Memory Optimizations for Sparse Linear Algebra on GPU Hardware

Aaron Walden
Computational AeroSciences Branch
NASA Langley Research Center
Hampton, VA, USA
aaron.c.walden@nasa.gov

Mohammad Zubair
Department of Computer Science
Old Dominion University
Norfolk, VA, USA
zubair@cs.odu.edu

Christopher P. Stone
National Institute of Aerospace
Hampton, VA, USA
christopher.stone@nianet.org

Eric J. Nielsen
Computational AeroSciences Branch
NASA Langley Research Center
Hampton, VA, USA
eric.j.nielsen@nasa.gov

***Abstract*—An effort to maximize memory bandwidth utilization for a sparse linear algebra kernel executing on NVIDIA® Tesla V100 and A100 Graphics Processing Units (GPUs) is described. The kernel consists of a block-sparse matrix-vector product and a series of forward/backward triangular solves. The computation is memory-bound and exhibits low arithmetic intensity. Along with a relatively small block size, the data layout poses a challenge to effectively utilize the available memory bandwidth on common GPU architectures. An earlier implementation using a warp to process a single row of the matrix was found to yield good memory performance on the V100 architecture. However, a new approach, which assigns a warp to six rows of the matrix, is proposed for the A100. In addition, two new features offered by the A100 architecture are explored. *L2* residency control enables a portion of the *L2* cache to be used for persistent data access, and the asynchronous copy instruction allows data to be loaded directly from main memory into shared memory. Demonstrations show that the new implementation improves memory bandwidth utilization from 71.5% to 81.2% of the peak available on the A100 architecture.**

***Index Terms*—memory bandwidth, Graphics Processing Unit, cache, asynchronous**

I. Introduction

FUN3D is a suite of computational fluid dynamics software 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 [1]. 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 nonlinear 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. This effort focuses on the multicolor point-implicit linear solver that accounts for a significant fraction of the overall runtime in virtually all FUN3D simulations.

The solver kernel consists of a block-sparse matrix-vector product and a series of forward/backward triangular solves. The dominant computation is a block-sparse matrix-vector product; for a broad range of applications encountered in practice, 5×5 blocks are common. The off-diagonal matrix coefficients are stored in a compressed sparse row (CSR) format [2], where two integer arrays capture the sparsity pattern of the nonzero blocks in the matrix. The nonzero blocks in a row are stored contiguously in memory, and the scalar entries within a block are stored in column-major order. The kernel is memory-bound with a low arithmetic intensity. In such cases, it is critical to understand the increasingly complex memory hierarchies of today's advanced architectures and how memory bandwidth and potential reuse of computations can be effectively leveraged. For example, in the case of an NVIDIA® Graphics Processing Unit (GPU), it is essential to understand how to accommodate the application data layout and restructure the solver algorithm to utilize the registers, shared memory, $L1$ and $L2$ caches, and DRAM effectively.

The data layout of the sparse matrix in memory, along with relatively small blocks, poses a challenge for GPU architectures to utilize the memory bandwidth effectively. Modern GPUs support the Single Instruction Multiple Thread (SIMT) model, with a group of threads referred to as a warp. The dimension of this thread group can vary from one GPU architecture to another, and the group must process consecutive memory locations to achieve coalesced memory accesses (Figure 1). This requires mapping a warp to one or more blocks of a sparse matrix and restructuring the computation accordingly. In summary, restructuring the computation is essential, and in some cases, modifications to the underlying data layout may even be required.

Many researchers have studied efficient implementations of iterative solvers on GPUs [3], [4], and considerable effort has focused on the optimization of the underlying sparse matrix-vector product often required. Some representative work for sparse matrix-vector products can be found in Refs. [5]–[11].

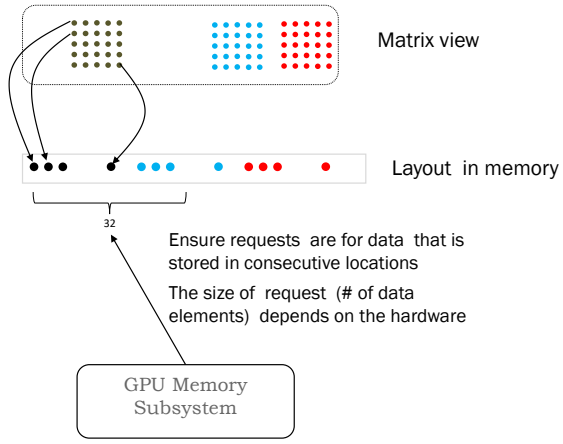


Fig. 1: Structuring the computation to ensure coalesced memory accesses.

In addition, some implementations have been developed using *cuSPARSE* library functions [12], [13]. An algorithm for high-performance block-sparse matrix-vector products for PDE-based applications on multiple GPUs has been addressed in Ref. [14].

In our early work to optimize performance for block-sparse matrix-vector products, an implementation that allocates a number of warps to process a subset of the blocks in one row of the sparse matrix was proposed [15]. Here, we refer to this implementation as the baseline implementation. Several challenges were 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, effort was focused on coalesced memory loads, the use of shared memory and prefetching, minimal thread divergence within warps, and strategic use of shuffle instructions available on recent hardware. This baseline implementation of the block-sparse matrix-vector product showed performance gains of up to 7x over the existing CUDA library functions available in *cuSPARSE* when running on an NVIDIA® Tesla K40. In more recent work, the baseline implementation was evaluated on the NVIDIA® Tesla V100 and A100 architectures (hereafter referred to as V100 and A100). It was found that the implementation does not require modification on the V100 and yields 78% of the peak bandwidth available on that hardware.[1] However, on the A100, the baseline implementation yields just 71.5% of the peak bandwidth.

For the A100 GPU, we propose a new implementation that results in improved performance over the baseline. The major architectural features that improve the performance of the new implementation include higher HBM2 memory bandwidth, larger cache sizes, and $L2$ residency support. The A100 has a peak memory bandwidth of 1555 GB/s, a 73% increase from the 900 GB/s available on the V100. The $L2$ cache is 40 MB, which is 7x larger than that of the V100. Moreover, the $L2$ cache bandwidth is 2.3x that of the V100. The A100 also supports $L2$ cache residency controls that enable a portion of the $L2$ cache to be used for persistent data access. The combined size of the $L1$ cache and shared memory is 192 KB, which is 1.5x larger than that of the V100. Furthermore, the A100 supports asynchronous copy instructions that load data directly from device memory into shared memory, optionally bypassing the $L1$ cache. In the current effort, these features are leveraged to improve the memory performance of the baseline implementation from 1104 GB/s to 1262 GB/s, which represents 81% of the peak memory bandwidth on the A100.

The remainder of the paper is organized as follows. First, details of the multicolor point-implicit algorithm are presented in Section II. The test case used for evaluating the performance of different implementations is discussed in Section III. In Section IV, specific performance concerns for the GPU implementation are introduced. For completeness, we review the baseline implementation of the solver for prior NVIDIA® GPUs in Section V. Section VI describes the new implementation optimized for the A100 architecture. Performance results are presented in Section VII, and conclusions are provided in Section VIII.

II. Multicolor Point-Implicit Solver

For a spatial mesh containing n grid vertices, the implicit approach used within FUN3D requires frequent solutions of a large $n \times n$ linear system of block equations. The linear system is of the form $\mathbf{A}\mathbf{\Delta Q} = \mathbf{R}$, where $\mathbf{R}$ represents the vector of discrete nonlinear residual equations and $\mathbf{A}$ is an $n \times n$ block-sparse matrix composed of dense $n_b \times n_b$ blocks. The quantity $\mathbf{\Delta Q}$ is the vector of unknowns required to advance the nonlinear solution $\mathbf{Q}^k$ at time-level k to $k+1$. The coefficient matrix $\mathbf{A}$ is based on a strictly nearest-neighbor stencil. To provide flexibility in the implementation, $\mathbf{A}$ is additively split into diagonal and off-diagonal components that are stored separately, namely,

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

where $\mathbf{D}$ and $\mathbf{O}$ represent the diagonal and off-diagonal blocks of $\mathbf{A}$, respectively. The implementation in FUN3D uses 32-bit precision for $\mathbf{O}$ and $\mathbf{\Delta Q}$, while 64-bit precision is used for $\mathbf{D}$ and $\mathbf{R}$. An FP16 representation of $\mathbf{A}$ was also explored in [16].

The block-sparse $n \times n$ matrix $\mathbf{O}$ contains nnz nonzero $n_b \times n_b$ blocks that are stored using a CSR format. Each of the n rows and columns containing $n_b \times n_b$ blocks are referred to as a $brow$ and a $bcol$, respectively. Two integer arrays iam and jam are used to efficiently capture the sparsity pattern of the matrix. The array iam is a rank-1 array of size $n+1$ whose i-th entry indicates the leading nonzero block index in the i-th $brow$ of $\mathbf{O}$. The array includes an entry in the $n+1$ location to facilitate traversal of the elements through the n-th $brow$. The jam array is a rank-1 array of size nnz that provides the $bcol$

[1] Since the algorithm is memory bound, we evaluate the performance of a particular implementation by observing the percentage of theoretical peak bandwidth obtained.

$$\begin{bmatrix} & & \times & \times \\ & & \times & \\ \times & \times & & \\ \times & & & \end{bmatrix}$$

(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 nonzero block. Figure (b) shows $\mathbf{O}$ for a block size of 2×2.

Algorithm 1 MULTICOLOR LINEAR SOLVER

1: $\Delta\mathbf{Q} = 0$
2: **for** $i \leftarrow 1$ **to** n_{iter} **do**
3: **for** $c \leftarrow 1$ **to** n_c **do**
4: $\Delta\mathbf{r} \leftarrow \mathbf{R_c} - \mathbf{O_c}\Delta\mathbf{Q}$
5: $\Delta\mathbf{Q_c} \leftarrow \mathbf{D_c^{-1}}\Delta r$
6: **end for**
7: **end for**

index for each nonzero block. A third array is used to store the block entries proceeding from $iam(1)$ to $iam(n+1)-1$, where the scalar entries within each $n_b \times n_b$ block are stored in column-major order. Figures 2 (a) and 2 (b) show a sample block-sparse matrix with $n_b = 2$ and the corresponding CSR arrays, respectively.

Several linear-solver options are provided within FUN3D; the scheme most commonly used in practice is the multicolor point-implicit relaxation [15], [16] shown in Algorithm 1. In this approach, the grid vertices are grouped into n_c color groups, such that no two adjacent vertices are assigned the same color. Typical values of n_c for meshes encountered in practice are 10-15. Since the matrix $\mathbf{A}$ involves only a nearest-neighbor stencil, unknowns within a color may be updated in parallel in a Jacobi-like fashion. Color groups are processed sequentially, where solution updates within each color depend solely on the latest values of $\mathbf{\Delta Q}$ in neighboring color groups. The overall process may be repeated using n_{iter} sweeps over the entire system. Exact solutions to the linear system are not sought in practice; small values of n_{iter}, which provide suitable convergence of the nonlinear solution, are generally used.

To improve memory performance, the system of equations is renumbered such that unknowns within a color appear in consecutive order. In Algorithm 1, $\mathbf{O_c}$ and $\mathbf{D_c}$ represent

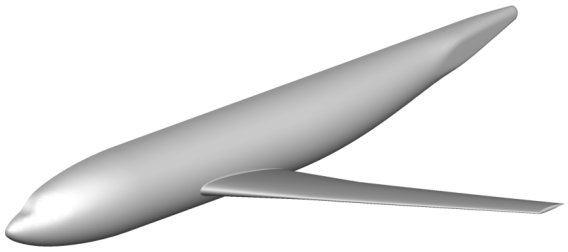


Fig. 3: Wing-body configuration taken from Ref. [17].

submatrices of $\mathbf{O}$ and $\mathbf{D}$, respectively, for the unknowns contained in color c. $\mathbf{R}_c$ represents the nonlinear residual subvector defined by unknowns belonging to color c. Line 4 of Algorithm 1 represents a standard block-sparse matrix-vector product. Line 5 requires an inversion of each $n_b \times n_b$ block of the matrix $\mathbf{D_c}$. Here, a lower-upper (LU) decomposition of these blocks is computed beforehand and stored in place. The solution for the current block row is then obtained through a forward-backward substitution procedure. Throughout this work, the terms *block row* and *row* are used interchangeably, both referring to a matrix row of $n_b \times n_b$ dense blocks.

In addition to the shared-memory programming model presented here, the solver also accommodates an MPI message-passing approach using a standard domain-decomposition strategy for architectures with multiple sockets and/or multiple NUMA domains, as well as general multinode, distributed-memory environments necessary for large-scale simulations. To recover the serial algorithm using this approach, halo exchanges of partition boundary data are performed before processing the next color. To hide communication latencies associated with these halo exchanges each color group is further subdivided into values along partition boundaries and those remaining values lying entirely interior to the partition. When processing the unknowns within a color group, values along partition boundaries are determined first, then non-blocking MPI calls are used to initiate halo exchanges with neighboring partitions. Values interior to the partition are then evaluated while halo values are in flight. At the completion of the current color, each process waits for communication to complete prior to initiating the next color.

III. TEST CASE

The test case used here is based on transonic turbulent flow over the semispan wing-body [17] configuration shown in Fig. 3. The freestream Mach number is 0.85, the angle of attack is zero degrees, and the Reynolds number, based on the mean aerodynamic chord, is 5 million. The computational mesh consists of 1.1 million grid vertices, 1.2 million prisms, 3.0 million tetrahedra, and 7.3 thousand pyramids. This problem size is representative of the workload that would typically be placed on a single compute node in practice. For the purposes of the current study, a single linear system with $n_b = 5$ is extracted from an arbitrary time step during the nonlinear convergence of the mean flow equations. The linear system

contains a total of 19 million nonzero off-diagonal blocks, or an average of approximately 17 off-diagonal blocks per mesh vertex. Timings reported below correspond to $n_{iter} = 15$.

IV. GPU-Related Performance Concerns

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 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 (SMs), which in turn consist of multiple cores. Threads are organized in blocks, or cooperative thread arrays (CTAs), where one or more blocks run on an SM. On NVIDIA® GPUs, the threads in a block are further partitioned into subgroups of 32 threads, or warps. Threads within the same block are able to explicitly synchronize and have access to a small (e.g., 48 KB) but fast shared-memory scratchpad that is local to each SM. This facilitates efficient sharing of data across threads in the block. Threads within the same warp can exchange data directly using register shuffles, which are generally faster than shared-memory accesses, and can synchronize more efficiently than at the block level.

One of the significant challenges in achieving high performance for the current algorithm is its low arithmetic intensity. The block-sparse matrix-vector product dominates the overall execution time and is characterized by an arithmetic intensity of approximately 0.5, resulting in a memory-bound scenario on the GPU architecture. A naive implementation might map a GPU thread to process a single row block. To process rows of the same color concurrently, one could launch a GPU kernel with as many threads as the number of row blocks in that color. Such an approach generally yields very poor performance, exhibiting approximately 8% of peak memory bandwidth utilization on the V100. The roofline model shown in Fig. 4 illustrates qualitatively the shortcomings of such an approach, where the low achieved bandwidth ultimately results in very low computational throughput. The objective is therefore to design a kernel that can maximize performance by utilizing the available memory bandwidth as effectively as possible. Increasing the arithmetic intensity of the kernel by enabling efficient reuse of vector elements available in cache offers an opportunity to further improve performance.

V. Baseline GPU Implementation

To develop an efficient GPU implementation of the multi-color point-implicit solver, functions provided by the *cuSPARSE* [12] and *cuBLAS* [18] libraries were initially 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 an LU-decomposition of the diagonal block. Experiments showed that this approach yields suboptimal performance for linear systems representative of those encountered in typical FUN3D simulations.

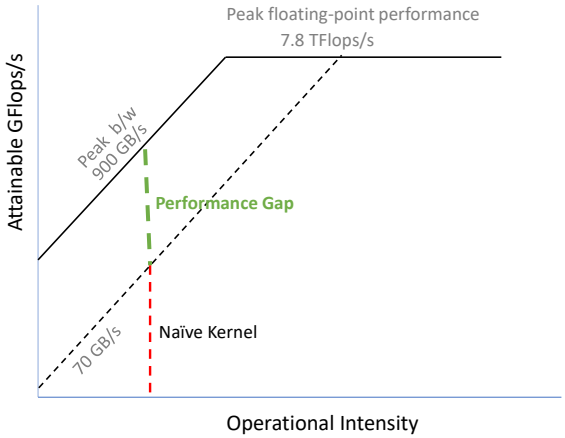
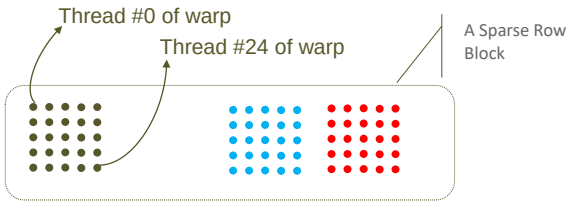


Fig. 4: Roofline model for V100 and a naive kernel implementation.

Instead, optimized CUDA implementations of these functions were developed in [15]. To perform a block-sparse matrix-vector product, the proposed algorithm allocates a number of warps to process a subset of the blocks in a single row of the sparse matrix. The mapping of a warp to process a block of a sparse matrix with $n_b = 5$ is illustrated in Figure 5. To perform forward and backward substitutions, a second kernel is invoked that assigns a single warp to process one diagonal block. Several challenges were 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 was placed on coalesced memory loads, the use of shared memory and prefetching, minimal thread divergence within warps, and strategic use of shuffle instructions available on recent hardware. Depending on the value of n_b, the new implementations realized performance gains of up to 7x over existing *cuSPARSE* and *cuBLAS* library functions.

VI. NVIDIA® A100 GPU Implementation

A new implementation that distributes workload amongst threads in a different manner is proposed. Recall that a warp is used to process a single row-block of the matrix in the baseline approach. In the new formulation, a warp is used to process six row blocks of the sparse matrix, as shown in Figure 6. Note that five consecutive warp threads are assigned to process a row block in the new algorithm. In this manner, 30 threads of a warp are utilized, resulting in a higher efficiency of $30/32$ compared to that of the baseline approach, $25/32$. In addition, the new algorithm works with fewer blocks than the baseline algorithm for identical block sizes. Consequently, the new algorithm executes fewer rounds of thread blocks on an SM. Instead, each thread in a block is doing more work,




```
k = threadIdx.x % 5;
l = threadIdx.x / 5;
n = start+blockIdx.x*blockDim.y+threadIdx.y-1;

if ( (n < end) && (l < 5)) {
    istart = iam[n];
    iend = iam[n + 1] - 1;
    for (j=istart-1 ; j < iend; j++) {
        colid = jam[j];
        fk += A_OFF(k, l, j)*DQ(l,colid-1);
    }
    sm_f[k][l][threadIdx.y] = fk;
}
.
.
```

Fig. 5: Assignment of a warp to process a complete 5×5 block to ensure that consecutive threads of the warp load and process data from consecutive locations of device memory. The 25 active threads of a warp process a complete row one block at a time, and aggregate partial results into a 5×5 block. Note that the active threads are selected by the *if* construct. The columns of the final aggregated block are reduced using shuffle instructions or shared memory (not shown here).

and at any given time, there are many more active memory requests than the baseline implementation.

The code segment that highlights the kernel computation is shown in Figure 7. Note that a thread loads five values of row k of the dense 5×5 block, lines $8 - 12$, along with the five vector values to be multiplied. After multiplication, the five partial results are stored in registers $fk0$ to $fk4$. The partial results are accumulated for all dense blocks in a row inside the *for* loop. At the end of the loop, the five consecutive warp threads collectively hold a dense 5×5 block of accumulated results. The accumulated results in registers $fk0$ to $fk4$ are aggregated in register fk (line 14).

We now discuss how a bigger $L2$ cache on the A100 architecture helps this implementation. Initially, we assume that a warp performs aligned accesses. The impact of misaligned accesses will be considered later. At the beginning of the iteration, each active thread of the warp loads an element of the sparse matrix into register $fk0$ (line 8). This results in a warp loading six cache lines into $L2$ that are far apart in memory, while only five values from the beginning of each cache line are used. In subsequent processing, lines 9-12, the rest of the cache line values in $L2$ are used by the warp. A bigger $L2$ is able to hold these cache lines for subsequent consumption before evicting them.

The impact of alignment is now considered for the specific case of 5×5 dense blocks (25 values). Clearly, misaligned accesses will occur while processing dense blocks in a row.

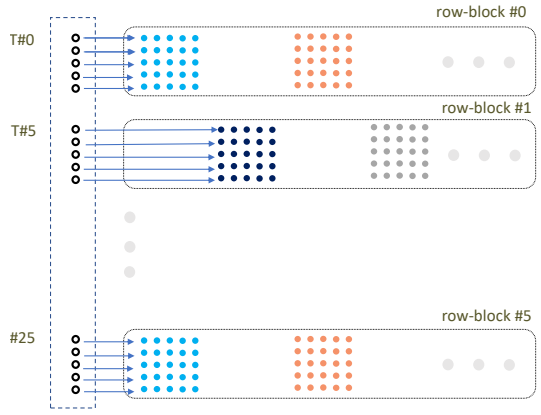


Fig. 6: Assignment of a warp to process 6 row blocks. Five consecutive threads of a warp process a row block. A total of 30 threads of a warp are active.

```
1  // sub-matrix row index
2  int k = threadIdx.x % 5;
3  // Block row index offset by block/warp.
4  int brow = threadIdx.x / 5 + 6 * blockIdx.x;
5  int istart = iam[brow], iend = iam[brow+1];
6  for (int j = istart; j < iend; j++) {
7    bcol = jam[j] - 1;
8    fk0 += A_OFF(k, 0, j) * DQ(0, bcol);
9    fk1 += A_OFF(k, 1, j) * DQ(1, bcol);
10   fk2 += A_OFF(k, 2, j) * DQ(2, bcol);
11   fk3 += A_OFF(k, 3, j) * DQ(3, bcol);
12   fk4 += A_OFF(k, 4, j) * DQ(4, bcol);
13 }
14 fk = fk0 + fk1 + fk2 + fk3 + fk4;
```

Fig. 7: Code segment of the kernel to illustrate how five threads of a warp cooperate to process a row block.

Since a cache line is 128 bytes, an aligned access only requires a single cache line to process a dense block. However, most accesses will be misaligned, and as a result we will be fetching two cache lines to process a dense block. However, the fetched values in $L2$ that are not being used in the current iteration of the loop will be used in the subsequent iteration. In this manner, the effects of misaligned accesses are minimized.

A. *Storing Persistent Data in L2*

For the block-sparse matrix-vector operation, the sparse matrix values are only used once. However, there is a reuse of the vector values, and enabling the vector data to persist in the $L2$ cache offers an opportunity to reduce the memory traffic between device memory and $L2$. The A100 architecture offers a new feature that allows a portion of the $L2$ cache to perform persistent data accesses to device memory, which ultimately enables higher bandwidth and lower latency accesses to device memory.

On the A100, the CUDA version 11 toolkit offers API functions to set aside a portion of the 40-MB $L2$ cache to perform persistent data accesses to global memory. Note that these accesses have priority use of the allocated $L2$ cache;

```
cudaDeviceProp prop;
cudaGetDeviceProperties( &prop, device_id);
// set-aside 3/4 of L2 cache for persisting accesses
size_t size = min( int(prop.l2CacheSize * 0.75),
              prop.persistingL2CacheMaxSize );
cudaDeviceSetLimit(
    cudaLimitPersistingL2CacheSize,
                    size);
// Stream level attributes data structure
cudaStreamAttrValue stream_attribute;
// Global Memory data pointer
stream_attribute.accessPolicyWindow.base_ptr =
            reinterpret_cast<void*>(dq_dev);
// Number of bytes for persisting access
stream_attribute.accessPolicyWindow.num_bytes =
            (1123718 * 1.0 )  * 5 * sizeof(float);
stream_attribute.accessPolicyWindow.hitRatio = 1.0;
```

Fig. 8: Setting accesses to $\mathbf{\Delta Q}$ in global memory to persist in the $L2$ cache.

however, conventional accesses may use this reserved $L2$ cache when unused by persistent accesses. The stream level attributes data structure is used to set the region of the device memory which will persist in $L2$ cache when initially accessed as shown in Figure 8. The starting address in global memory is specified along with the number of bytes. If some fraction of global memory accesses are designated as persistent, the value of the hit ratio must be specified. A value of 1 is used here.

B. Prefetching the $\mathbf{\Delta Q}$ *Vector*

Two shared memory buffers are used to prefetch the $\mathbf{\Delta Q}$ vector. While processing a dense 5×5 block, the algorithm reads the required $\mathbf{\Delta Q}$ values from a preloaded buffer. The $\mathbf{\Delta Q}$ values needed for processing the next dense block are simultaneously loaded into the second buffer. These buffers are then toggled and the process continues until completion. The code segment that highlights this prefetching is shown in Figure 9.

C. Asynchronous Copy

The A100 GPU supports an asynchronous copy instruction that loads data in the background while computations are occurring. Moreover, the data are loaded directly from global memory into shared memory without the use of intermediate registers. This feature has the potential to hide memory latency and reduce register file usage. Here, the capability has been used for the forward/backward substitution procedure that follows the block-sparse matrix-vector product in the solver algorithm. Loading the diagonal block required for forward/backward substitution is overlapped with the matrix-vector computation using this asynchronous memory copy operation, as shown in Figure 10.

```
// number of row blocks processed by a warp
const int NRWP = 6;
__shared__ float s_dq[2][5 * NRWP];
float* dq_next = s_dq[0], dq_cur = NULL;
// sub-matrix row (k) and block row (r)
int k = threadIdx.x % 5, r = threadIdx.x / 5;
dq_next[k * NRWP + r] = DQ(k, jam[istart]-1);
int bind = 1;
for ( j = istart; j < iend - 1; j++) {
  dq_cur = s_dq[(bind - 1) % 2];
  // prefetch next index
  auto bcol = jam[j+1] - 1;
  dq_next = s_dq[bind % 2];
  fk0 += A_OFF(k,0,j) * dq_cur[0 * NRWP + r];
  fk1 += A_OFF(k,1,j) * dq_cur[1 * NRWP + r];
  fk2 += A_OFF(k,2,j) * dq_cur[2 * NRWP + r];
  fk3 += A_OFF(k,3,j) * dq_cur[3 * NRWP + r];
  dq_next[k * NRWP + r] = DQ(k, bcol);
  fk4 += A_OFF(k,4,j) * dq_cur[4 * NRWP + r];
  bind = bind + 1;
}
dq_cur = s_dq[(bind - 1) % 2];
j = iend - 1;
fk0 += A_OFF(k,0,j) * dq_cur[0 * NRWP + r];
fk1 += A_OFF(k,1,j) * dq_cur[1 * NRWP + r];
fk2 += A_OFF(k,2,j) * dq_cur[2 * NRWP + r];
fk3 += A_OFF(k,3,j) * dq_cur[3 * NRWP + r];
fk4 += A_OFF(k,4,j) * dq_cur[4 * NRWP + r];
```

Fig. 9: Prefetching $\mathbf{\Delta Q}$ using shared memory.

```
namespace cg = cooperative_groups;
auto block = cg::this_thread_block();
__shared__ double s_a_diag_lu[NRWP*25];

int rowid_local = lane_id / nb ;
int n = start + blockIdx.x  * NRWP +
    rowid_local - 1;
int n0 = start + blockIdx.x  * NRWP - 1;
if ((n < end) && ((lane_id < NRWP * nb))) {
  auto active_block = cg::coalesced_threads();
  // prefetch diag matrix values into shared
    memory
  cg::memcpy_async(
    active_block,
    s_a_diag_lu,
    &a_diag_lu[n0 * 25],
    sizeof(double) * 25 * NRWP
  );
  // prefetching overlaps with row block SpMV
  for ( j = istart; j < iend - 1; j++) {
    // process a row block
  }
  // wait for prefetch access to complete
  cg::wait(active_block);

  // use a_diag_lu_shared here
}
```

Fig. 10: Asynchronous memory copy to hide latency.

TABLE I: Designation for various implementations.

Designation	Approach
11_0	One warp per row
11_1	11_0 with $L2$ residence control
11_2	11_1 with asynchronous memory copy
16_0	One warp per six rows
16_1	16_0 with $L2$ residence control
16_2	16_1 with asynchronous memory copy

VII. Results

To evaluate the performance of different mapping algorithms and the new features of the A100 hardware, six variants of the solver kernel were explored as outlined in Table I. Recall that $L2$ persistency and asynchronous memory transfers to shared memory are not available on the V100 architecture. Experiments were performed using V100 and A100 GPUs with 16 and 40 GB of memory, respectively. Computations using the V100 were based on version 11.0 of the CUDA toolkit, while the A100 results were generated using version 11.2. The correctness of outputs was validated against CPU-generated data using a checksum. To determine execution time, the minimum time observed over ten successive executions of the kernel was used.

Table II shows the overall performance in milliseconds (ms) for each of the kernel variants on both architectures. The baseline algorithm yields results of 48.6 ms and 30.9 ms on the V100 and A100, respectively. The use of $L2$ persistency and asynchronous memory copies improve the result on A100 to 27.6 ms. When a warp is applied to six rows of the matrix, performance for both GPUs is degraded; however, the options to leverage $L2$ persistency and asynchronous memory copies enable improved timings. The most performant approach is the six-row formulation on the A100 with the use of $L2$ control. Note that all tests using $L2$ control kept the complete $\mathbf{\Delta Q}$ vector persistent in $L2$.

The results shown in Table III indicate that the performance of the original mapping algorithm designated as 11_0 is lower on the A100 than the V100 in terms of bandwidth efficiency. On the V100, 78.6% of the peak bandwidth was observed, but only 71.5% was noted on the A100. However, when the new features available on the A100 are used, the performance improves from 71.5% to 80.1%.

The A100 performance improvement is primarily due to the $L2$ residency control feature. A further slight improvement is observed when using asynchronous memory copies. The new mapping algorithm and $L2$ residency control give an overall performance of 81.2%, slightly better than the original mapping algorithm with the new A100 features. Attempts were made to measure the impact of the $L2$ residency control feature on memory traffic; however, these attempts were unsuccessful since the NVIDIA profiler clears the $L2$ cache after multiple passes through the kernel to gather data.

TABLE II: Execution time in milliseconds for various implementations on V100 and A100.

	11_0	11_1	11_2	16_0	16_1	16_2
V100	48.6	-	-	60.5	-	-
A100	30.9	27.8	27.6	31.6	27.2	28.2

TABLE III: Memory bandwidth performance for various implementations on V100 and A100 in GB/s and percentage of peak.

	11_0	11_1	11_2	16_0	16_1	16_2
V100	707	-	-	568	-	-
	(78.6%)	-	-	(63.1%)	-	-
A100	1,112	1,238	1,246	1,088	1,262	1,220
	(71.5%)	(79.6%)	(80.1%)	(70.0%)	(81.2%)	(78.4%)

VIII. Summary and Conclusions

This paper described two implementations for a memory-bound sparse linear algebra kernel on the NVIDIA® Tesla V100 and A100 GPU architectures. To achieve optimal performance for such algorithms, it is critical to utilize the memory bandwidth effectively, which requires global memory requests to access consecutive memory locations. Results showed that restructuring the computation to support the desired memory access pattern improves performance. In addition, new features available on the A100 architecture were explored, including $L2$ residency control and asynchronous memory copies. The new implementation improves memory bandwidth performance on the A100 from 1104 GB/s to 1262 GB/s, or 81% of the peak memory bandwidth available.

Acknowledgments

This research was sponsored by the NASA Langley Research Center CIF/IRAD program, the NASA Transformational Tools and Technologies (TTT) Project of the Transformative Aeronautics Concepts Program under the Aeronautics Research Mission Directorate, and the National Institute of Aerospace Cooperative Agreement award NNL09AA00A. The authors would like to thank NVIDIA Corporation and Dr. Sameer Shende of the University of Oregon. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] R. Biedron, J.-R. Carlson, J. Derlaga, P. Gnoffo, D. Hammond, K. Jacobson, W. Jones, W. Kleb, E. Lee-Rausch, E. Nielsen, M. Park, C. Rumsey, J. Thomas, K. Thompson, A. Walden, L. Wang, and W. Wood, *FUN3D Manual 13.7*, NASA/TM-2020-5010139, 2020.

[2] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003.

[3] R. Li and Y. Saad, "GPU-Accelerated Preconditioned Iterative Linear Solvers." *Journal of Supercomputing*, p. 443–466, 2013.

[4] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid," in *ACM SIGGRAPH 2003 Papers*, ser. SIGGRAPH '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 917–924. [Online]. Available: https://doi.org/10.1145/1201775.882364

[5] Y. Liu and B. Schmidt, "LightSpMV: Faster CUDA-Compatible Sparse Matrix-Vector Multiplication Using Compressed Sparse Rows," *J. Signal Process. Syst.*, vol. 90, no. 1, p. 69–86, Jan. 2018. [Online]. Available: https://doi.org/10.1007/s11265-016-1216-4

[6] 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: Association for Computing Machinery, 2015, p. 99–108. [Online]. Available: https://doi.org/10.1145/2751205.2751244

[7] 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: https://www.sciencedirect.com/science/article/pii/S0167819113001178

[8] 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*, 2009, pp. 1–11.

[9] H. Anzt, W. Sawyer, S. Tomov, P. Luszczek, I. Yamazaki, and J. Dongarra, "Optimizing Krylov Subspace Solvers on Graphics Processing Units," in *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, 2014, pp. 941–949.

[10] 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, p. 231–242, Jan. 2011. [Online]. Available: https://doi.org/10.14778/1938545.1938548

[11] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms," in *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, 2007, pp. 1–12.

[12] NVIDIA. (2021) cuSPARSE User Guide. [Online]. Available: https://docs.nvidia.com/cuda/cusparse

[13] M. Naumov. (2021) Incomplete-LU and Cholesky Preconditioned Iterative Methods Using cuSPARSE and cuBLAS. [Online]. Available: https://docs.nvidia.com/cuda/incomplete-lu-cholesky

[14] A. Abdelfattah, H. Ltaief, and D. Keyes, "High Performance Multi-GPU SpMV for Multi-Component PDE-Based Applications," in *Euro-Par 2015: Parallel Processing*, J. L. Träff, S. Hunold, and F. Versaci, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 601–612.

[15] M. Zubair, E. Nielsen, J. Luitjens, and D. Hammond, "An Optimized Multicolor Point-Implicit Solver for Unstructured Grid Applications on Graphics Processing Units," in *Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms*. Piscataway, NJ, USA: IEEE Press, 2016, pp. 18–25.

[16] A. Walden, E. Nielsen, B. Diskin, and M. Zubair, "A Mixed Precision Multicolor Point-Implicit Solver for Unstructured Grids on GPUs," in *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, 2019, pp. 23–30.

[17] K. R. Laflin, S. M. Klausmeyer, T. Zickuhr, J. C. Vassberg, R. A. Wahls, J. H. Morrison, O. P. Brodersen, M. E. Rakowitz, E. N. Tinoco, and J.-L. Godard, "Data Summary from Second AIAA Computational Fluid Dynamics Drag Prediction Workshop," *AIAA Journal of Aircraft*, vol. 42, no. 5, pp. 1165 – 1178, 2005.

[18] NVIDIA. (2021) cuBLAS User Guide. [Online]. Available: https://developer.nvidia.com/cublas