Optimization of a Solver for Computational Materials and Structures Problems on NVIDIA Volta and AMD Instinct GPUs

Mohammad Zubair
Old Dominion University
Norfolk, Virginia

James Warner
NASA Langley Research Center (LaRC)
Hampton, Virginia

David Wagner
NASA Langley Research Center (LaRC)
Hampton, Virginia

***Abstract*—The Scalable Implementation of Finite Elements by NASA (ScIFEN) is a software package developed to solve complex computational materials and structures problems using the finite element method (FEM). In this paper, we describe optimization techniques to speed up the linear solver computation that occurs within the ScIFEN application. We consider GPUs from two different vendors, NVIDIA and AMD as our target platforms for optimization and highlight differences in performance and optimization techniques. The NVIDIA GPU Volta V100 is used in the Summit system deployed at Oak Ridge National Laboratory, and the new exascale system, Frontier, will be using AMD Radeon Instinct GPU. We evaluated the performance of various optimization techniques on test matrices, ranging in size from 100K to 4M, that are representative of ScIFEN applications. The linear solver computation is memory-bound on both GPUs. Our experiments show that on the NVIDIA GPU we obtained up to 79% of the theoretical peak bandwidth, while the AMD GPU achieved 59%. Overall, the NVIDIA V100 GPU outperforms the AMD MI 25 GPU[1]. We observed an overall speedup of up to 37X on an NVIDIA V100 compared to an Intel Skylake 12-core machine. The solver for a 4M degree of freedom system took under 2.5 seconds.**

***Index Terms*—Iterative Solver, Sparse Matrices, Unstructured Grid, High performance computing, Optimization, GPUs**

I. Introduction

The Scalable Implementation of Finite Elements by NASA (ScIFEN) is a software package developed to solve large-scale computational materials and structures problems using the finite element method (FEM) [1]. ScIFEN has target applications demanding the solution of a single large scale problem (e.g., for detailed microstructure models from X-ray CT scans), as well as efficient solution of thousands of moderate-sized analyses (e.g., for uncertainty quantification or structural design/optimization). In either case, the dominant computational bottleneck is the repeated solution of linear systems of equations. The linear solvers used in the ScIFEN software package are based on Krylov subspace methods [2], and is available in public domain as ScIFEN Solver Mini-App [3]. The performance and scalability of the linear solver are critical for the overall scalability of the ScIFEN code on emerging computer architectures.

In this paper, we describe an approach to port the ScIFEN application to GPUs. We also discuss an optimized implementation of the linear solver within ScIFEN. The ScIFEN application currently uses an iterative solver on CPUs from the Portable, Extensible Toolkit for Scientific Computation (PETSc) library [4]–[6] . Many researchers have studied efficient implementations of linear iterative solvers on GPUs [7], [8]. 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 [9]–[14]. To achieve improved performance, implementations have been developed using *cuSPARSE* library functions [15], [16]. An algorithm for high-performance block-sparse matrix-vector products for PDE-based applications on multiple GPUs has also recently been proposed [17].

The matrices arising in the ScIFEN solver are large, block-sparse, and unstructured, which leads to indirect memory addressing and low arithmetic intensity. The result is a memory-bound computation that makes it harder to optimize performance on most emerging computer architectures. The ScIFEN solver currently uses an iterative solver from the PETSc library that has been optimized for Intel CPUs. Based on our convergence studies, we found that the conjugate gradient and conjugate residual schemes work well for a class of ScIFEN applications.

In this work, we focus on a formulation of the conjugate residual method that can be implemented using BLAS and sparse BLAS routines. A sparse matrix-vector computation dominates the overall solver computation. For the initial GPU implementation on NVIDIA V100, we used the NVIDIA cuSparse library. Next, we developed an optimized implementation of the sparse matrix-vector operation which performs better than the NVIDIA library function. We also merged Level-1 BLAS routines to minimize the memory traffic.

In terms of hardware, we targeted the latest GPUs from two vendors: NVIDIA and AMD. The NVIDIA GPU Volta V100 is used in the Summit system deployed at Oak Ridge National Laboratory in 2018 [18]. The new exascale system, Frontier, to be deployed in 2021 at ORNL, will be using AMD

[1]Please note that AMD MI25 GPU is a lower-end model of AMD Instinct series based on the Vegas architecture, and for a fair comparison we should consider MI60 model. Unfortunately, we do not have access to MI60, but as our computation is memory-bound, the percentage of theoretical peak bandwidth metric is a good indicator of performance on the higher model.

Radeon Instinct GPU technology [19]. The Radeon Instinct accelerators are based on the Vegas architecture, and there are several GPU models available, for example, MI25, MI50, and MI60. The Frontier system is expected to use MI60. We selected NVIDIA V100 and AMD MI25 as target models for optimization. As demonstrated later, the architectures of these two GPUs, V100 and MI25, differ significantly and necessitate developing different optimization techniques.

We evaluated the performance of various optimization techniques on test matrices, ranging in size from 100K to 4M, that are representative of ScIFEN applications. We compared the performance on GPUs with the optimized PETSc library implementation on CPU. We observed an overall speedup of up to 37X on an NVIDIA V100 compared to an Intel Skylake 12-core machine. Here, our optimized solver took under 2.5 seconds for a 4M degree of freedom system. The MI25 is a lower-end model with a memory bandwidth of 484 GB/s, compared to 900 GB/s on the V100. The iterative solver computation is memory-bound on both GPUs. For comparison, we estimate what fraction of the theoretical bandwidth peak we can achieve on the GPUs. Our experiments show that on the NVIDIA GPU we obtained up to 79% of the theoretical peak bandwidth, while the AMD GPU achieved 59%. Since the linear solver kernel represents a large majority of the computation time for ScIFEN, the performance gains observed here can translate to substantial speedup for large scale computational materials and structures problems.

II. Background

The ScIFEN application was developed to efficiently solve large-scale nonlinear problems that arise in computational materials and structures applications. The most computationally intensive portion of the code, the repeated solution of a linear system

$$\mathbf{A}\Delta\mathbf{x} = \mathbf{b}, \tag{1}$$

is the focus of the optimization effort in this study. Though the total ScIFEN analysis time is dominated by the solution of the linear system, it will ultimately be necessary to port all of the computation to the GPU in order to realize the true computational benefits. This effort will be the subject of future work (Figure 1). This section provides a brief background describing the linear system of equations generated by ScIFEN and structure of the sparse matrix $\mathbf{A}$.

A. Linear System

The linear system being optimized in this study arises from the application of Newton's method [20] to a nonlinear system of equations of the form $\mathbf{F}(\mathbf{x}) = 0$. In the case of ScIFEN, $\mathbf{F}(\mathbf{x})$ represents nonlinear equilibrium equations in a three-dimensional solid domain under prescribed boundary conditions, where $\mathbf{F}$ is the imbalance between external and internal forces and $\mathbf{x}$ is a vector of point-wise displacements throughout the domain. It is assumed that the FEM has been used to discretize the governing partial differential equations and that the nonlinearity in the system is the result of history- or displacement-dependent material models.

Starting from an initial guess $\mathbf{x}^0$, Newton's method progresses from iteration k to $k+1$ as follows:

$$[\mathbf{J}(\mathbf{x}^k)]\Delta\mathbf{x}^{k+1} = -\mathbf{F}(\mathbf{x}^k) \tag{2}$$

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \Delta\mathbf{x}^{k+1} \tag{3}$$

where $[\mathbf{J}] = \frac{\partial \mathbf{F}}{\partial \mathbf{x}}$ is the Jacobian. These equations are iterated on until a convergence criterion is met, requiring the repeated solution of the linear system in Equation (2). Moving forward, the notation $\mathbf{A} = [\mathbf{J}(\mathbf{x}^k)] \in \mathbb{R}^{n\times n}$, $\Delta\mathbf{x} = \Delta\mathbf{x}^{k+1}$, and $\mathbf{b} = -\mathbf{F}(\mathbf{x}^k)$ is assumed for the linear system in Equation (1).

The primary responsibilities of the ScIFEN application is to assemble the linear system matrix, $\mathbf{A}$, based on specified material models and a FEM discretization, and to assemble the right hand side, $\mathbf{b}$, based on prescribed boundary conditions. ScIFEN then leverages the PETSc implementation of Newton's method to execute Equations (2) - (3). For more information, see [21] for details of the finite element formulation and PETSc [5] for the solution of the resulting nonlinear system of equations.

B. Matrix Storage

The storage scheme for the sparse matrix $\mathbf{A}$ influences the choice of algorithm and its implementation on the GPU architecture. The two common formats for matrices arising in FEM applications are compressed sparse row (CSR) and block CSR format.

1) CSR Format: In CSR format, a sparse matrix of size $n \times n$ with nnz number of non-zero entries is stored using three arrays. The array ia is an array of size $n+1$ whose i-th entry indicates the location of the leading non-zero entry in the i-th row of $\mathbf{A}$. The array includes a $n+1$ entry to facilitate easy traversal of the elements through the n-th row. The second array ja of size nnz provides the column index for each non-zero entry. A third array, $data$, of size nnz, is used to store the non-zero entries.

2) Block CSR Format : A block-sparse matrix $\mathbf{A}$ of size $n\times n$ with nnz number of non-zero blocks is stored in a block CSR format [22]. Note that n here is the number of $brow$/$bcol$ (block rows/block columns). In this format, like CSR format, two integer arrays ia and ja are used to efficiently capture the sparsity pattern of the matrix. The array ia is an array of size $n+1$ whose i-th entry indicates the location of the leading non-zero block index in the i-th $brow$ of $\mathbf{A}$. The array includes a fictitious $n+1$ entry to facilitate easy traversal of the elements through the n-th $brow$. The ja array is an array of size nnz that provides the $bcol$ index for each non-zero block. A third array, $\mathbf{A}$, of the form (nb, nb, nnz), is used to store the non-zero blocks. Each $nb \times nb$ block is stored in a column-major order.

Note that we are overloading the notation $\mathbf{A}$. We use $\mathbf{A}$ to refer to the sparse matrix and also to refer to an array holding the non-zero data values of the matrix. For keeping indexing simple and code more readable, we view $\mathbf{A}$ as a 3-d array, In the code, we use 1-d array $\mathbf{A}$ to hold the non-zero values.

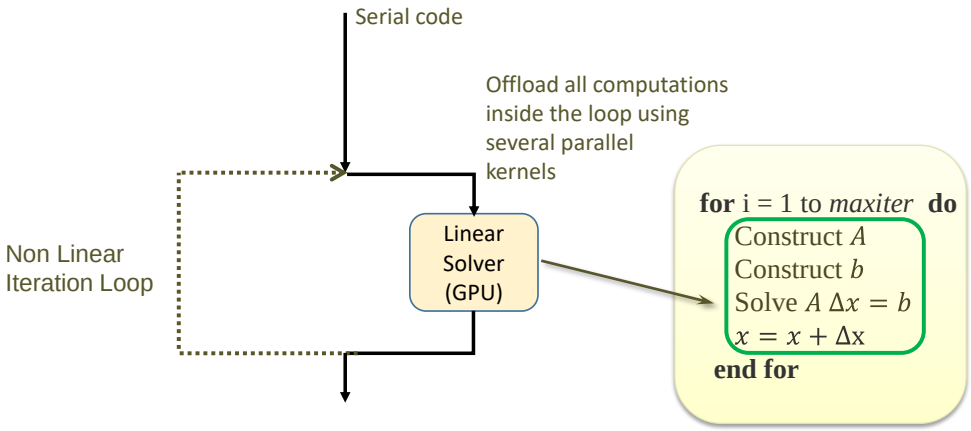

Fig. 1: Schematic of moving all computations in the nonlinear loop to GPU.

More specifically, an element $\mathbf{A}(i, j, k)$ is mapped to $\mathbf{A}[i + j * nb + k * nb * nb]$.

III. 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 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, where one or more blocks run on a streaming multiprocessor. The threads in a block are further partitioned into subgroups of threads known as warps.

The programming environments for the NVIDIA GPU and AMD GPU are quite similar. NVIDIA has developed CUDA, a C/C++ API for programming their GPUs [23]; and AMD has developed HIP, a C/C++ API to program their GPUs [24]. We target our implementation on NVIDIA V100 GPU and AMD MI 25. The significant differences in the two GPUs are: (a) the ratio between double precision and single precision on V100 is 0.5, and on MI 25 it is 0.0625, (b) the memory bandwidth on V100 is 900 GB/s and on MI 25 it is 484 GB/s, (c) the warp size is 32 on V100 and it is 64 on MI 25, and (d) atomic operations are not supported in the hardware on AMD MI25. Throughout our discussion, we assume readers are familiar with the CUDA programming environment and, in particular, programming with warps [25].

A. Approach

Our approach to implement and optimize a conjugate residual (CR) algorithm on GPUs is first to formulate the algorithm using BLAS and Sparse BLAS routines. Next, we focus on the performance of BLAS and Sparse BLAS routines and optimize them for the two GPUs. Finally, we merge a handful of Level-1 BLAS routines to improve the performance further.

Our starting CR algorithm is based on [26]. The BLAS and Sparse BLAS formulation of the preconditioned CR algorithm is shown in Algorithm 1.

Algorithm 1 Preconditioned CR($\mathbf{A}$, $\mathbf{M}$, $\mathbf{b}$, tol)

1: $\mathbf{d_x} = 0$
2: $\mathbf{r} = \mathbf{M} * \mathbf{b}$ (*custom GPU kernel*)
3: $\mathbf{a_r} = \mathbf{Ar}$ (*cusparseDcsrmv*)
4: $r_{old} = \mathbf{r^t a_r}$ (*cublasDdot*)
5: $bnorm = \sqrt{\mathbf{b^t b}}$ (*cublasDdot*)
6: $\mathbf{p} = \mathbf{r}$
7: $\mathbf{a_p} = \mathbf{Ap}$ (*cusparseDcsrmv*)
8: $r_2 = \mathbf{r^t r}$ (*cublasDdot*)
9: **while** $\sqrt{r_2} > tol * bnorm$ **do**
10: $\quad \mathbf{m_p} = \mathbf{M} * \mathbf{a_p}$ (*custom GPU kernel*)
11: $\quad denom = \mathbf{a_p^t m_p}$ (*cublasDdot*)
12: $\quad \alpha = r_{old}/denom$
13: $\quad \mathbf{d_x} = \mathbf{d_x} + \alpha \mathbf{p}$ (*cublasDaxpy*)
14: $\quad \mathbf{r} = \mathbf{r} - \alpha \mathbf{m_p}$ (*cublasDaxpy*)
15: $\quad \mathbf{a_r} = \mathbf{Ar}$ (*cusparseDcsrmv*)
16: $\quad r_{new} = \mathbf{r^t a_r}$ (*cublasDdot*)
17: $\quad \beta = r_{new}/r_{old}$
18: $\quad \mathbf{p} = \mathbf{r} + \beta \mathbf{p}$ (*cublasDscal, cublasDaxpy*)
19: $\quad \mathbf{a_p} = \mathbf{a_r} + \beta \mathbf{a_p}$ (*cublasDscal, cublasDaxpy*)
20: $\quad r_{old} = r_{new}$
21: $\quad r_2 = \mathbf{r^t r}$ (*cublasDdot*)
22: **end while**
23: **return** $\mathbf{d_x}$

B. Optimization

The sparse matrix storage provided in the ScIFEN Solver Mini-App [3] is in CSR format. The matrices used in ScIFEN applications have a dense block of size 3×3. We converted the matrix storage format from CSR to block CSR format as this results in less memory accesses and a better memory access pattern.

During our experimentation with NVIDIA BLAS and cuSparse library, we discovered that the sparse library for the matrix-vector operation could be further improved. For the AMD GPU, we did not find a sparse library to multiply a block

CSR matrix with a vector. As part of the first optimization, we developed an efficient implementation of a block CSR matrix with a vector operation for NVIDIA and AMD GPUs. Next, we merged Level-1 BLAS routines wherever possible and developed a custom implementation of them that minimizes the memory traffic. More specifically, the following modifications were made to Algorithm 1: (a) the vector multiplications (line 10) was merged with the *cublasDdot* (line 11), (b) the two *cublasDaxpy*s (line 13, line 14) were merged with the cublasDdot (line 21), and (c) the two *cublasDscal*s were merged with the two *cublasDaxpy*s (line 18, line 19).

In the following subsections, we first describe algorithms for block-sparse matrix-vector operation for NVIDIA and AMD GPUs. Next, we look at the efficient implementation of basic Level-1 BLAS that are used in developing efficient merged Level-1 BLAS routines.

1) Block-Sparse Matrix Vector Multiplication on NVIDIA GPU: We describe our algorithm for computing $\mathbf{y} = \mathbf{Ax}$, where $\mathbf{A}$ is a block-sparse matrix with a block size of 3. A single warp is assigned to multiply a block-row of the matrix $\mathbf{A}$ with the appropriate elements of $\mathbf{x}$, to compute 3 elements of $\mathbf{y}$. The warp processes three consecutive non-zero blocks of a row concurrently. As a warp on NVIDIA GPU consists of 32 threads, we have 7 left-over threads that are not used. The warp processes a row of the block-sparse matrix in a loop, where 3 consecutive non-zero blocks are processed by the warp at each iteration. The warp handles 27 $(3\times(3\times3))$ 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 $\mathbf{x}$ are also loaded from the read-only data cache, multiplied by the corresponding elements of $\mathbf{A}$, and the results are accumulated. After completion of the loop, the 27 partial results are aggregated into an output of 3 elements, which is stored on the device.

We now consider details for processing ith block row. For simplicity we assume, it consists of 12 non-zero blocks. These blocks are stored in the array $\mathbf{A}$ from index $istart$ to $iend$ where $istart = ia[i]$ and $iend = ia[i+1]$. More specifically, the first and last non-zero elements of block row i are $\mathbf{A}(0,0,istart)$ and $\mathbf{A}(2,2,iend)$. Note that an element $\mathbf{A}(i,j,k)$ is mapped to $\mathbf{A}[i+3j+9k]$. During the first iteration, a thread $tid \in \{1\ldots32\}$ of the warp works with an element of $\mathbf{A}$ from the first three non-zero blocks in block row i. Thread tid multiplies $\mathbf{A}(k,l,istart+nbki)$ with $\mathbf{x}(l,ja(istart+nbki))$, where

$$\begin{aligned} k &= \mathrm{mod}(tid,3) \\ l &= \mathrm{mod}(tid,9)/3 \\ nbki &= tid/9 \end{aligned}$$

During the second iteration, thread tid multiplies $\mathbf{A}(k,l,istart+3+nbki)$ with $\mathbf{x}(l,ja(istart+3+nbki))$. Once all non-zero blocks are processed, the 27 partial contributions are stored in the registers fk of the 27 threads. These 27 values can be viewed as 9 column vectors of size 3 each, which needs to be aggregated to generate a single column vector of 3 values. The aggregation (reduction) step is done using shuffle instructions as shown in Listing 1. Finally, the single column vector of 3 elements is stored in the device memory using the three threads of the warp.

```
1  tidx = threadIdx.x ;
2  k = threadIdx.x % 3;
3  l = (threadIdx.x % 9) / 3;
4  nbki = tidx/9;
5  nbk=3;
6  n =  blockIdx.x * blockDim.y + threadIdx.y;
7  istart = iam[n];
8  iend = iam[n + 1];
9  fk = 0.0;
10 //  loop over a block row while processing three non
       -zero blocks at a time
11 for (j = istart; j < iend-nbk+1; j+=nbk) {
12     fk += A(k, l, j+nbki) * x(l, jam[j+nbki]);
13 }
14 //  use shuffle to reduce 9 columns into a single
       column
15 f1 = fk;
16 f1 = f1 + __shfl_down_sync(0xfffffffe0, f1, 1 * 3);
17 f1 = f1 + __shfl_down_sync(0xfffffffe0, f1, 2 * 3);
18 f1 = f1 + __shfl_down_sync(0xfffffffe0, f1, 4 * 3);
19 f1 = f1 + __shfl_down_sync(0xfffffffe0, fk, 8 * 3);
20 if ((l == 0) && (nbki==0)) {
21     y(k, n) = f1;
22 }
```

Listing 1: Block-Sparse matrix vector multiplication

2) Block-Sparse Matrix Vector Multiplication on AMD GPU: For the AMD GPU, we have to account for warp size of 64. We consider two algorithms: (a) One block-row per warp as in the case of NVIDIA GPU, and (b) Two block-rows per warp. We now discuss details of the two algorithms.

One block-row per warp. In this algorithm, a warp processes seven consecutive non-zero blocks of a row concurrently. As a warp on AMD GPU consists of 64 threads, we have 1 left-over threads that is not used. The warp processes a row of the block-sparse matrix in a loop, where 7 consecutive non-zero blocks are processed by the warp at each iteration. The warp handles 63 $(7\times(3\times3))$ matrix entries during each iteration. The appropriate elements of $\mathbf{x}$ are also loaded from the read-only data cache, multiplied by the corresponding elements of $\mathbf{A}$, and the results are accumulated. After completion of the loop, the 63 partial results are aggregated into an output of 3 elements, which is stored on the device. The code segment to illustrate this computation is shown in Listing 2.

```
1  tidx = threadIdx.x ;
2  k = threadIdx.x % 3;
3  l = (threadIdx.x % 9) / 3;
4  nbki = tidx/9;
5  nbk=7;
6  // thread block is 64x4
7  n =  blockIdx.x * blockDim.y + threadIdx.y;
8  istart = iam[n];
9  iend = iam[n + 1];
10 fk = 0.0;
11 //  warp loop over a block row
12 for (j = istart; j < iend-nbk+1; j+=nbk) {
13     fk += A(k, l, j+nbki) * x(l, jam[j+nbki]);
14 }
15 //  shuffle to reduce 21 columns into a single
       column
16 f1 = fk;
```

```
f1 = f1 + __shfl_down(f1, 1 * 3, 64);
f1 = f1 + __shfl_down(f1, 2 * 3, 64);
ft = f1;
f1 = f1 + __shfl_down(f1, 4 * 3, 64);
f1 = f1 + __shfl_down(f1, 8 * 3, 64);
f1 = f1 + __shfl_down(ft, 16 * 3, 64);
f1 = f1 + __shfl_down(fk, 60, 64);
if ((l == 0) && (nbki==0)) {
    Y_OUT(k, n) = f1;
}
```

Listing 2: Block-Sparse matrix vector multiplication on AMD GPU

Two block-rows per warp. In this algorithm, we assign a warp to process two consecutive block-rows with the first set of 32 threads (half-warp) processing the first block-row and the second set of 32 threads processing the second block-row. A half-warp processes three consecutive non-zero blocks of a row concurrently. Note that in this algorithm, it is not necessary that the two consecutive block-rows have an identical number of non-zero blocks. Consequently, not all the 54 threads of a warp will be active all the time. The implementation of this algorithm is very similar to the GPU version shown in Listing 1.

3) Custom Dot Product using Atomics for NVIDIA and AMD GPUs: We adapt the parallel reduction technique suggested in [27] for the dot product. To keep our explanation simple, we assume the two input arrays, $in1$ and $in2$, for the dot product are very long and their sizes are a multiple of 1024×512. We partition each array into partitions of 1024×512 sizes each. We further divide a partition into 1024 blocks, where a block consists of 512 elements, see Figure 2. We first use grid-stride loop using thread blocks of 512 threads to multiply the two arrays and perform partial reduction to create a single partition. The grid-stride loop computes reduction per-thread in a register [28]. Figure 2 illustrates this phase of computation.

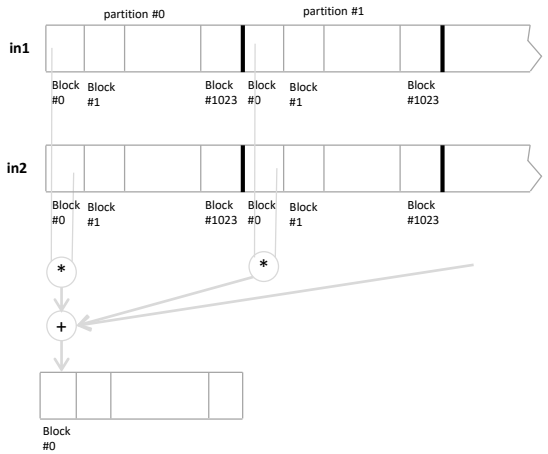

Fig. 2: Grid-stride loop to multiply the two arrays and perform partial reduction to create a single partition.

Next, we reduce 512 elements in a block by warps in two stages. In the first stage, we use eight warps to reduce 512 elements into eight elements. A warp uses shuffle instruction for reduction. In the second stage, one warp reduces the eight elements into a single result, which is then used to update a global variable that holds the final result. The update to the global variable is performed using an atomic update. Note that there is a maximum of 1024 elements that needs to be aggregated using atomic updates. Figure 3 illustrates this part of the computation.

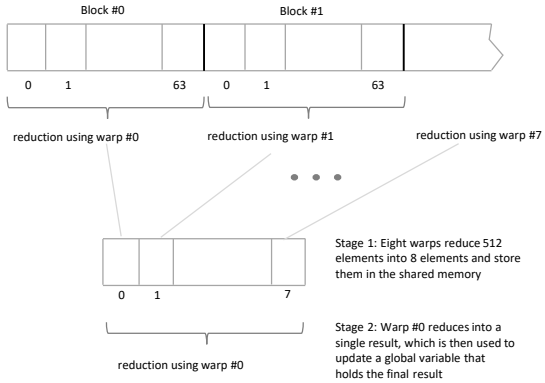

Fig. 3: Two-stage block reduction.

```
void custom_dot(double *in1, double *in2,
  double *dout, int N) {
  double *d_r2;
  cudaMalloc((void **)&d_r2,  sizeof(double));
  cudaMemsetAsync(d_r2,0,sizeof(double));
  int threads = 512;
  int blocks = min((N + threads - 1) / threads,
    1024);
  blockReduceAtomic<<<blocks, threads>>>(in1, in2,
    d_r2, N);
  cudaMemcpy(dout, d_r2, sizeof(double),
    cudaMemcpyDeviceToHost);
}
```

Listing 3: Custom Dot Product

```
__global__ void blockReduceAtomic(double *in1,
    double *in2, double *out, int N) {
  double sum = 0.0;
  for(int i = blockIdx.x * blockDim.x + threadIdx.x;
      i < N;
      i += blockDim.x * gridDim.x) {
    sum += in1[i]*in2[i];
  }
  sum = blockReduceSum(sum);
  if (threadIdx.x == 0)
    atomicAdd(out, sum);
}
```

Listing 4: Atomic Block Reduce

4) Custom Dot Product using two Kernels and without Atomics: The implementation of the dot product on AMD GPU using atomics is very similar to the NVIDIA GPU implementation. However, the implementation needs to be adjusted for a warp size of 64, as shown in Figure 4. The code segment for the custom dot product is shown in Listing 3. The two-stage block reduction is shown in Listing 4, and the supporting

functions for NVIDIA GPU, warp size of 32, are shown in Listing 5 and Listing 6.

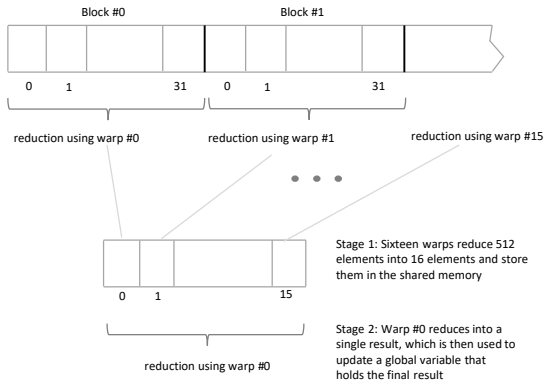

Fig. 4: Two-stage block reduction for AMD GPU.

```
__inline__ __device__
  double blockReduceSum(double val) {
  // shared mem to store partial reductions
  static __shared__ double shared[16];
  int lane = threadIdx.x % warpSize;
  int wid = threadIdx.x / warpSize;
  // Each warp computes partial reduction
  val = warpReduceSum(val);
  // Write reduced value to shared memory
  if (lane==0) shared[wid]=val;
  __syncthreads();
  //read only needed data from shared memory
  val = (threadIdx.x < blockDim.x /
         warpSize) ? shared[lane] : 0;
  //Final reduction using first warp
  if (wid==0) val = warpReduceSum(val);
  return val;
}
```

Listing 5: Two-Stage Block Reduction

```
__inline__ __device__
double warpReduceSum(double val) {
  val += __shfl_down(val,16,32);
  val += __shfl_down(val,8,32);
  val += __shfl_down(val,4,32);
  val += __shfl_down(val,2,32);
  val += __shfl_down(val,1,32);
  return val;
}
```

Listing 6: Warp Reduction using Shuffle

5) Merge Routine: Vector times Vector with Dot Product: We merge the two functions at Line 10 and Line 11 of the Algorithm 1. For this, we modify the function, *blockReduceAtomic()*, of the custom dot product kernel, discussed earlier, see Listing 4. Observe that while performing the vector times vector operation, we have values in the registers that need to be multiplied for computing the dot product in the next step (Line 11 of Algorithm 1). The code is modified to reuse the values in the register for the dot product operation to reduce the memory traffic, see Listing 7.

```
__global__ void blockReduceAtomicModified(double* mi
    , double* ap,
                double* miap, double *out, int N) {
  double sum = 0.0;
  double tmp0, tmp1;
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  for(int i = idx ;
      i < N;
      i += blockDim.x * gridDim.x) {
      tmp0 = ap[i];
      tmp1 = mi[i]*tmp0;
      miap[i] = tmp1;
      sum += tmp1*tmp0;;
  }
  sum = blockReduceSum(sum);
  if (threadIdx.x == 0)
    atomicAdd(out, sum);
}
```

Listing 7: Merge Routine: Vector times Vector with Dot Product

6) Merge Routine: Two DAXPY with Dot Product: We merge the three functions at Line 13, Line 14, and Line 21 of Algorithm 1. The goal here is to reuse the value of r that is brought in to the register for the DAXPY operation. The reuse avoids going back to the memory while performing the dot vector operation. This requires merging of DAXPY operation with the *blockReduceAtomic()*, see Listing 8. The reason for doing two DAXPY together is to utilize the core pipeline better.

```
__global__ void dual_daxpy_kernel(double* xk, double
    * p,
    double* r, double *miap, double alpha, double *
    out, int N) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  double sum=0.0;
  for(int i = idx ;
      i < N;
      i += blockDim.x * gridDim.x) {
        xk[i] = xk[i] + alpha * p[i];
        r[i] = r[i] - alpha * miap[i];
        sum += r[i]*r[i];
  }
  sum = blockReduceSum(sum);
  if (threadIdx.x == 0)
    atomicAdd(out, sum);

}
```

Listing 8: Merge Routine: Two DAXPY with Dot Product

7) Merge Routine: Two DAXPY with Two Dscal: In this merging, we reduce the memory by combining the DAXPY and scaling operation, see Listing 9.

```
__global__ void scale_daxpy_kernel(double* p, double
    * ap,
    double* r, double *ar, double beta, int N) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  for(int i = idx ;
      i < N;
      i += blockDim.x * gridDim.x) {
      p[i] = r[i] + beta*p[i];
      ap[i] = ar[i] + beta*ap[i];
  }
}
```

Listing 9: Merge Routine: Two DAXPY with Two Dscal

8) Single Precision Implementation: Using single precision to store the non-zero values of the sparse matrix can improve performance by reducing the memory traffic and speeding up the execution of arithmetic operations. However, the use of lower precision can result in accuracy and convergence issues. To improve performance further, we developed a single-precision implementation for the iterative solver.

IV. Results

We evaluated the performance results of the proposed algorithms for a testbed of sparse matrices arising in the ScIFEN application. These matrices range from size 100K to 4M with non-zeros ranging from $8M$ to $350M$, see Figure 5.

MATRIX	N	Total Non-Zeros	Avg Non-Zeros Per Row
100K	100704	8201934	81
250K	251733	20732877	82
500K	511167	42448329	83
1M	1070391	89131761	83
2M	2046246	170856450	83
4M	4185654	350095572	84

Fig. 5: Test Matrices.

The ScIFEN Solver Mini-App [3] currently uses an iterative solver from the PETSc library that has been optimized for Intel CPUs. For the baseline performance, we use the PETSc library solver on a 12-cores Intel Skylake 3104 CPU. The baseline performance results are summarized in Figure 6. Observe that for bigger matrices the time increases linearly with the number of non-zeros.

We implemented different algorithms discussed in Section III on NVIDIA GPU Volta V100 and AMD Instinct GPU MI 25. All implementations use the same tolerance and result in almost identical preconditioned residual. The speedup, relative to Skylake 12-cores, of various implementations, are summarized in Figure 7. Observe that the performance of our custom V100 implementation outperforms the one based on NVIDIA library. On average, 80% of the iterative solver time is spent on the sparse matrix-vector operation. We compute the percentage of theoretical peak-bandwidth for the optimized double-precision implementations on V100 and MI 25 (see Figure 8). For the AMD GPU, two block-rows per warp algorithm performs around 5% better compare to one block-row per warp algorithm. We suspect it is due to the additional complexity in aggregation of partial results for the one block-row per warp algorithm. The bandwidth results in Figure 8 for the AMD GPU are for the two block-rows per warp algorithm.

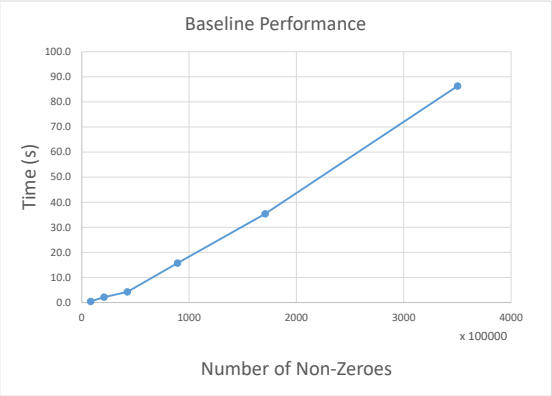

Fig. 6: Test Matrices.

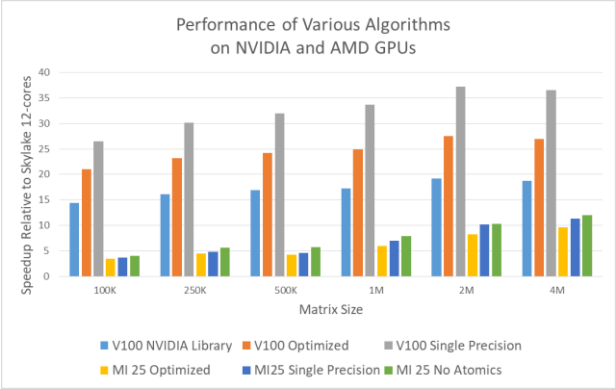

Fig. 7: Performance Comparison on NVIDIA V100 and AMD MI 25.

V. Summary and Future Work

In this paper, we proposed different optimization techniques to speed up the ScIFEN mini-app on two different GPUs: NVIDIA V100 and AMD MI25. The GPU accelerator is an integral part of a node of future exascale systems. For example, the Frontier exascale system that will be deployed in 2021 at Oak Ridge National Laboratory, has AMD Instinct GPU as the accelerator. We demonstrated that the optimized solver outperforms the solver that is realized using vendor-supplied libraries. We evaluated the performance of various optimization techniques on test matrices ranging from 100K to 4M arising in the ScIFEN application. We observed an overall speedup of up to 37X on NVIDIA V100 compared to Intel Skylake 12-cores machine. The solver for a 4M degree of freedom system took under 2.5 seconds.

Acknowledgments

We are thankful to the High Performance Computing Incubator (HPCI) program at NASA Langley Research Center for

Matrix	V100 Double Precision Bandwidth (GB/s)	Percentage of Theoretical Peak Bandwidth (V100)	MI 25 Double Precision Bandwidth (GB/s)	Percentage of Theoretical Peak Bandwidth (MI 25)
100K	605	67%	208	43%
250K	663	74%	249	51%
500K	688	76%	224	46%
1M	698	78%	236	49%
2M	712	79%	265	55%
4M	713	79%	284	59%

Fig. 8: Percentage of Peak Bandwidth for Sparse Matrix Vector Multiplication.

providing support for this work.

REFERENCES

[1] J. E. Warner, G. B. Bomarito, G. Heber, and J. D. Hochhalter, "Scalable implementation of finite elements by NASA - implicit (ScIFEi)," NASA Langley Research Center, Tech. Rep. NASA/TM-2016-219180, 2016.

[2] Y. Saad, "Krylov subspace methods for solving large unsymmetric linear systems," *Mathematics of Computation*, vol. 37, no. 155, pp. 105–126, 1981.

[3] J. E. Warner and D. Wagner, "Scifen solver mini-app (hpci)," https://software.nasa.gov/software/LAR-19417-1, 2019.

[4] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, D. Karpeyev, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, and H. Zhang, "PETSc Web page," https://www.mcs.anl.gov/petsc, 2019. [Online]. Available: https://www.mcs.anl.gov/petsc

[5] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, D. Karpeyev, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, and H. Zhang, "PETSc users manual," Argonne National Laboratory, Tech. Rep. ANL-95/11 - Revision 3.11, 2019. [Online]. Available: https://www.mcs.anl.gov/petsc

[6] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient management of parallelism in object oriented numerical software libraries," in *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds. Birkhäuser Press, 1997, pp. 163–202.

[7] 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

[8] 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

[9] 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.

[10] 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

[11] 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

[12] 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

[13] 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

[14] 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

[15] NVIDIA-II, "cuSPARSE User Guide," http://docs.nvidia.com/cuda/cusparse/index.html#axzz4Ggg2gNCX, 2015.

[16] 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.

[17] A. Abdelfattah, H. Ltaief, and D. Keyes, *High Performance Multi-GPU SpMV for Multi-component PDE-Based Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 601–612.

[18] ORNL-I, "Summit," https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/, 2019.

[19] ORNL-II, "U.s. department of energy and cray to deliver record-setting frontier supercomputer at ornl."

[20] P. Deuflhard, *Newton Methods for Nonlinear Problems: Affine Invariance and Adaptive Algorithms*. Springer Publishing Company, Incorporated, 2011.

[21] T. Belytschko, W. Liu, and B. Moran, *Nonlinear Finite Elements for Continua and Structures*. Wiley, 2000.

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

[23] NVIDIA-III, "CUDA C Programming Guide," http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz4Hicq83a9, 2015.

[24] AMD-I, "Hip programming guide," https://rocm-documentation.readthedocs.io/en/latest/Programming_Guides/HIP-GUIDE.html, 2019.

[25] Y. Lin and V. Grover, "Using cuda warp-level primitives," https://devblogs.nvidia.com/using-cuda-warp-level-primitives/, 2018.

[26] G. H. Golub and C. F. Van Loan, *Matrix Computations (3rd Ed.)*. Baltimore, MD, USA: Johns Hopkins University Press, 1996.

[27] J. Luitjens, "Faster parallel reductions on kepler," https://devblogs.nvidia.com/faster-parallel-reductions-kepler/, 2014.

[28] M. Harris, "Cuda pro tip: Write flexible kernels with grid-stride loops," https://devblogs.nvidia.com/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/, 2013.