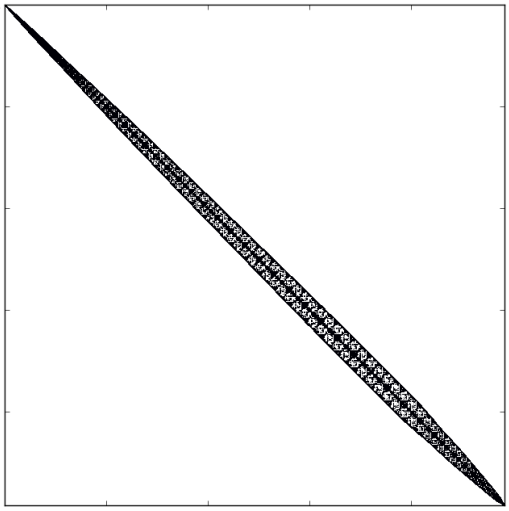

Fig. 1: Off-diagonal matrix structure after applying RCM renumbering.

compressed sparse row (CSR) [14] format in which the diagonal blocks are omitted. In this approach, two integer arrays ia and ja are used to efficiently capture the sparsity pattern of the matrix. The array ia is a rank-1 array of size $n + 1$ whose i-th entry indicates the leading non-zero block index in the i-th $brow$ of $\mathbf{O}$. The array includes a fictitious $n+1$ entry to facilitate easy traversal of the elements through the n-th $brow$. The ja array is a rank-1 array of size nnz that provides the $bcol$ index for each non-zero block. A third array, O, of the form (nb, nb, nnz), is used to store the non-zero entries. Each $nb \times nb$ block is stored in column-major order. Thirty-two bit precision is used for the off-diagonal entries, which substantially reduces the overall memory requirement and significantly improves cache performance during the linear solve. Figures 2 and 3 show a sample block-sparse matrix with $nb = 2$ and the corresponding CSR arrays, respectively.

III. Multicolor Point-Implicit Solver

Several linear-solver options are provided within FUN3D; the scheme most commonly used in practice is a multicolor point-implicit relaxation. In this scheme, the grid points are grouped, or colored, such that no two adjacent points are assigned the same color. All unknowns associated with a grid point are assigned the color of the point. Since an approximate nearest-neighbor flux Jacobian is used to construct the matrix $\mathbf{A}$, unknowns of the same color carry no data dependencies and may be updated in parallel in a Jacobi-like fashion. Colors are processed sequentially. Updates of unknowns of each color use the latest updated values of $\mathbf{x}$ corresponding to other colors. The overall process may be repeated using several outer sweeps over the entire system.

Since unknowns of the same color cannot be topologically adjacent, the multicolor approach produces an inherently poor memory access pattern that discourages cache reuse. To improve cache performance, the system of algebraic equations

(a)

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 3 & 5 & 7 \\ 0 & 0 & 0 & 0 & 2 & 4 & 6 & 8 \\ 0 & 0 & 0 & 0 & 9 & 11 & 0 & 0 \\ 0 & 0 & 0 & 0 & 10 & 12 & 0 & 0 \\ 13 & 15 & 17 & 19 & 0 & 0 & 0 & 0 \\ 14 & 16 & 18 & 20 & 0 & 0 & 0 & 0 \\ 21 & 23 & 0 & 0 & 0 & 0 & 0 & 0 \\ 22 & 24 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(b)

Fig. 2: Figure (a) shows the sparsity structure of a matrix $\mathbf{O}$. An entry $\times$ indicates a non-zero block. Figure (b) shows $\mathbf{O}$ for a block size of 2×2.

$$\begin{aligned} ia &= [1, 3, 4, 6, 7] \\ ja &= [3, 4, 3, 1, 2, 1] \\ O &= [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, \\ &\quad 15, 16, 17, 18, 19, 20, 21, 22, 23, 24] \end{aligned}$$

Fig. 3: CSR storage for the matrix of Figure 2.

is renumbered such that unknowns of the same color appear in consecutive order and the arrays ia and ja are modified appropriately to reflect the new matrix structure. In this fashion, stencil-based operations may be performed in a memory-efficient manner while their contributions to $\mathbf{A}$ and $\mathbf{b}$ may be mapped directly to a data layout more amenable to multicolor relaxation. Upon completion of the linear solve, an inverse map is used to update the nonlinear solution of the PDEs at each grid point.

Eleven colors are required to process the matrix shown in Figure 1. Grouping the matrix rows according to their assigned colors produces the off-diagonal matrix shown in Figure 4, where only rows associated with three colors are highlighted for clarity. The resulting matrix structure has multiple bands; each band corresponds to a color. The number of rows in each color band is seen to decrease significantly with increasing color index. The first few colors typically account for the majority of the unknowns, while groups corresponding to colors with a large index may contain only a few entries.

Solutions of linear systems based on matrices of the form shown in Figure 4 present several challenges for modern accelerator and many-core programming environments. (1) With an average of fourteen off-diagonal blocks per row, the arithmetic intensity of the computation is quite low (≈ 0.5). This results in a memory-bound scenario on the NVIDIA® K40 GPU

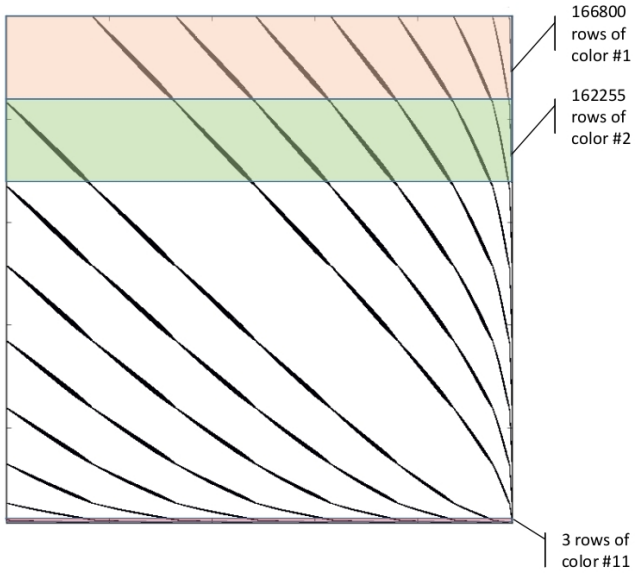

Fig. 4: Off-diagonal matrix structure after RCM renumbering and reordering based on row colors. Only three colors are highlighted for clarity.

as indicated by the roofline model shown in Figure 5. (2) Since the number of unknowns within a color may vary substantially, the amount of parallelism available may also vary. (3) The solver uses indirect memory addressing. (4) The implementation must support a broad range of block sizes.

To facilitate discussion of the proposed algorithms, the following nomenclature is used. The scalar value nc represents the total number of colors required for the linear system. The arrays $bc(nc)$ and $ec(nc)$ are used to keep track of the starting and ending $brow$ indices for each color. The scalar $n_k = ec(k) - bc(k) + 1$ is the number of $brows$ of color k for $1 \leq k \leq nc$.

For color k, the solution procedure consists of three steps:

1) Compute $\mathbf{q_i} = \mathbf{O_i x}$, for $bc(k) \leq i \leq ec(k)$. Here, $\mathbf{O_i}$ is the i-th $brow$ of $\mathbf{O}$. Note that the output vector $\mathbf{q}$ has n_k components and its ith component $\mathbf{q_i}$ has nb elements.
2) Solve for $\mathbf{y_i}$ in $\mathbf{L_i y_i} = \mathbf{q_i}$, for $bc(k) \leq i \leq ec(k)$.
3) Solve for $\mathbf{x_i}$ in $\mathbf{U_i x_i} = \mathbf{y_i}$, for $bc(k) \leq i \leq ec(k)$. The output vector $\mathbf{x}$ has n_k components and its i-th component $\mathbf{x_i}$ has nb elements.

Step 1 represents a block-sparse matrix-vector multiplication between a sub-matrix of $\mathbf{O}$ of size $n_k \times nb$ and corresponding elements of the vector $\mathbf{x}$. The *cuSPARSE* library function that implements this computation is $cusparseSbsrmv$. The operations in Steps 2 and 3 represent forward and backward substitutions, respectively. These computations can be performed using the *cuBLAS* function $cublasStrsmBatched$.

IV. GPU Implementation

The GPU device is best suited for computations that can be executed concurrently on multiple data elements. In general, a computation is partitioned into thousands of fine-grained

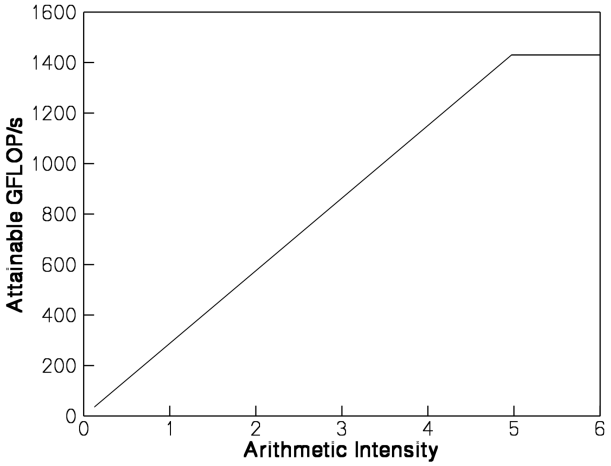

Fig. 5: Roofline model for the NVIDIA® K40 GPU.

operations, which are assigned to thousands of threads on a GPU device for parallel execution. The GPU hardware consists of a number of streaming multiprocessors, which in turn consist of multiple cores. Threads are organized in blocks, or cooperative thread arrays (CTAs), where one or more blocks run on a streaming multiprocessor. The threads in a block are further partitioned into subgroups of 32 threads known as warps. A warp runs on eight or sixteen cores of a streaming multiprocessor in multiple clock cycles.

The initial implementation of the multicolor solver was performed using functions available in the NVIDIA® libraries $cuSPARSE$ and $cuBLAS$. However, numerical experiments showed that the performance of these functions was sub-optimal for matrices representative of those encountered in FUN3D simulations. Therefore, optimized implementations of these functions have been developed for block sizes $1 \leq nb \leq 64$. The existing library functions are used for block sizes $nb > 64$.

Individual kernels have been developed to perform block-sparse matrix-vector products for the ranges $nb \leq 16$ and $16 < nb \leq 64$. These functions are referred to as $cuda\_matvec\_kernel\_1\_16$ and $cuda\_matvec\_kernel\_17\_64$, respectively. The kernel $cuda\_fb\_kernel\_1\_64$ has been developed to solve an array of block triangular systems in the range $nb \leq 64$. The overall structure of the optimized solver is shown in Figure 6. Details of the optimized kernels are presented below.

A. Block-Sparse Matrix-Vector Multiplication

Two algorithms are proposed for the block-sparse matrix-vector multiplication $\mathbf{q_i} = \mathbf{O_i x}$, for $bc(k) \leq i \leq ec(k)$. The first algorithm is for $nb \leq 16$ and the second algorithm is for $17 \leq nb \leq 64$. The first algorithm uses one warp across multiple blocks; the second algorithm uses one or two warps to process a single block.

```
if (nb <= 16 ) then
  do color = 1, nc
    call cuda_matvec_kernel_1_16()
    call cuda_fb_kernel_1_64()
  end do
else if ((nb > 16) .and. (nb <= 64)) then
  do color = 1, nc
    call cuda_matvec_kernel_17_64()
    call cuda_fb_kernel_1_64()
  end do
else
  do color = 1, nc
    call cusparseSbsrmv()
    call cublasStrsmBatched() // forward
    call cublasStrsmBatched() // backward
  end do
end if
```

Fig. 6: Overall structure of the optimized solver.

TABLE I: Parameter values for different block sizes.

nb	nw	nbk	$nrbk$	nb	nw	nbk	$nrbk$
1	1	32	4	9	4	1	1
2	1	38	4	10	4	1	1
3	1	3	4	11	4	1	1
4	1	2	4	12	5	1	1
5	1	1	4	13	6	1	1
6	2	1	2	14	7	1	1
7	2	1	2	15	8	1	1
8	2	1	2	16	8	1	1

1) Algorithm for $nb \leq 16$: A single warp is used to process a variable number of non-zero blocks in a $brow$. The exact number of non-zero blocks processed by a warp is determined by the block size. For a given block size, three parameters are defined that control the allocation of work to a warp. These parameters are as follows:

nw:	number of warps allocated to process a $brow$
nbk:	number of non-zero blocks processed by nw warps
$nrbk$:	number of $brows$ allocated to a CTA

These parameters are depicted graphically in Figure 7 and set using a lookup table based on the values shown in Table I, which have been selected to improve load efficiency and occupancy of the GPU device. The number of threads in a CTA is given by $nw \times nrbk \times 32$.

Consider an example of a block-sparse matrix with a block size of 4. In this case, $nw = 1$, $nbk = 2$, and $nrbk = 4$. A single warp is used to process a row of the block-sparse matrix in a loop, where $nbk = 2$ consecutive non-zero blocks are processed by the warp at each iteration. The warp handles 32 $(2 \times (4 \times 4))$ scalar matrix entries during each iteration. This allows a single warp to load the required elements of the matrix in a coalesced fashion. The appropriate elements of x are also loaded from the read-only data cache, multiplied by the corresponding elements of O, and the results are accumulated. After completion of the loop, the partial results are stored in shared memory to be aggregated at a later time.

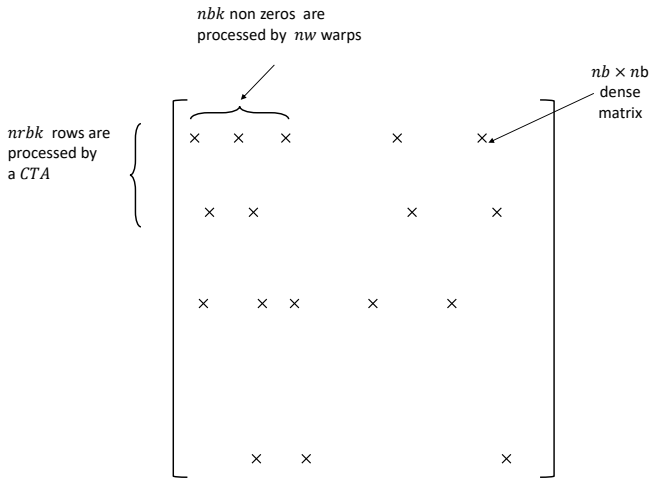

Fig. 7: Processing of a block-sparse matrix by a CTA.

Now consider the details associated with four non-zeros occurring in $brow\ i$. The relevant range of blocks stored in the array O is the segment from $istart$ to $iend$ where $istart = ia(i)$ and $iend = ia(i+1)-1$. More specifically, the first and last non-zero elements of $brow\ i$ are $O(1, 1, istart)$ and $O(4, 4, iend)$, respectively. During the first iteration, a thread $tid \in \{1 \ldots 32\}$ of the warp works with an element of O from the first two non-zero blocks in $brow\ i$. Thread tid multiplies $O(k, l, istart + nbki)$ with $x(l, ja(istart + nbki))$, where

$$\begin{aligned} k &= \mathrm{mod}(tid - 1, 4) + 1 \\ l &= \mathrm{mod}(tid - 1, 16)/4 + 1 \\ nbki &= (tid - 1)/16 \end{aligned}$$

During the second iteration, thread tid multiplies $O(k, l, istart+2+nbki)$ with $x(l, ja(istart+2+nbki))$. Once all non-zero blocks are processed, the partial contributions reside in shared memory as indicated in Figure 8. Note that a CTA has four warps and is processing $nrbk = 4$ consecutive rows of the matrix with one warp for each row. In Figure 8, the value $0 \leq nrbki \leq 3$ is used to identify which of the four rows a warp is processing.

After all threads have stored their partial terms, they must be aggregated. Note that a warp creates a block of 4×8 ($nb \times nb \times nbk$) partial terms which must be aggregated along the second dimension. Four threads of a warp are used to aggregate the terms for a 4×8 block to generate the final four outputs. In total, the first 16 threads for the first warp of the CTA are used to aggregate the partial sums.

2) Algorithm for $17 \leq nb \leq 64$*:* For this range of block sizes, $nw = (nb - 1)/32 + 1$ warps are assigned to process a non-zero block in a $brow\ i$. A CTA consisting of $4 \times nw$ warps processes all non-zero blocks in a $brow$ by working with four non-zero blocks at a time.

```
nbki = (tid-1)/4*4
fk = 0
do j = istart,iend-1, 2
  fk = fk + O(k,l,j+nbki)*x(l,ja(j+nbki))
end do
shared_mem_fk(k+nbki*16+nrbki*32)=fk
```

Fig. 8: Processing of $brow\ i$ in the first matrix-vector product algorithm. The $brow\ i$ has four non-zero blocks.

```
istart = ia(i)
iend   = ia(i+1)-1
fk = 0.0
do j = istart+threadIdx%y, iend, 4
 do i = 1 , nb
  fk=fk + O(threadIdx%x,i,j)*x(i,ja(j))
 end do
end do
shared_mem_ps(threadIdx%x,threadIdx%y)=fk
```

Fig. 9: Processing of $brow\ i$ in the second matrix-vector product algorithm. The $brow\ i$ has eight non-zero blocks.

Consider the example of a block-sparse matrix with a block size of 32. In this case, $nw = 1$, and the CTA (32×4) consists of 4 warps. Now consider a $brow\ i$ with 8 non-zero blocks. The CTA assigned to this row processes all non-zero blocks of $brow\ i$ in two iterations. In the first iteration, warp l of the CTA works with the lth non-zero block of $brow\ i$ for $1 \leq l \leq 4$. During the second iteration, warp l of the CTA works with the $(l + 4)$th non-zero block of $brow\ i$. The appropriate elements of x are also loaded from the read-only data cache, multiplied by the corresponding elements of O, and the results are accumulated. After completion of the loop, the partial results are stored in shared memory to be aggregated at a later time.

Figure 9 shows the processing of $brow\ i$ by the assigned CTA. The value $threadIdx\%x$ is initialized to the thread index within a warp and ranges from 1 to 32, and the value of $threadIdx\%y$ is initialized to the warp number and ranges from 1 to 4. Note that for aggregating the partial sums stored in shared memory, only one of the four warps in the CTA is used.

B. Triangular Solver

For this phase of the solve, all diagonal blocks of $\mathbf{D}$ can be processed in parallel. As outlined in Section III, each diagonal block $\mathbf{D_i}$, $1 \leq i \leq n$, is processed in a forward phase and a backward phase. The forward phase requires the lower triangular portion of the diagonal block $\mathbf{D_i}$, while the upper phase processes the upper portion. An algorithm is proposed that uses one warp to process a diagonal block for both forward and backward phases and accommodates block sizes up to 64. The algorithm is first described for the case $nb \leq 32$, followed by matrices where $33 \leq nb \leq 64$.

1) $nb \leq 32$*:* During the forward phase, columns of the lower triangular portion of $\mathbf{D_i}$ are processed by a warp from

```
k = mod( threadIdx%x-1, nb) + 1
if (threadIdx%x <= nb) f1 = q(k, i)
do j = 1, nb-1
 fb = f1
 fb = __shfl(fb, j)
 fd = 0.0
 if ((threadIdx%x >= j+1) .and. &
 (threadIdx%x<=nb)) fd=D(threadIdx%x,j+1,i)
 f1 = f1 - fd*fb
end do
```

Fig. 10: Processing of a diagonal block $\mathbf{D_i}$ during the forward phase.

```
k = mod( threadIdx%x-1, nb) + 1
if (threadIdx%x <= nb) f1 = q(k, n)
pa = D(threadIdx%x,1,i)
do j = 1, nb-1
 fb = f1
 fb = __shfl(fb, j)
 fd = 0.0
 if ((threadIdx%x >= j+1) .and. &
 (threadIdx%x <= nb)) fd = pa
 pa = D(threadIdx%x,j+1,i)
 f1 = f1 - fd*fb
end do
```

Fig. 11: Processing of a diagonal block $\mathbf{D_i}$ with prefetching during the forward phase.

left to right. The first column, of size $nb-1$, is processed first, followed by the second column, and so forth. Note that a column cannot be processed until the preceding column has been completed. As the loop progresses, smaller columns are encountered, shrinking the amount of parallelism available to a warp. For each column, results from a thread must be made available to other threads. This is achieved using the shuffle instruction provided on the NVIDIA® K40 hardware.

Figure 10 shows a code segment to process a diagonal block $\mathbf{D_i}$. The kernel is called with a CTA of size 32×4. Furthermore, the thread index $threadIdx\%x$ must be initialized to a value between 1 and 32 within a warp, and $threadIdx\%y$ must be initialized to a warp value between 1 and 4.

Figure 11 shows how prefetching of D is also performed to hide memory latency. Experiments have shown performance benefits of $10-15\%$ using this strategy. Finally, the backward phase used to process the upper trianglular portion of a diagonal block $\mathbf{D_i}$ is analogous to the forward phase. An example showing the computation for the backward phase for a diagonal block $\mathbf{D_i}$ is shown in Figure 12.

2) $33 \leq nb \leq 64$*:* In this range, a single warp is still used to process a diagonal block. Consider the forward phase in which the lower triangular region of a diagonal block $\mathbf{D_i}$ is processed. The triangular structure is decomposed into three subregions consisting of upper and lower triangles and a rectangular block as shown in Figure 13. The procedure for the triangular subregions is identical to that outlined earlier for $nb \leq 32$.

```
if (threadIdx%x == nb)f1= f1*D(nb,nb,i)
pa = D(threadIdx%x,nb,i)
do j = nb, 2, -1
  fb = f1
  fb = __shfl(fb, j)
  fd = 0.0
  if (threadIdx%x <= j-1) fd = pa
  pa = D(threadIdx%x,j-1,i)
  f1 = f1 - fd*fb
  if (threadIdx%x == j-1) f1 = f1 *  &
    D(threadIdx%x,threadIdx%x,i)
end do
```

Fig. 12: Processing of a diagonal block $\mathbf{D_i}$ with prefetching during the backward phase. Note that the value of $f1$ is computed in the forward phase.

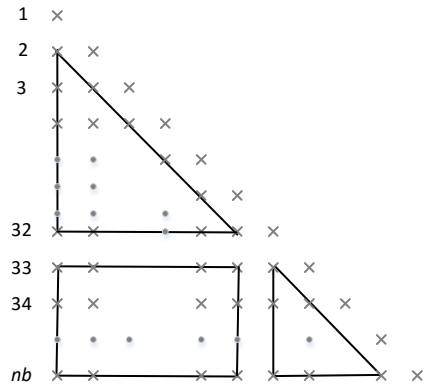

Fig. 13: Partitioning of the lower triangle of a diagonal block of size $nb > 32$ into three parts: upper triangle, rectangle, and lower triangle.

The rectangular subregion is processed in a similar manner; however, the column size is fixed. Therefore, the amount of parallelism available to the warp remains constant for this phase. Processing of the backward phase is similar.

V. Results

The optimized solver has been implemented using CUDA Fortran. Compilation and execution are based on the Portland Group® Fortran Compiler version 15.10 and CUDA Toolkit 7.5. All results have been generated using an Intel® Xeon E5-2670 dual socket, eight-core host processor with an NVIDIA® K40 GPU device.

Performance is evaluated using a series of block-sparse matrices based on tetrahedral meshes ranging in size from $n = 6,309$ to $n = 983,633$ grid points. Specific attributes of each test matrix are shown in Table II. For a given sparsity structure, matrices are populated using random values over a range of block sizes. Due to memory constraints, testing is limited to small values of nb for matrices with large values of n. Timings shown are based on a single traversal of the matrix.

TABLE II: Test matrices.

n	nnz	nb	nc	Max $brows$ in a Color	Min $brows$ in a Color
6,309	85,768	1-100	10	1,076	5
103,178	1,370,396	1-10	10	17,826	15
204,983	2,740,266	1-10	10	35,218	42
394,745	5,309,536	1-10	10	67,748	96
800,322	10,823,162	1-6	11	138,080	1
983,633	13,567,574	1-6	11	166,800	3

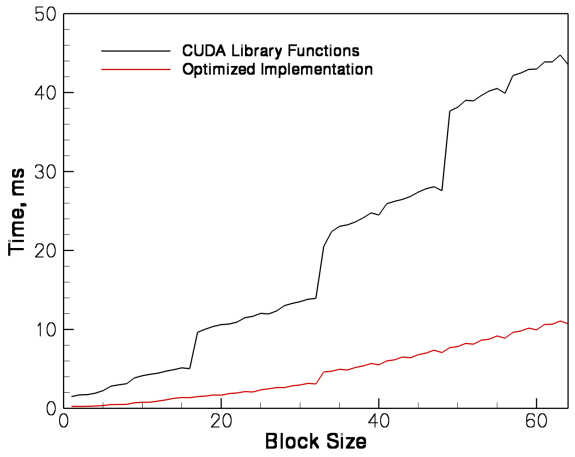

Fig. 14: Performance comparison of an optimized solver with a solver based on $cusparseSbsrmv$ and $cublasStrsmBatched$ for matrix size $6,309$ and block sizes varying from 1 to 64.

The first test is based on a matrix with $n = 6,309$. Due to the relatively small size of the mesh, a broad range of block sizes can be accommodated within the available device memory. The performance of the library-based implementation is shown as the black line in Figure 14. The performance of this implementation shows a linear trend with block size over subintervals of the range $nb \leq 64$; however, abrupt increases in the computational time occur at $nb = 18$, $nb = 34$, and $nb = 50$. Results from the optimized implementation are included as the red curve in the figure. The performance of this implementation is also linear with nb, but is more consistent than the library-based approach. The optimized implementation is considerably faster across the entire range of nb considered, with speedups of up to 7x evident.

The same test case is also used to evaluate the integrated solver up to $nb = 100$. Here, the optimized implementation is invoked for $nb \leq 64$, while the library-based implementation is used for $65 \leq nb \leq 100$. Results are shown in Figure 15. The red curve previously shown in Figure 14 is recovered for $nb \leq 64$, after which a dramatic increase in cost is observed as the library-based implementation is used for larger values of nb.

Results for the larger-matrix test cases shown in Table II are limited to smaller block sizes due to memory constraints. Table III summarizes the performance of the two implementations for the next three larger matrix sizes, across the range $nb \leq 10$. The optimized implementation outperforms the

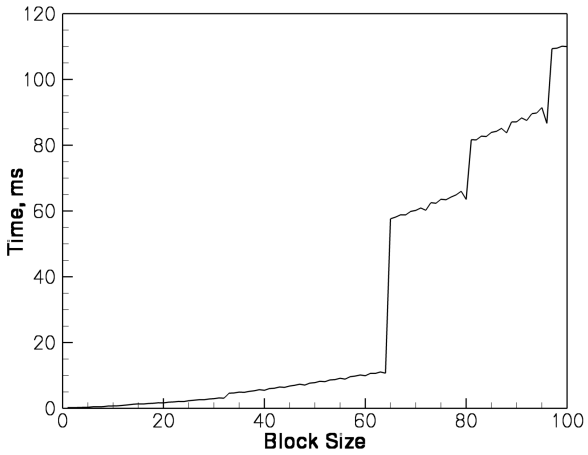

Fig. 15: Performance of an integrated solver, where optimized kernels are used for block sizes below 65 and existing library functions are used for larger block sizes.

library-based approach by factors similar to those observed in the smaller-matrix test case. Similarly, results for $nb \leq 6$ using the largest two matrices with $n = 800,322$ and $n = 983,633$ are included in Table IV. Again, the optimized solver is substantially more efficient than the solver based on existing library functions.

VI. Summary and Future Work

An optimized implementation of a multicolor point-implicit solver for unstructured grid applications executing on a single GPU device has been developed. Factors critical to the performance of the underlying matrix-vector multiplication and forward and backward substitution operations have been identified, and several challenges specific to the target application have been addressed. Efforts have been made to ensure coalesced memory loads, to achieve optimal use of shared memory and pre-fetching strategies, to minimize thread divergence within warps, and to leverage specialized shuffle instructions. Speedups of as much as 7x over an implementation based on existing CUDA library functions have been observed for a broad range of matrix and block sizes relevant to the target application.

To impact practical applications, the methodology developed in the current work must be extended to accommodate multiple devices across leadership-class distributed memory systems. In these environments, the baseline implementation relies on conventional domain decomposition and standard message passing techniques. The serial algorithm is recovered by performing halo exchanges of the updated values of the unknowns upon completion of each color group. The message passing infrastructure within FUN3D has been extended to accommodate Remote Direct Memory Access calls between GPU devices, and extension of the solver approach developed here is a focus of ongoing efforts. Finally, to minimize costly host/device transfers, GPU implementations of other critical kernels across FUN3D are also being examined.

TABLE III: Performance comparison of an optimized solver with a solver based on $cusparseSbsrmv$ and $cublasStrsmBatched$ for medium-sized matrices with block sizes varying from 1 to 10.

nb	$n = 103,178$		$n = 204,983$		$n = 394,745$	
	Baseline (ms)	Optimized (ms)	Baseline (ms)	Optimized (ms)	Baseline (ms)	Optimized (ms)
1	9.3	1.5	17.2	2.7	31.4	5.1
2	11.8	1.6	20.5	3.0	39.1	5.6
3	14.7	1.9	24.4	3.6	47.5	6.8
4	16.2	2.2	27.7	4.3	52.5	8.1
5	19.6	2.9	35.3	5.5	66.5	10.4
6	27.1	4.8	49.9	9.3	96.5	17.8
7	28.5	5.0	55.1	9.7	104.7	18.7
8	30.3	5.1	57.3	10.0	107.9	19.2
9	42.8	8.4	81.7	16.5	155.9	31.8
10	45.7	9.0	85.8	17.8	165.9	34.3

TABLE IV: Performance comparison of an optimized solver with a solver based on $cusparseSbsrmv$ and $cublasStrsmBatched$ for large-sized matrices with block sizes varying from 1 to 6.

nb	$n = 800,322$		$n = 983,633$	
	Baseline (ms)	Optimized (ms)	Baseline (ms)	Optimized (ms)
1	61.9	10.1	74.9	12.5
2	78.2	11.1	92.7	13.8
3	93.0	13.7	112.1	16.9
4	104.6	16.3	130.0	20.0
5	126.8	21.1	164.9	25.7
6	188.7	36.1	236.0	44.2

ACKNOWLEDGMENTS

The authors wish to recognize Dominik Ernst of the Friedrich-Alexander University Erlangen-Nürnberg for his initial efforts to optimize the solver kernel for a fixed block size of five. The support of the NASA Langley Research Center Comprehensive Digital Transformation initiative and Dr. Mujeeb Malik, Technical Lead for the Revolutionary Computational Aerosciences sub-project within the NASA Aeronautics Research Mission Directorate Transformational Tools and Technologies Project, is also acknowledged.

REFERENCES

[1] R. T. Biedron *et al.*, "FUN3D Manual 12.9," NASA/TM-2016-219012, 2016.

[2] R. Li and Y. Saad, "GPU-Accelerated Preconditioned Iterative Linear Solvers," *The Journal of Supercomputing*, vol. 63, no. 2, pp. 443–466, 2013. [Online]. Available: http://dx.doi.org/10.1007/s11227-012-0825-3

[3] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 917–924, Jul. 2003. [Online]. Available: http://doi.acm.org/10.1145/882262.882364

[4] Y. Liu and B. Schmidt, "LightSpMV: Faster CSR-Based Sparse Matrix-Vector Multiplication on CUDA-Enabled GPUs," in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, July 2015, pp. 82–89.

[5] N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan, "Automatic Selection of Sparse Matrix Representation on GPUs," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: ACM, 2015, pp. 99–108. [Online]. Available: http://doi.acm.org/10.1145/2751205.2751244

[6] H.-V. Dang and B. Schmidt, "CUDA-Enabled Sparse Matrix-Vector Multiplication on GPUs Using Atomic Operations," *Parallel Computing*, vol. 39, no. 11, pp. 737 – 750, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819113001178

[7] 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*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 18:1–18:11. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654078

[8] X. Yang, S. Parthasarathy, and P. Sadayappan, "Fast Sparse Matrix-Vector Multiplication on GPUs: Implications for Graph Mining," *Proc. VLDB Endow.*, vol. 4, no. 4, pp. 231–242, Jan. 2011. [Online]. Available: http://dx.doi.org/10.14778/1938545.1938548

[9] 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*, ser. SC '07. New York, NY, USA: ACM, 2007, pp. 38:1–38:12. [Online]. Available: http://doi.acm.org/10.1145/1362622.1362674

[10] NVIDIA-II, "cuSPARSE User Guide," http://docs.nvidia.com/cuda/cusparse/index.html#axzz4Ggg2gNCX, 2015, [Online; accessed 01-Aug-2016].

[11] M. Naumov, "Incomplete-LU and Cholesky Preconditioned Iterative Methods Using cuSPARSE and cuBLAS," http://developer.download.nvidia.com/assets/cuda/files/psts_white_paper_final.pdf?autho=1463929612_204e4e3eae2e5b25cb84923a15bf1e4e&file=psts_white_paper_final.pdf, 2011, [Online; accessed 22-May-2016].

[12] NVIDIA-I, "cuBLAS User Guide," http://docs.nvidia.com/cuda/cublas/index.html#axzz4Ge8iuMcz, 2015, [Online; accessed 01-Aug-2016].

[13] E. Cuthill and J. McKee, "Reducing the Bandwidth of Sparse Symmetric Matrices," in *Proceedings of the ACM National Conference*, July 1969, pp. 157–172.

[14] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelphia, PA: SIAM, 2003.