

A Hierarchical Jacobi Iteration for Structured Matrices on GPUs using Shared Memory

Mohammad Shafaet Islam

Department of Aeronautics and Astronautics
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Email: moislam@mit.edu

Qiqi Wang

Department of Aeronautics and Astronautics
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Email: qiqi@mit.edu

Abstract—This paper presents an algorithm to accelerate the Jacobi iteration for solving linear systems of equations arising from structured problems on graphics processing units (GPUs). Acceleration is achieved by utilization of on-chip GPU shared memory via a domain decomposition procedure. In particular, the problem domain is partitioned into subdomains whose data is copied to the shared memory of each GPU block. Jacobi iterations are performed internally within each block's shared memory while avoiding expensive global memory accesses every iteration, resulting in a hierarchical algorithm (which takes advantage of the GPU memory hierarchy). We investigate the algorithm performance on the linear systems arising from the discretization of Poisson's equation in 1D and 2D, and observe an 8x speedup in convergence in the 1D problem and a nearly 6x speedup in 2D compared to a conventional GPU implementation of Jacobi iteration which only relies on global memory.

I. INTRODUCTION

High fidelity scientific simulations typically require solving large linear systems of equations which result from the discretization of a partial differential equation (PDE). The solution of such linear systems often takes a vast amount of computational time to complete. Solving these linear systems efficiently requires the use of massively parallel hardware with high computational throughput, as well as the development of algorithms which respect the memory hierarchy of these hardware architectures to achieve high memory bandwidth.

Graphics processing units (GPUs) have emerged as a popular hardware architecture for scientific simulation [1], [2]. There has been great interest in the study and development of linear solver algorithms for GPUs, due to their high computing power and low bandwidth relative to CPUs [3], [4]. The Jacobi iteration is a linear solver algorithm which is exceptionally well suited to implementation on GPUs due to its highly parallel nature and remarkable simplicity. As a result, a number of studies have been conducted and show promising speedup. For example, the Jacobi iteration has demonstrated speed up on GPUs for sparse [5] (up to 100x) and dense [6] test matrices, relative to counterpart CPU implementations. Speedups have also been demonstrated in scientific applications such as cardiac modeling [7]. A particularly relevant study is performed by Cecilia et. al who investigate the performance benefits of utilizing on-chip GPU shared memory in their Jacobi iteration solver on the 1D and 2D Laplace equation. They

observe a three to four times speedup for various domain sizes compared to their unoptimized CUDA implementation (and even higher speedups relative to their CPU implementation) [8]. Other authors develop asynchronous implementations of Jacobi which remove the need for complete synchronization at each iteration, demonstrating roughly 2x speedup relative to synchronous approaches on the GPU [9], [10]. These studies demonstrate that the performance of Jacobi iteration can be improved through the use of GPUs, with further improvements available by utilizing on-chip shared memory and performing iterations in an asynchronous manner.

The goal of this paper is to develop and investigate the performance of a Jacobi iterative solver for GPUs which respects the GPU memory hierarchy. In particular, we develop an algorithm which utilizes on-chip shared memory while applying iterations in an asynchronous or hierarchical fashion, using the ideas in previous studies of the method on GPUs. We investigate the performance of this algorithm compared to a traditional approach which only uses global memory.

This paper is organized as follows. Section II provides background for this work; a review of the Jacobi iterative method for solving linear systems of equations and an overview of the CUDA framework for GPU computing. Section III introduces a shared memory algorithm for performing Jacobi iteration on 1D problems, and provides performance comparisons relative to a traditional GPU implementation which uses global memory. Section IV presents an analogous algorithm and comparison results in the context of 2D problems. Section V provides concluding remarks for this work.

II. BACKGROUND

A. Jacobi iteration

The Jacobi iteration is an iterative method which can be used to solve a linear system of equations $Ax = b$, $A \in \mathbb{R}^{n \times n}$, $x \in \mathbb{R}^n$, $b \in \mathbb{R}^n$. This is done by progressively applying the Jacobi update equation (1) (represented in its elemental form).

$$x_i^{(n+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=i} a_{ij} x_j^{(n)} \right) \quad (1)$$

In (1), $x_i^{(n)}$ and $x_i^{(n+1)}$ denote the solution for the i th degree of freedom (DOF) after n and $n + 1$ iterations, while a_{ij} and

b_i denote the corresponding matrix and right hand side entries. Convergence of Jacobi iteration is guaranteed for diagonally dominant matrices.

B. Shared Memory on GPUs

The CUDA programming model is typically used in order to program NVIDIA GPUs [11]. The most fundamental unit of parallel execution is the thread. A large number of threads can run simultaneously. This gives GPUs high computing power and the ability to perform mathematical operations very efficiently. Groups of threads are organized into GPU blocks (each with a distinct ID denoted by `blockIdx.x`). The number of threads within a GPU block is denoted by `blockDim.x` in CUDA and can be specified by the user. Additionally, each block contains its own on-chip shared memory storage which is accessible to the threads comprising the block [12]. The amount of shared memory in a block is usually limited (approximately 48 kB of shared memory per block) and accessible only to threads within a block. However, shared memory on the GPU has a higher bandwidth and lower latency compared to GPU global memory which is accessible by all threads (approximately 10 times higher bandwidth [11] and 100 times lower latency [13]). For this reason, adapting GPU applications to use shared memory has become a popular optimization technique for improving performance.

III. A SHARED MEMORY JACOBI SOLVER FOR 1D PROBLEMS

We develop a Jacobi iterative solver which utilizes GPU shared memory for one dimensional problems. As a model problem, we consider the one dimensional Poisson equation on a 1D domain $x \in [0, 1]$ with Dirichlet boundary conditions as represented by Equation (2):

$$-\frac{d^2u}{dx^2} = f, \quad u(0) = u(1) = 0 \quad (2)$$

A finite difference discretization of Equation (2) on a one dimensional grid with a second order central difference scheme leads to the tridiagonal linear system of equations $Ax = b$ where A is given by (where Δx denotes the grid spacing)

$$A = \frac{1}{\Delta x^2} \begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix}$$

Applying the Jacobi iteration (Equation (1)) to advance the i th DOF value (denoted by x_i) in our tridiagonal system from iteration n to $n + 1$ results in the following update equation (where b_i represents the right hand side value at the i th DOF)

$$x_i^{(n+1)} = \frac{1}{2} \left(b_i(\Delta x)^2 + x_{i-1}^{(n)} + x_{i+1}^{(n)} \right) \quad (3)$$

Equation (3) shows that the solution at the next iteration at a particular DOF depends only on its neighbor values.

One way to utilize shared memory in our application is to partition the solution data between the available shared

memory and operate on these partitions independently. We consider a one dimensional grid which can be partitioned into several subdomains as shown in Figure 1. The DOFs in each subdomain may be updated by a distinct GPU block. Figure 1 shows an example of a 12 point grid, divided into 4 point subdomains which will be operated on by 3 GPU blocks.

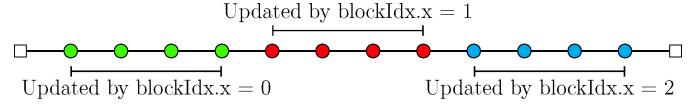


Fig. 1: 1D domain partitioned between different GPU blocks. Each GPU block is responsible for updating its own set of DOFs.

The first step in our approach is to copy the DOF values in each subdomain from global memory (where the initial solution resides) to the shared memory of the GPU block which will be responsible for the update. Updating the edge values in each subdomain requires information from the adjacent DOFs which will not be accessible to the block. We address this by augmenting the subdomains by an extra DOF to the left and right (beyond the DOFs that will be updated). These augmented subdomains (shown in Figure 2 for our example) can be copied to the shared memory of each GPU block. The shared memory simply holds a temporary copy of the solution which is to be operated on, while the final solution is always stored in global memory.

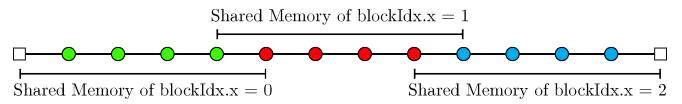


Fig. 2: Data allocated from global memory (containing the entire grid) to the shared memory of each GPU block. Each block receives data corresponding to DOFs it is responsible for updating plus an additional left and right neighbor.

After the copy to shared memory is complete, we can perform the Jacobi iteration on the interior DOFs of each subdomain using the stencil update given by Equation (3). To maximize parallelism, each DOF update should be handled by a distinct GPU thread. For our example, the number of threads per block should be set to 4 so that 4 threads can be used to update the 4 interior points within each subdomain. In general, the threads per block specification is exactly equal to the number of interior points in a subdomain (i.e. the subdomain size is directly tied to the user specified thread per block choice). Given a thread per block specification of `blockDim.x`, the subdomain size will be `blockDim.x + 2` with `blockDim.x` interior points to be updated. To execute the previous global to shared memory transfer efficiently, each of the `blockDim.x` threads should copy one value from global to shared memory and the remaining two values can be copied by threads 0 and 1. One can alter the subdomain size by changing the thread per block specification. For efficiency, the threads per block specification should be a multiple of 32 because threads are launched in groups of 32 threads (known as warps). This ensures all

threads in a warp are active and avoids thread divergence. As a consequence, subdomains with a multiple of 32 interior DOFs are preferable.

Since data movement (rather than computation) tends to be the bottleneck in many scientific computing applications such as iterative linear solvers, we recommend performing many Jacobi updates within shared memory before executing expensive data transfers back to global memory. We refer to the Jacobi iterations performed within shared memory as subiterations, and denote the number of subiterations performed by k . Upon performing the k subiterations, the updated values can be copied from the shared memory of every GPU block to their global memory position. Only the interior `blockDim.x` values of each subdomain must be copied back from shared to global memory since the two edgemost points are not updated. Each thread should copy the same DOF it updated from shared to global memory. Shared memory is terminated at the end of this step and the updated solution now exists in global memory.

This completes one cycle of our shared memory algorithm. The algorithm is implemented in a single GPU kernel (since shared memory is only active as long as a GPU kernel is running). The main parameters in the algorithm are the number of threads per block (`blockDim.x`) and the number of subiterations (k). The number of threads per block controls the size of the subdomains in shared memory, while the number of subiterations controls the number of Jacobi iterations performed within shared memory before communication back to global memory. Our algorithm is depicted in Figure 3. In summary, each cycle of the algorithm involves the following three steps:

- 1) Partition the gridpoints between the different GPU blocks and copy values from global memory to shared memory. Each subdomain has size `blockDim.x` + 2.
- 2) Update the interior `blockDim.x` points in each GPU block by performing k Jacobi iterations (we refer to these as subiterations).
- 3) Update the global memory solution array by copying the interior `blockDim.x` values from shared memory to their appropriate positions in global memory (shared memory is terminated at this point).

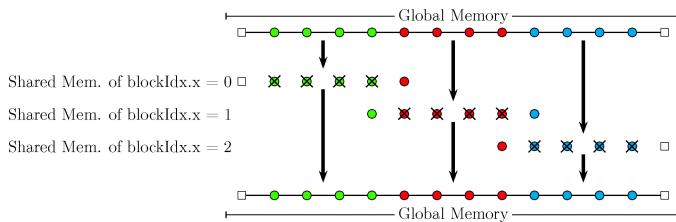


Fig. 3: Schematic of the shared memory algorithm for 1D iteration. Initially, data is copied from global memory to the shared memory of different GPU blocks. Each GPU block is responsible for a distinct set of DOFs. After performing k Jacobi iterative updates (marked by X) the updated DOFs are transferred back to global memory.

A GPU kernel that uses shared memory requires the user to specify the number of bytes of shared memory required. In our

implementation, we reserve enough shared memory for twice the length of the local solution data (as Jacobi iteration requires two solution containers) as well as the local right hand side. Therefore, the total amount of shared memory required for the algorithm is 2 times the subdomain size (`blockDim.x` + 2) for the solution array plus an array of size `blockDim.x` for the right hand side values corresponding to the interior DOFs. This should be multiplied by the size of the representation (4 bytes for oats, or 8 bytes for doubles).

One practical question to consider when applying the algorithm is what is a good number of subiterations to perform within shared memory. Using a large value of k is beneficial as we can perform many Jacobi updates without requiring expensive global memory accesses. The drawback of setting a large k is the lack of frequent update of the boundary values of each subdomain. As a result, convergence can be negatively impacted as subiterations used to update interior DOFs within a subdomain will rely on old boundary data. Setting $k = 1$ ensures that subdomains have the most up to date boundary values every iteration, and is exactly equivalent to performing Jacobi iteration. However, setting such a low k value requires many expensive global memory accesses which will inhibit algorithm performance. There is a tradeoff between receiving up to date subdomain values more frequently to improve convergence (by setting a small k), and performing fewer global memory accesses to improve performance (by setting a large k). We explore good values for k in our numerical tests.

A. Effect of Subiterations

We investigate the performance of the shared memory Jacobi solver on the model problem (2). We compare the performance to a classical global memory implementation where each GPU thread consistently updates one DOF (without the use of shared memory via a domain decomposition approach). We consider a problem size of $N = 1024$ interior DOFs (1026 points in total) and measure the time required to reduce the L_2 residual norm of the initial solution (set to a vector of ones) by a factor of 10^{-4} using Jacobi iteration, where the right hand side vector b has all entries set to one.

Because this problem is too small to fully utilize the streaming multiprocessors of the GPU, 1024 copies of the problem are operated on simultaneously. This ensures that the GPU is fully utilized and that our algorithm can scale to higher dimensional problems (i.e. an analogous 2D problem on a 1024 by 1024 grid). The time required for the classic GPU global memory and shared memory approaches to converge is measured. All numerical tests were run using a TITAN V GPU with double precision.

The thread per block specification is varied between the values of [32, 64, 128, 256, 512] for the classic GPU approach, and the best time is recorded. For the shared memory implementation, the thread per block value of 32 is used. This corresponds to a required shared memory allocation of 800 bytes per block (much lower than the shared memory limit). Although larger thread per block values could be used in 1D (which from our experience result in better performance), we

wish to extend our findings to higher dimensions in which case the available threads would have to be split between different dimensions. The number of blocks is always set to accommodate all of the DOFs in the problem (i.e. ceiling of the total number of DOFs from all 1024 copies of the problem divided by the thread per block value). The following 6 subiteration values are explored in this study: $k = 4, 8, 16, 32, 64, 128$.

Figure 4 shows the performance of the shared memory approach for the various subiteration values specified above compared to the best classic GPU performance. The performance results include the time required for host to device and device to host transfers, and were recorded using the CUDA Event API. For the thread per block values of [32, 64, 128, 256, 512], the classic GPU times measured were [7248.11, 5690.35, 5675.69, 5677.27, 5680.42] ms. The optimal time measured is 5675.69 ms when the thread per block value is 128, although the time is essentially constant for thread per block values of 64 and above. The shared memory approach outperforms the classic GPU approach in all cases except when the number of subiterations is set to the highest value of 128. The best performance is obtained when $k = 8$. Past $k = 16$, the time to converge increases linearly with the number of subiterations, suggesting that performing too many subiterations within shared memory can hurt performance.

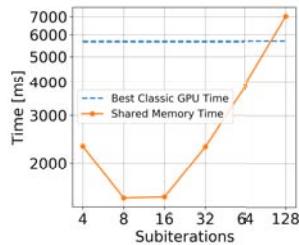


Fig. 4: Comparison of time required for classic GPU global memory and shared memory implementations with different subiteration values to reduce the residual of the solution by 10^{-4} for a problem size of $N = 1024$ (solving 1024 copies simultaneously). The shared memory approach outperforms the classic approach in all cases except when $k = 128$, and performs the best when $k = 8$.

Table I shows the speedup achieved going from the best classic GPU performance to the shared memory performance for each subiteration value specified. The best speedup achieved is nearly four times the classic GPU approach when $k = 8$ (setting $k = 16$ results in a nearly similar speedup). The results demonstrate that the time for convergence of Jacobi iteration can be reduced using shared memory.

Number of Subiterations	4	8	16	32	64	128
GPU to Shared Speedup	2.45	3.82	3.79	2.47	1.46	0.81

TABLE I: Speedup from classic GPU to shared memory approach for various subiteration values. A nearly four times speedup is achieved when $k = 8$.

B. Effect of Overlapping Subdomains in 1D

One key feature of our shared memory algorithm is that many Jacobi updates are performed each cycle without trans-

fer to global memory. While this minimizes global memory accesses to improve performance, convergence is hampered by the lack of up to date values used for the updates. In particular, boundary data in each subdomain is only updated every cycle (after k subiterations). Therefore, DOFs close to the subdomain edges may contribute a larger residual value to the overall residual norm. Improving the solution at DOFs close to the edges can greatly improve the convergence of the algorithm and reduce the number of cycles necessary.

We attempt to alleviate this problem by using overlapping subdomains. The approach is inspired by overlapping domain decomposition methods, where the convergence rate is improved as more overlap is introduced between subdomains [14]. The key idea of this approach is that DOFs which are further away from subdomain edges (closer to the subdomain center) have less error than those closer to the edges. With overlapping subdomains, we permit sets of points close to the subdomain edges to be allocated to two subdomains (copied to the shared memory of two GPU blocks). A DOF which exists in two subdomains will have a different local position in each subdomain. The subdomain in which the DOF is further from the edge is the one which will contribute the final value to global memory.

To illustrate this approach more clearly, an example is shown in Figure 3 for the 12 point grid. In this example, each subdomain has two interior DOFs in common with its neighboring subdomains. These overlapping DOFs are updated in the shared memory of two GPU blocks. Either block could copy its DOF value back to global memory at the end of the cycle. However, we choose to have the left subdomain provide the final left DOF value and the right subdomain provide the final right DOF value in each set of overlapping points. This is because the local position of the left DOF within the left subdomain is further from the edge while compared to its local position in the right subdomain (a similar argument applies for the right DOF). Figure 5 shows how the GPU blocks contribute to the overall global memory solution in this approach. In general, any even number of overlapping DOFs is permitted (as long as it is less than the number of interior points in the subdomain) and in this case the left subdomain should contribute the left half of DOFs while the right subdomain should contribute the right half.

Numerical tests are performed to study the effect of overlap on the performance of the shared memory algorithm. In these tests, a thread per block value of 32 is used as before. Each subdomain can have an even number of interior overlapping points with its neighboring subdomain (as long as the number of overlapping points is less than the number of interior points in a subdomain). For our thread per block specification of 32, we permit up to 30 interior overlapping points between subdomains. The left subdomain will contribute the left half of the overlapping points while the right subdomain contributes the right half. Figures 6a and 6b show the time and number of cycles required for the reduction in residual by a factor of 10^{-4} as a function of overlap for subiteration values of $k = 4, 8, 16, 32, 64, 128$.

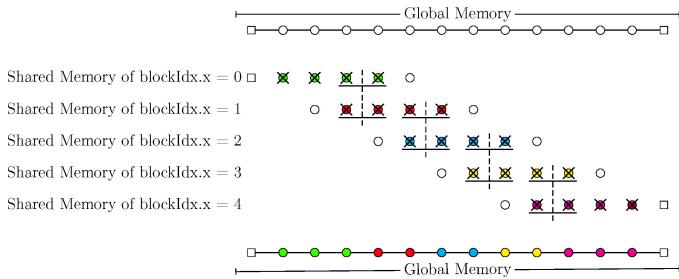


Fig. 5: Schematic of the shared memory algorithm with overlapping subdomains and the associated transfer of data between global and shared memory. In this example, two interior points overlap between each subdomain. This approach improves the global memory solution of interior DOFs which reside near subdomain edges.

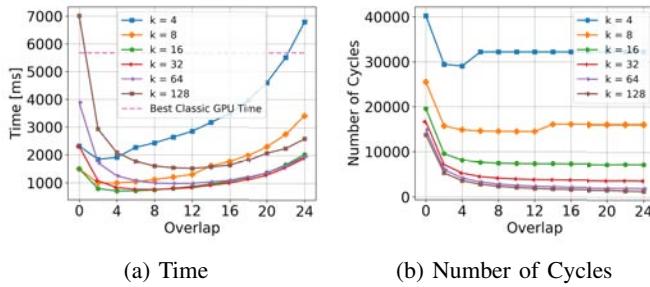


Fig. 6: Time and Number of Cycles required to reduce the residual as a function of overlap for different subiterations k . For a given k , performing overlap initially decreases the number of cycles required for convergence (causing the time to decrease as well). The number of cycles eventually saturates as overlap increases further. However, the time increases due to the extra computational resources required. The best performance is achieved with $k = 16$.

For all subiteration cases, allowing just two points to overlap causes a drop in the number of cycles required for convergence. This is a result of improving poor DOF values adjacent to subdomain edges. The reduction in cycles leads to an initial reduction in time, particularly for larger values of k . As the amount of overlap increases further the number of cycles required for convergence continues to decrease slowly and eventually saturates. The time; however, grows dramatically as overlap is increased further. This is because the amount of computation required per cycle grows with overlap. Given a problem size with N interior DOFs, threads per block tpb and overlap o , the number of operational blocks required is:

$$\text{Operational blocks} = \frac{N - tpb}{tpb - o} + 1 \quad (4)$$

This is because the first block has a subdomain size of $tpb + 2$, and every subsequent block added handles $tpb - o$ unique points in our grid of $N + 2$ points (including boundary points). The corresponding number of total GPU threads required is the number of blocks times the threads per block value tpb . Equation (4) shows that the computational resources required on the GPU increases with overlap, so it is only beneficial to perform overlap as long as it sufficiently reduces the number of cycles required for convergence.

For each value of k tested, Table II shows the optimal overlap for minimum time and corresponding speedup of the shared memory algorithm relative to the classic GPU implementation. The results illustrate that the optimal number of overlapping points increases as the number of subiterations is increased. This is expected because increasing k causes more DOFs near the subdomain edges to have poor values due to lack of up to date boundary values (so more overlap is necessary to improve these DOFs). The best performance is achieved when setting $k = 16$ with 4 interior DOFs overlapping between subdomains. This corresponds to an eight times speedup compared to the classic GPU performance. The best performance was previously achieved using $k = 8$, but now higher k values yield the best speedup because poor edge values (associated with high k) can be improved with overlap.

The numerical test results demonstrate that the convergence of Jacobi iteration for 1D problems can be accelerated using shared memory. Although the initial results suggested a four times speedup for this problem (resulting from tuning the number of subiterations), introducing overlap to improve edge DOF values results in an eightfold speedup. The optimal setting for the number of subiterations is on the order of the number of interior DOFs in a subdomain. Furthermore, a small amount of overlap is enough to decrease the number of cycles dramatically.

Number of Subiterations	4	8	16	32	64	128
Overlap	2	4	4	8	10	12
Speedup	3.09	5.74	8.00	7.50	5.84	3.75

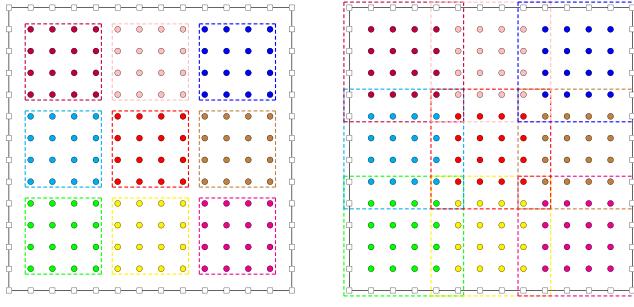
TABLE II: Speedup from classic GPU to shared memory approach for various subiteration values using overlapping subdomains. The optimal overlap increases with the number of subiterations performed. An eight times speedup is achieved when $k = 16$ and 4 interior points overlap between each subdomain.

IV. A SHARED MEMORY JACOBI SOLVER FOR 2D PROBLEMS

We extend the shared memory algorithm presented in Section III to the two-dimensional case. The domain decomposition of a sample two-dimensional domain into smaller subdomains is illustrated in Figure 7a. In this example, a 2D domain with 12 by 12 interior points is partitioned into smaller 4 by 4 subdomains which would be updated by 9 GPU blocks (each represented by a different color).

The analogous shared memory algorithm in 2D involves the following three steps, where the main parameters are the number of threads per block in the x -direction (blockDim.x) and y -direction (blockDim.y), and the number of subiterations performed within shared memory (k):

- 1) Partition the gridpoints between the different GPU blocks and copy values from global memory to shared memory. Each subdomain has size $\text{blockDim.x} + 2$ by $\text{blockDim.y} + 2$.
- 2) Update the interior blockDim.x by blockDim.y points in each GPU block by performing k Jacobi iterations.



(a) 2D Domain and Subdomains (b) 2D Augmented Subdomains

Fig. 7: Illustration of 2D Domain which is partitioned into several subdomains. The augmented subdomains (shown on the right with dashed lines identifying each subdomain) are copied from global memory to the shared memory of different GPU blocks. Each GPU block will perform updates for one subdomain.

- 3) Update the global memory solution array by copying the interior `blockDim.x` by `blockDim.y` values from shared memory to their appropriate positions in global memory.

We study the performance of the shared memory algorithm in solving the pentadiagonal system arising from discretization of the 2D Poisson equation in a square domain $x \in [0, 1] \times y \in [0, 1]$, with Dirichlet boundary conditions. We consider a domain containing 1024×1024 interior grid points and measure the time to reduce the L_2 residual norm of the initial solution by a factor of 10^{-4} . This problem is analogous to the previous 1D problem in terms of computational complexity, and requires full utilization of the GPU. The entries of the initial solution and the right hand side vector are set to one. A TITAN V GPU is used for the numerical tests, which are performed using double precision.

The number of threads per block in the x direction is set to 32. For the classic GPU approach, the threads per block value in the y direction is varied between 4, 8, 16, 32 and the best time is recorded. For the shared memory approach, the threads per block value in the y direction is also set to 32 so that the problem domain is split into subdomains with 32 by 32 interior points. The amount of shared memory required using these specifications is 26.688 kB per block (less than the maximum allowable value of 48 kB). The number of subiterations is set to $k = 4, 8, 16, 32, 64, 128$ as before. We also permit overlapping subdomains in this study. In this case, a subdomain can have up to 30 interior points overlap with its neighboring subdomains in each direction. We restrict the overlap in the x and y directions to be the same value. Figures 8a and 8b show the time and number of cycles required for the shared memory approach to converge as the overlap in both directions is increased.

Figure 8b shows that an initial amount of overlap can significantly drop the number of cycles required for convergence, especially when the number of subiterations is large. The corresponding time also decreases, except in the case when

$k = 4$, as the decrease in the number of cycles is not enough to outweigh the extra work incurred due to overlap. As the number of overlapping points increases further, the number of cycles saturates. However, the time continues to increase because a higher overlap value corresponds to a larger GPU resource requirement and more computational time per cycle.

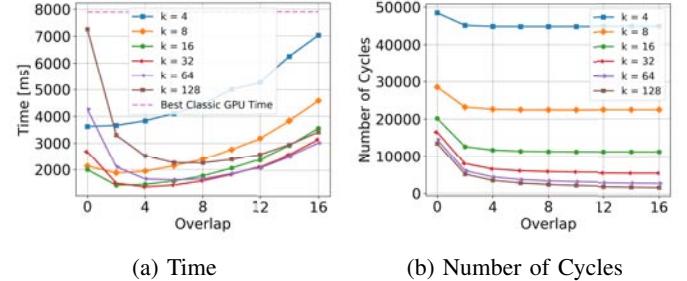


Fig. 8: Time and Number of Cycles required for convergence as a function of overlap for different number of subiterations k for the 2D problem. The behavior is similar to the 1D case. The best performance is achieved with $k = 16$.

Table III shows the optimal overlap value and corresponding speedup of the shared memory algorithm relative to the classic GPU implementation for the 2D problem, for each of the subiteration values used. As before, the optimal overlap increases with the number of subiterations. A nearly six times speedup is achieved using $k = 32$ with 4 overlapping DOFs in the x and y directions. The numerical test results demonstrate

Number of Subiterations	4	8	16	32	64	128
Overlap in x and y	0	2	2	4	6	8
Speedup	2.18	4.22	5.58	5.84	4.88	3.50

TABLE III: Speedup from classic GPU to shared memory approach for various subiteration values used in the 2D problem, with overlapping subdomains. The best speedup (nearly sixfold) is achieved when $k = 32$ with 4 overlapping points between subdomains in both directions.

that the convergence of Jacobi iteration for 2D problems can also be accelerated using shared memory by tuning the number of subiterations and overlap. In general, the optimal number of subiterations is on the order of the number of interior DOFs in a subdomain along a single dimension. Furthermore, introducing a small amount of overlap in both directions was enough to decrease the number of cycles dramatically.

V. CONCLUSION

In this work, we have developed and investigated a hierarchical Jacobi solver for 1D and 2D structured problems that utilizes shared memory to improve performance. The shared memory approach demonstrates a 6–8 times speedup relative to a classical GPU implementation which relies solely on global memory. The algorithm can also be used to accelerate other scientific computing applications which involve memory bound stencil computations (e.g. the explicit time-stepping of PDEs) on GPUs, and demonstrates the benefit of adapting classical algorithms to emerging hardware to enable both high computational throughput as well as efficient memory access.

REFERENCES

- [1] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.
- [2] R. Fernando, *GPU Gems*, 1st ed. Boston, MA: Addison-Wesley, 2004.
- [3] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid," *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 917–924, 7 2003.
- [4] R. Li and Y. Saad, "GPU-accelerated preconditioned iterative linear solvers," *The Journal of Supercomputing*, 2012.
- [5] T. Wang, Y. Yao, L. Han, D. Zhang, and Y. Zhang, "Implementation of Jacobi iterative method on graphics processor unit," in *2009 IEEE International Conference on Intelligent Computing and Intelligent Systems*, vol. 3, 2009, pp. 324–327.
- [6] Z. Zhang, Q. Miao, and Y. Wang, "CUDA-Based Jacobi's Iterative Method," in *2009 International Forum on Computer Science-Technology and Applications*, vol. 1, 2009, pp. 259–262.
- [7] R. Amorim, G. Haase, M. Liebmann, and R. W. dos Santos, "Comparing CUDA and OpenGL implementations for a Jacobi iteration," in *2009 International Conference on High Performance Computing Simulation*, 2009, pp. 22–32.
- [8] J. M. Cecilia, J. M. Garc'a, and M. Ujald'on, "CUDA 2D Stencil Computations for the Jacobi Method," in *Applied Parallel and Scientific Computing*, K. Jónasson, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 173–183.
- [9] S. Venkatasubramanian and R. W. Vuduc, "Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems," 01 2009, pp. 244–255.
- [10] H. Anzt, S. Tomov, J. Dongarra, and V. Heuveline, "A block-asynchronous relaxation method for graphics processing units," *Journal of Parallel and Distributed Computing*, vol. 73, no. 12, pp. 1613–1626, 2013.
- [11] N. Wilt, *The CUDA Handbook: A comprehensive Guide to GPU Programming*, 1st ed. Addison-Wesley Professional, 2013.
- [12] D. B. Kirk and W. mei W. Hwu, *Programming Massively Parallel Processors, Third Edition: A Hands-on Approach*, 3rd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016.
- [13] M. Harris, "Using Shared Memory in CUDA C/C++," January 2013. [Online]. Available: <https://devblogs.nvidia.com/using-shared-memory-cuda-cc/>
- [14] V. Dolean, P. Jolivet, and F. Nataf, *An Introduction to Domain Decomposition Methods: Algorithms, Theory, and Parallel Implementation*. SIAM, 2015.