Hybrid Implementation of a Multicolor Point-Implicit Linear Solver on CPU and GPU

Mohammed Zubair with Seyedmeysam Abolghasemi and Jason Orender, Old Dominion University, Norfolk, VA USA, zubair@cs.odu.edu

***Abstract*—Navier-Strokes equations, which are broadly used in the field of computational fluid dynamics (CFD), are often solved using an unstructured grid approach to enable geometric complexity. Implicit solution methodologies for such spatial discretization techniques generally require frequent solutions for large tightly-coupled systems of block-sparse linear equations [1]. The Multicolor Point-Implicit solver, developed at NASA Langley Research Center, is one of the most commonly used linear solvers to solve Navier-Stokes equations. Computing solutions to these linear systems accounts for a significant fraction of the overall runtime of the simulation. This paper focuses on a hybrid implementation of a multicolor point-implicit linear solver on both Graphical Processing Units (GPUs) and Central Processing Units (CPUs) to leverage all computational resources of a single machine to optimize the linear solver.**

I. Introduction

The point solver mini-app presented in this paper is a functional copy of the solver used in the large scale CFD application "Fun3D" at the NASA Langley Research Center; the application is utilized to solve Navier-Stokes (NS) equations using an unstructured grid approach to represent the geometric complexity. This mini-app takes advantage of an implicit time-integrated solution for solving a large tightly-coupled system of block-sparse linear equations which must be solved at each physical time step. Multicolor point-implicit relaxation is employed in the linear solver used in the mini-app to solve for those linear systems. Despite the existing implementation being quite efficient on a CPU, the task of computing solutions for these linear systems takes a large fraction of the overall runtime of the application. New studies have been made to port these computationally-intensive solvers to GPUs in order to optimize the performance[1], However, a hybrid implementation in which all the computational resources are involved in solving the linear solver has heretofore not been accomplished.

II. Background

A. The State of the Art

Past researchers have looked at efficient implementation of iterative solvers on GPUs before [2, 3]. Most of these solvers have been for general sparse matrices with a block size of one and the focus has been on optimizing the underlying SpMV (Sparse Matrix-Vector multiplication) kernel, which has also been explored separately by many researchers [4, 5, 6, 7, 8, 9]. In addition to these baseline optimizations, for a solver in a CFD application, both sparse block-matrix vector multiplication operations and the forward and backward computations need to be optimized. For performance reasons, there has been an effort to implement solvers using BLAS (Basic Linear Algebra Subprograms) from the cusparse library [10]. Again, the matrices considered here are sparse matrices with a block size of one and the main sparse BLAS used is "cusparseDcsrmv". All matrices encountered in the CFD simulations are sparse block matrices.

B. Problem Definition

The result of the mini-app solver's implicit solution approach is a set of linear equations of the form $Ax = b$ that must be solved frequently during the simulation; A represents a spatial mesh containing n grid points. This mesh is a matrix of size $n \times n$ in which each entry is itself a $n_b \times n_b$ matrix that is the result of linearization of nonlinear equations at each grid point. The matrix A is divided into diagonal D and off-diagonal O matrices:

$$A = D + O \tag{1}$$

The mini-app initializes the grid points by renumbering them with the reverse Cuthill–McKee algorithm (RCM) [11] to permute the sparse matrix into a band matrix [12]. Figure 1 shows an example of applying this renumbering technique to form an off-diagonal matrix of a fixed size.

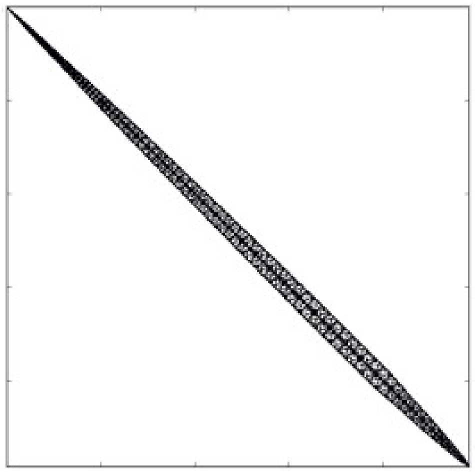

Fig. 1. Off-diagonal matrix (O) structure after applying RCM renumbering.

An array of size $[n \times n_b \times n_b]$ is used to store n blocks of the diagonal matrix D. For each block D_i, two triangular sub-matrices, the lower L_i and the upper U_i, are generated in-place before running each linear solver for $1 \leq i \leq n$. The L_i and the U_i matrices are then computed using Forward and back substitution algorithm [13]. This is another useful technique used to help improve cache locality.

The off-diagonal matrix O contains *nnz* non-zero blocks, where each block is stored using a modified compressed sparse row (CSR) [14] format. In the modified CSR format, three arrays are used: *ia* and *ja*, to efficiently capture the sparsity pattern of the matrix and a third array of format $[n \times n_b \times n_b]$, to store all of the non-zero elements. The integer array *ia* of size $(n + 1)$ is used to keep indexes of all leading non-zero blocks in each row of O. The *ja* array of size *nnz* stores the column indexes of all non-zero blocks. Figures 2 and 3 demonstrate an example of a blocksparse matrix with $n_b = 2$ and the resulting arrays after applying CSR techniques on them.

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

(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. Subfigure (a) shows the sparsity structure of a matrix O. An entry "×" indicates a non-zero block. Subfigure (b) shows 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.

The multi-coloring scheme used in the point-implicit linear solver exposes the parallelism in the solver computation. It groups colors and grid points such that no two neighbor points are colored the same. All unknowns associated with a grid point are assigned the color of that point.

In this context a "color" is a grouping of grid points that are not expected to influence the calculations on any other grid points in a cohort if they are assigned identical colors; all grid points that are assigned "red", to extend the analogy, are expected to be computationally independent of each other, while the unknowns associated with "red" points might well have an impact on any given "green" or "black" points.

An approximate nearest-neighbor flux Jacobian is used to generate A which results in no data dependencies between the unknowns of the same color; this provides the possibility of updating them in parallel fashion. The process is repeated several times over the entire system, and each time the unknowns are updated with the latest values of x from the other colors. To improve memory access and consequently cache performance, the system of algebraic equations is renumbered so that the unknowns of the same color are grouped together by organizing them consecutively and the arrays of *ia* and *ja* are modified to adopt the new matrix structure. Once the linear solver computation is done, an inverse map is then used to update the nonlinear solution of the partial differential equations (PDEs) at each grid point.

An example of a grid mesh with n = 968,633 would generate eleven colors. Figure 4 demonstrates the resulting off-diagonal matrix that is produced after applying RCM renumbering and reordering to row color. The number of rows in each color decreases as the color index increases with the first color index having 166800 rows compared with the last color index containing only 11 rows.

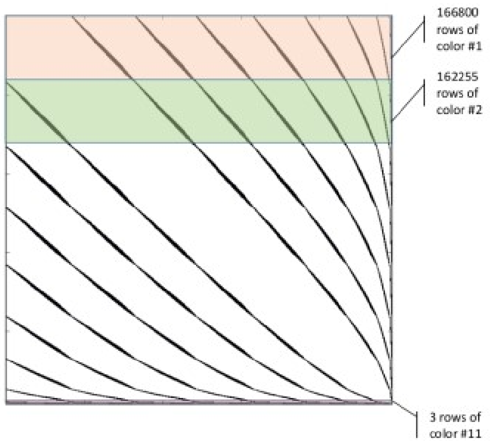

Fig. 4. Off-diagonal matrix structure after RCM renumbering and reordering based on row colors. Only three colors are highlighted for clarity.

III. Algorithm Implementation

The general pseudocode for the main program follows:

```
[transfer data to device]
for i = 1 to sweeps
  for j = 1 to num_colors
    solve_subroutine_1 (CPU or GPU)
    solve_subroutine_2 (GPU)
    solve_subroutine_3 (GPU)
    solve_subroutine_4 (GPU)
[transfer data from device]
```

The general pseudocode for the solve subroutine follows:

```
Start solve_subroutine:

set f1, f2, f3, f4, f5 to residual
     array elements for this node.
start = ia[node]
end   = ia[node+1]-1
for i = start to end
  icol = ja[i]
  decrement f1..f5 by the product of
     the off-diagonal matrix values and
     the previous solution matrix
     values five times (over each
     column, if the f1..f5 variables
     are viewed as rows*)

  // solve forward
  decrement f2..f5 by the product of
     the diagonal matrix values and f1
  decrement f3..f5 by the product of
     the diagonal matrix values and f2
  decrement f4..f5 by the product of
     the diagonal matrix values and f3
  decrement f5 by the product of the
     diagonal matrix value and f4

  // solve backward
  decrement f1..f4 by the product of
     the diagonal matrix values and a
     factor of f5
  decrement f1..f3 by the product of
     the diagonal matrix values and a
     factor of f4
  decrement f1..f2 by the product of
     the diagonal matrix values and a
     factor of f3
  decrement f1 by the product of the
     diagonal matrix value and a factor
     of f4

  set the new solution values to f1..f5

End solve_subroutine

* Refer to figure 2, but instead of 2x2
  block sizes, 5x5 block sizes are
  used.
```

All calculations in this implementation will take place on a single node. The data set was split into four segments and the combined result is compared with a result from a single-threaded sequential Fortran version known to be correct.

For the hybrid implementation, three of the four segments of the dataset were processed on the GPU, while one was processed on the CPU. Even with this uneven distribution, the three GPU segments were able to finish before the CPU segment.

For the GPU only implementation, the code was kept virtually identical with the exception that all four segments were processed on the GPU.

IV. Results

Three Scenarios were tested:

1. Hybrid Implementation.
2. GPU Only Implementation.
3. CPU Only Implementation.

The code was kept nearly identical between the three cases except for the particular parts of the code that were executed. In the hybrid case, both the solve-on-GPU and solve-on-CPU functions were used, while in the other two cases only one or the other was used.

TABLE I. Results overview

Impl	Time Completion Statistics		
	Mean (s)	*Sample Stdev*	*GPU time (ms)*
Hybrid	1.217	0.0233	405.3
GPU	0.728	0.0454	551.8
CPU	3.913	0.0720	

Fig. 5. Results statistics calculated over 20 trial runs for each implementation. Note that the completion time listed for the hybrid model is both the time required for all solve subroutines (CPU and GPU) and the time required for only the CPU solve subroutine. This is because both the CPU and GPU solve subroutines are running in parallel and the CPU solve subroutine takes longer (it is the "longest pole in the tent", so to speak).

As expected, the amount of GPU time increased approximately linearly as the number of data set segments being calculated on the GPU went from 3 (hybrid) to 4 (GPU only). An exact linear extrapolation would have been 540.4 ms, which compares favorably with the 551.8 ms actually observed. Since the amount of calculation required on each data set segment is only approximately equal, this is a reasonable result and affirms that the amount of calculation was in fact evenly spread over each of the four data set segments.

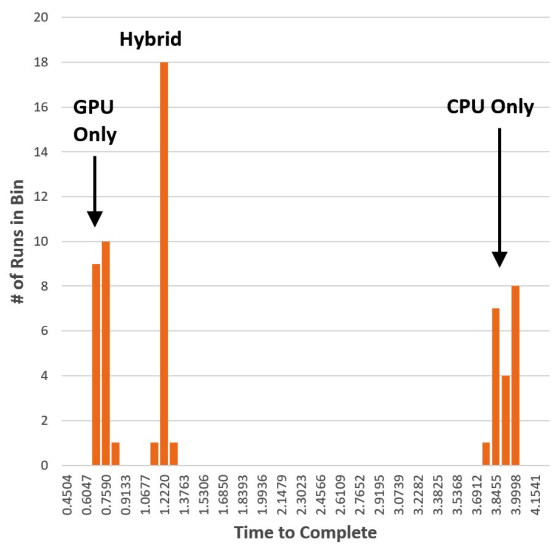

Fig. 6. Graphical reqpresentation run times. There are 50 bins, and each bin is 0.077 seconds wide. The number of runs that fell into each bin is on the y-axis and the center of the bin is demarcated on the x-axis. Shorter run times are better.

Unexpectedly, the hybrid implementation took significantly longer to execute than the GPU-only implementation, even though it was only processing one third of the data that the GPU was processing. In order for the hybrid implementation to take less time, based on these results, the data set would have to be split into at least six segments. To get this estimate, simply divide the average time required for the entire calculation on the CPU-only with the time required for the entire calculation on the GPU-only, and then round up. At or near this level of division, the long pole in the tent with respect to the hybrid implementation will become the GPU processing. This sort of load balancing would be unique to each GPU/CPU combination.

V. Conclusions

A load balancing evaluation would be needed for each GPU/CPU combination in order to determine the optimal split between the data processed on the GPU platform and the data processed on the CPU platform. In this case the optimal split would be six ways – one partition being processed on the CPU and five being processed on the GPU (see figure 7). Due to this, the relatively little benefit that would result from this effort, and the prospect of having to transfer data back and forth between sweeps for more complex implementations on multiple nodes, it would seem that a hybrid implementation is not feasible for this application at this time.

TABLE II. Estimated Optimal Results

Impl	**Estimated Time Completion**		
	Time (s)	***Difference (s)***	***GPU time (ms)***
Hybrid	0.652	- 0.564	459.2
GPU	~0.728	(+)	~551.8
CPU	~3.913	(–)	■

Fig. 7. Estimated optimal results are based on extrapolating numbers from actual results using four partitions and do not take into account shifts in overhead that might occur by processing greater amounts of data on the GPU. The (-) notation indicates that this number is expected to be lesser by some small amount, while the (+) notation indicates that the number is expected to be greater by some small amount.

VI. Future Work

As new hardware architectures become available, the act of load balancing may in fact yield benefits that are out of proportion to the costs of implementing. At that time, this work should be revisited.

References

[1] M. Zubair, E. Nielsen, J. Luitjens, and D. Hammond. An optimized multicolor point-implicit solver for unstructured grid applications on graphics processing units. In 2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3), pages 18–25, Nov 2016.

[2] R. Li and Y. Saad. Gpu-accelerated preconditioned iterative linear solvers. Journal of Supercomputing, 63(2):443–466, 2 2013.

[3] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder. Sparse matrix solvers on the gpu: Conjugate gradients and multigrid. ACM Trans. Graph., 22(3):917–924, July 2003.

[4] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo. Sparse matrix-vector multiplication on gpgpus. ACM Trans. Math. Softw., 43(4):30:1–30:49, Jan. 2017.

[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, ICS '15, pages 99–108, New York, NY, USA, 2015. ACM.

[6] H.-V. Dang and B. Schmidt. Cuda-enabled sparse matrix-vector multiplication on gpus using atomic operations. Parallel Comput., 39(11):737–750, Nov. 2013.

[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, SC '09, pages 18:1–18:11, New York, NY, USA, 2009. ACM

[8] X. Yang, S. Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on gpus: Implications for graph mining. CoRR, abs/1103.2405, 2011.

[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, SC '07, pages 38:1–38:12, New York, NY, USA, 2007. ACM

[10] M. Naumov. Incomplete-lu and cholesky preconditioned iterative methods using cusparse and cublas. 2011

[11] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In Proceedings of the 1969 24th National Conference, ACM '69, pages 157–172, New York, NY, USA, 1969. ACM.

[12] Cuthill–McKee algorithm. Cuthill–mckee algorithm — Wikipedia, the free encyclopedia, 2017. [Online; accessed 30-November-2017].

[13] Triangular matrix. Triangular matrix — Wikipedia, the free encyclopedia, 2017. [Online; accessed 03-December-2017].

[14] Y. Saad. Iterative Methods for Sparse Linear Systems. Society for Industrial and Applied Mathematics, second edition, 2003.