A GPU Memory System Comparison for an Elliptic Test Problem

Yu Wang
UMBC
wang.yu@umbc.edu

Marc Olano
UMBC
olano@umbc.edu

Matthias K. Gobbert
UMBC
gobbert@umbc.edu

Wesley Griffin
UMBC
griffin5@cs.umbc.edu

ABSTRACT

This paper presents GPU-based solutions to the Poisson equation with homogeneous Dirichlet boundary conditions in two spatial dimensions. This problem has well-understood behavior, but similar computation to many more complex real-world problems. We analyze the GPU performance using three types of memory access in the CUDA memory model (direct access to global memory, texture access, and shared memory). Based on data locality, different CUDA algorithms are designed to accommodate the different device memory performance behaviors. We present a performance study on the speedup of our GPU-based solutions on an NVIDIA Tesla C2070 over serial code. By relating the data access pattern and its spatial locality, our results show that an algorithm using global memory with coalesced reads outperforms the other memory systems and allows effective solvers using single precision floating points.

1. INTRODUCTION

The Poisson equation with homogeneous Dirichlet boundary conditions on a unit square domain in two dimensions can be numerically approximated with the finite difference method, but also has a well known analytical solution, allowing direct computation of the convergence and accuracy of iterative algorithms. The finite difference form of this elliptic differential equation test problem has a sparse, highly structured system of linear equations. For high resolution meshes, storing the resulting linear systems can be challenging. Matrix-free methods use the assumed structure of the problem to avoid explicitly storing the system matrix. The conjugate gradient method invoked in our solver is iterative yet matrix-free, and is representative of memory-bound Krylov subspace methods. In this paper, we present three single-GPU versions of the algorithm, with kernels tuned to three alternate strategies for GPU memory access.

2. RELATED WORK

Even most already parallel algorithms need some adaptation to run effectively on parallel GPU architectures. Some research optimizes common parallel programming primitives on the GPU. Blelloch introduced the scan primitives for parallel array operations [3, 4]. Sengupta et al. [22] implemented the segmented scan primitives proposed by Iverson [12] on the GPU using CUDA. He et al. [9] improved the bandwidth of both gather and scatter operations by improving memory locality in data access.

The Basic Linear Algebra Subprogram (BLAS) library provides a wide range of highly optimized implementation for linear algebra operations between vectors (BLAS1), between matrix and vector (BLAS2) and between matrices (BLAS3). It has a CUDA accelerated implementation, cuBLAS library [19], and most of the BLAS functionalities are available on CUDA-enabled GPUs. Besides BLAS and cuBLAS libraries, other linear algebra libraries include MAGMA [23], which is a high performance dense linear algebra library similar to the LAPACK [2], but handles heterogeneous and hybrid architectures. Improvements have been made to many of the GPU linear algebra algorithms, especially the BLAS2 and BLAS3 functions [17, 24, 16].

Although CUDA provides the programmers with access to GPGPU instruction set, some researchers have been working on creating high-level languages on top of CUDA or building new libraries of GPU data structures to reduce the complexity for coding parallel GPU code. Lefohn et al. [15] proposed a library that hides the complexity of designing the data structures from GPU programmers. Larson et al. [14] proposed an applicative array language, Barracuda, for GPUs, which is embedded in Haskell [11]. Barracuda is a high-level programming language with optimization for CUDA code while hiding the complexity of hand-writing generic CUDA code. The new language has similar performance to the cuBLAS library in the cases of dot product and linear operation between vectors, but Barracuda is currently limited to optimization of array operations, with more effort need to be made to make it a practical new language.

We use the Conjugate Gradient method for solving a symmetric positive definite linear system with a large, sparse, and structured system matrix as example in this paper. The linear system arises from a finite difference discretization of the Poisson equation, as shown in detail in the next sec-

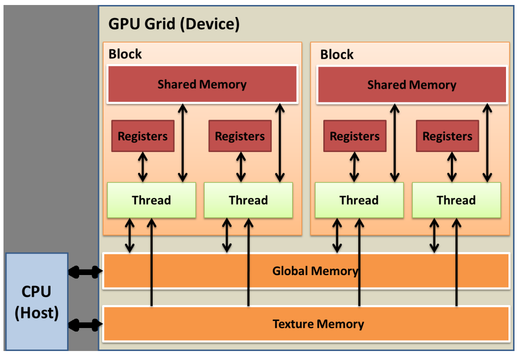

Figure 1: CUDA Memory Model

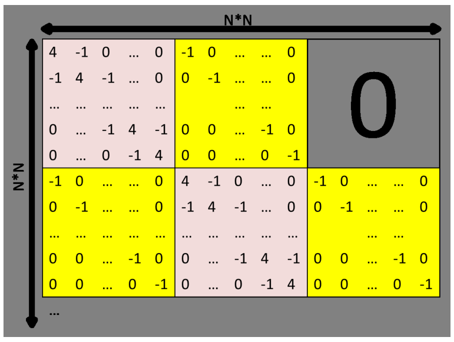

Figure 2: Structure of matrix A, yellow blocks are T and pink blocks are S

tion. Bolz et al. [5] showed that important numerical simulations in computer graphics research, such as sparse matrix conjugate gradient solvers and regular-grid multigrid solvers can be performed efficiently on the GPU. There have been previous works using FEM [7, 8]. Knittel [13] presents a matrix-free GPU Poisson solver using the FDM Conjugate Gradient method, though their work is in the context of a mini cluster with a custom FPGA interconnect. Ament et al. [1] presented a preconditioned conjugate gradient solver for the Poisson problem on a multi-GPU platform with overlapping memory transfers, thus increasing concurrency and scalability.

3. BACKGROUND

3.1 CUDA Memory Model

CUDA was introduced in 2007 by NVIDIA to enable programming general purpose computation on parallel GPU architectures [18]. Figure 1 shows different types of memory available: global memory, texture memory, and shared memory.

Global memory is device memory visible to every thread in the same compute grid with large size; Shared memory is visible to threads in the same compute block, and is very fast to access, but much smaller capacity than global memory. In addition, memory can be accessed using texture operations designed for the graphics pipeline. Texture memory shares space with the global memory, including caching in 2D blocks and the option to use specialized linear blending hardware. Some algorithms benefit from texture memory over global memory, if threads have sufficient spatial locality to benefit from the caching of the texture memory [6].

3.2 Poisson Equation

Let $\Omega = (0,1) \times (0,1) \subset R^2$ be the open unit square in two dimensions. The notation $\partial\Omega$ denotes the boundary of Ω and $\bar{\Omega}$ the closure of Ω, that is, $\bar{\Omega} = [0,1] \times [0,1] \subset R^2$. The Poisson equation with Dirichlet boundary condition is then

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \partial\Omega. \end{aligned} \tag{1}$$

The Laplace operator is defined as

$$\Delta u = \frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2} \tag{2}$$

To solve this problem using finite differences, the unit square is represented by $N+2$ mesh points in each dimension, so there are $N \times N$ interior unknown mesh points. We construct the mesh with uniform mesh spacing h, where $h = 1/(N+1)$; each mesh point is defined by $(x_{k_1}, x_{k_2}) \in \bar{\Omega}$, where $x_{k_i} = h\,k_i$, $k_i = 0, 1, \ldots, N, N+1$ in every dimension. The approximation of each mesh point is $u_{k_1,k_2} \approx u(x_{k_1}, x_{k_2})$; then the approximation of second-order derivative in the Laplace operator (equation 2) at N^2 interior mesh points is:

$$\begin{aligned} &\frac{\partial^2 u(x_{k_1}, x_{k_2})}{\partial x_1^2} + \frac{\partial^2 u(x_{k_1}, x_{k_2})}{\partial x_2^2} \\ &\approx \frac{u_{k_1,k_2-1} + u_{k_1-1,k_2} - 4u_{k_1,k_2} + u_{k_1+1,k_2} + u_{k_1,k_2+1}}{h^2} \end{aligned} \tag{3}$$

Collect the approximation of the $N \times N$ interior mesh points and store them in a vector $u \in R^{N^2}$. This vector can be considered as the N^2 unknowns of a linear equation system

$$Au = b \tag{4}$$

From equation (3), we know the matrix $A \in R^{N^2 \times N^2}$, has a structure as shown in Figure 2. The yellow matrices are negative identity matrices, where $T = -I \in R^{N \times N}$ and the pink matrices are the tri-diagonal matrices $S = \text{tridiag}(-1, 4, -1) \in R^{N \times N}$.

In this paper, we will be using an evaluation function for the right-hand side vector $b \in R^{N^2}$ defined in equation (5), and the solution to this test problem is as described in equation (6). This is the classical elliptic test problem of the Poisson equation with homogeneous Dirichlet boundary conditions.

$$\begin{aligned} f(x) = -2\pi^2\big(&\cos(2\pi x_1)\sin^2(\pi x_2) + \\ &\sin^2(\pi x_1)\cos(2\pi x_2)\big) \end{aligned} \tag{5}$$

$$u(x_1, x_2) = \sin^2(\pi x_1)\sin^2(\pi x_2) \tag{6}$$

Based on our knowledge of matrix A, the 2^{nd}-order derivative partial differential equation can be transformed into a problem of solving a system of linear equations, we will discuss the algorithms we designed for solving the linear system in the following section.

3.3 The Conjugate Gradient Method

The conjugate gradient method is an iterative method used to solve a system of linear equations with a symmetric positive definite system matrix, such as equation (4). Its pseudocode is presented in Algorithm 1, we refer readers to work by Hestenes and Stiefel [10] for a detailed description. In this algorithm, there are two instructions that require matrix-vector multiplications (line 6 and line 19, highlighted), one of which is in the while-loop (line 19), and is executed in every iteration. Since a larger matrix takes more iterations to converge, the impact of the performance of the matrix-vector multiplication operation is non-trivial.

Algorithm 1 Conjugate Gradient Method to solve $Ax = b$

```
Require: $A^T = A, x^T Ax > 0 \quad (0 \neq x \in R^n)$
 1: Input right-hand side $b$, initial guess $x$, tolerance $tol$,
    and maximum number of iterations $maxit$
 2: if $||b||_2 = 0$ then
 3:    set $x = 0$ and stop
 4: end if
 5: $tolb = tol||b||_2$
 6: Compute $r = b - Ax$
 7: if $||r||_2 \leq tolb$ then
 8:    the initial guess $x$ is within the tolerance and stop
 9: end if
10: Initialize $\rho = r^T r$ and $k = 0$
11: while $||r||_2 > tolb$ and $k < maxit$ do
12:    Increment $k \leftarrow k + 1$
13:    if $k = 1$ then
14:       Set $p = r$
15:    else
16:       $\beta = \rho \ / \ \rho^{(old)}$
17:       Update $p \leftarrow r + \beta p$
18:    end if
19:    Compute $q = Ap$
20:    $\alpha = \rho \ / \ p^T q$
21:    $x \leftarrow x + \alpha p$
22:    $r \leftarrow r - \alpha q$
23:    $\rho^{(old)} = \rho$
24:    $\rho = r^T r$
25: end while
26: if $k = maxit$ then
27:    issue warning and return last iterate $x$
28: end if
```

According to equation (3), the data accesses of the Conjugate Gradient method follow a certain pattern, as shown in Figure 3(a). We designed parallel algorithms utilizing direct access to global memory, texture memory, and explicit paging into shared memory of the GPUs to investigate the performance differences of these memory access strategies.

3.4 BLAS and cuBLAS Functions

In Algorithm 1, line 2, 7, 14, 17, 20, 21, 22 and line 24 involve algebraic operations between vectors, and we applied the CUDA accelerated level 1 BLAS functions to them. We use cuBLAS where we can, but cannot use existing level 2 BLAS functions or their counterparts in the cuBLAS library for any operations involving the structured matrix A, since it is not stored anywhere in the application.

4. CUDA ALGORITHMS

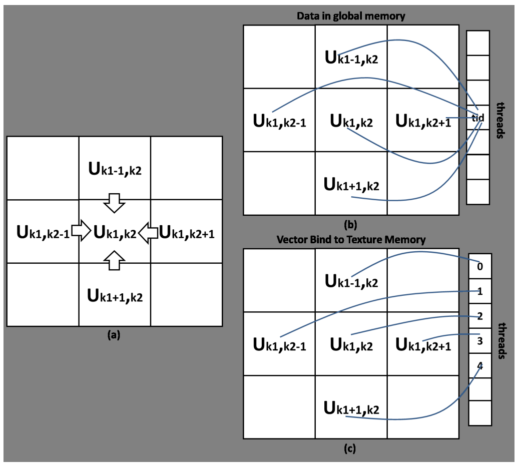

Figure 3: (a) Data Access Pattern for Matrix-free Matrix-vector Multiplication; (b) Coalesced Read From Global Memory; (c) Memory layout for a vector in texture memory: threads are accessing neighboring data (spatial locality)

Lines 6 and 19 in Algorithm 1 contain multiplications between the structured matrix A and a right-hand side vector. In the case of a matrix-free solution, we can solve the system using only the elements in the vector. Even so, we still have to store the right-hand side vector in the memory and fetch elements as needed. Higher-resolution meshes have large vectors, the overhead of fetching the data in each iteration is a performance bottleneck.

We have designed three algorithms to solve this linear system on the GPU. All three store the large vector data in global memory, but they differ in how the data is accessed each iteration. The first version, which we will call the *direct global memory* solution, uses *coalesced* data reads, with repeated reads for each access to the same data. The second version, which we will call the *texture memory* solution, organizes the vector data to enable access using texture operations to take advantage of texture caching. The final version, which we will call the *shared memory* solution, organizes the data to allow sharing data between threads in shared memory.

4.1 Direct Global Memory

The global memory is a large DRAM store with long access latency (hundreds of cycles) and limited access bandwidth. In order to achieve better global memory access efficiency, parallel threads in a warp should read data from consecutive memory addresses to form a coalesced read (as shown in Figure 4). When threads 0, 1, through n fetch data $M_{*,0}$, $M_{*,1}$, ..., and $M_{*,n}$, the reading operations are grouped into one consolidated read.

Unknown interior mesh points are stored in a vector $u \subset R^{N \times N}$, which is a 1D array, and memory is allocated in the same way in the global memory, as shown in Figure 3(b). For instance, in order to calculate the matrix-vector multiplication concerning the element u_{k_1,k_2}, we need its value and its four neighbors in the original matrix: u_{k_1-1,k_2}, u_{k_1,k_2-1},

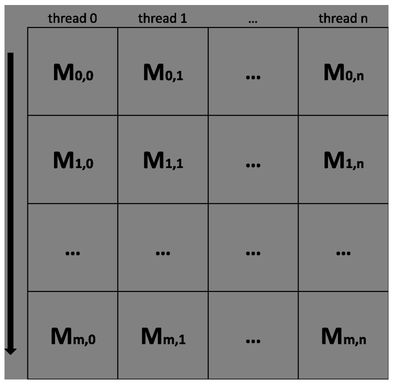

Figure 4: Coalesced Read From Global Memory

u_{k_1,k_2+1} and u_{k_1+1,k_2}. Thread *tid* is responsible for updating one element in the resulting vector by reading five elements in the global memory. Thread launches can be organized so that threads in a half-warp access contiguous global memory addresses, thus achieving higher bandwidth than non-coalesced read operations.

The CUDA kernel for this algorithm is presented in Algorithm 2. Using the global memory, every data element is read an average of five times throughout the calculation (dev_v[tid] in Line 3, dev_v[tid-size] in Line 5, dev_v[tid-1] in Line 7, dev_v[tid+1] in Line 11 and dev_v[tid+size] in Line 14). With the caching mechnism of the global memory in the Fermi GPUs, data elements dev_v[tid] and its two neighbors, dev_v[tid-1] and dev_v[tid+1] are located relatively close to each other, thus two cache hits, whereas dev_v[tid-size] and dev_v[tid+size] are *size* elements away from the element in the center, which are likely to cause cache misses. However, in our algorithm, in each block, threads are grouped into coalesced read operations, which hides the latency of fetching data from the global memory.

Algorithm 2 CUDA code: global memory

```
 1: __global__ model_mv_global(float* result, float* dev_v,
    int size)
 2: tid = threadIdx.x + blockIdx.x * blockDim.x
 3: result[tid] = 4 * dev_v[tid]
 4: if tid >= size then
 5:    result[tid] = result[tid] - dev_v[tid-size]
 6: end if
 7: if tid % size != 0.0 then
 8:    result[tid] = result[tid] - dev_v[tid-1]
 9: end if
10: if (tid+1) % size != 0.0 then
11:    result[tid] = result[tid] - dev_v[tid+1]
12: end if
13: if tid < size * size - size then
14:    result[tid] = result[tid] - dev_v[tid+size]
15: end if
```

4.2 Texture Memory

We present two algorithms using CUDA texture access functions, one using 1D texture and one using 2D texture. Both can only use single precision due to the limits of GPU texture hardware. Algorithm 3 uses 1D texture fetch operations in an attempt to utilize texture caching. Figure 3(c) presents the layout of data array after it has been bound to a 1D texture with a texture reference *tex_t*. Thread 0 through Thread 4 read from memory addresses close to each other.

Algorithm 3 Kernel: texture memory (1D)

```
 1: __global__ void model_mv_texture(float* result, int size)
 2: x = threadIdx.x + blockIdx.x * blockDim.x
 3: y = threadIdx.y + blockIdx.y * blockDim.y
 4: int offset = x + y * blockDim.x * gridDim.x
 5: result[offset] = 4 * tex1Dfetch(tex_p, offset)
 6: if offset>=size then
 7:    result[offset] -= tex1Dfetch(tex_p, offset - size)
 8: end if
 9: if offset % size != 0 then
10:    result[offset] -= tex1Dfetch(tex_p, offset -1)
11: end if
12: if (offset + 1) % size != 0.0 then
13:    result[offset] -= tex1Dfetch(tex_p, offset +1)
14: end if
15: if offset < size * size then
16:    result[offset] -= tex1Dfetch(tex_p, offset + size)
17: end if
```

Since 1D textures can only bind to a relatively small array, and texture caching is optimized for 2D textures, we also present Algorithm 4 using 2D textures. This algorithm is also able to take advantage of texture hardware boundary support, which eliminates the logic checking for boundary conditions; in addition, 2D texture memory caches data elements in blocks (2x2, 4x4, etc) depending on the implementation of the GPU.

Algorithm 4 Kernel: texture memory (2D)

```
 1: __global__ void model_mv_texture(float* result, int size)
 2: int x = threadIdx.x + blockIdx.x * blockDim.x
 3: int y = threadIdx.y + blockIdx.y * blockDim.y
 4: int offset = x + y * blockDim.x * gridDim.x
 5: float t = tex2D(tex_2, x-1, y)
 6: float l = tex2D(tex_2, x, y-1)
 7: float c = tex2D(tex_2, x, y)
 8: float r = tex2D(tex_2, x, y+1)
 9: float d = tex2D(tex_2, x+1, y)
10: result[offset] = 4*c - t - l - r - d
```

4.3 Shared Memory

Like the $L1$ cache on the CPU, shared memory is located close to the processors for each thread block, with similar performance to registers. Due to capacity limits, data blocks must be paged on and off the shared memory in a coalesced manner. According to NVIDIA's CUDA occupancy table [20], there are three factors affecting the maximum number of warps per multiprocessor: threads launched per block, registers used by each thread and the shared memory used in each thread block. For example, when 256 threads are launching the same kernel using 8 registers and 2048 bytes of shared memory in each thread block, a GPU processor with 2.0 compute capability can have up to 6 warps/block per multiprocessor which is the limiting factor of active threads in each thread block. The size of shared memory also differs between different GPU models.

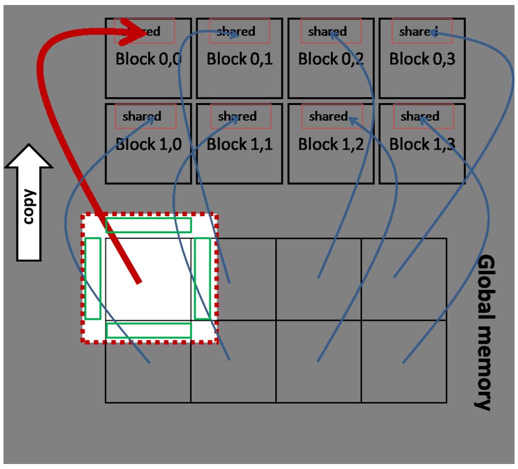

Figure 5: Data copied from global memory to shared memory in each block: the green blocks are the top, bottom, left and right ghost regions of the square block in the global memory

Shared memory is only visible to threads within the same block; threads in one thread block cannot access the shared memory in a different block. This characteristic guarantees the independence of kernel execution and memory update in each individual thread block. Communication between threads in different blocks must use global memory (Figure 1), and any synchronization is a blocking operation. To avoid inter-block communication, we duplicate data in a "ghost region" around each block.

For any problem of considerable size, the right-hand vector in line 19 of Algorithm 1 must be divided among thread blocks. Since we are doing a cross-pattern update on each element (as shown in Figure 3(a)), the larger problem we have, the more ghost regions we will have to handle. The shared memory code is implemented in several kernels, so we will not present it in the same detail as Algorithms 2-4. This code 1) copies square data blocks from the global memory to the shared memory in separate thread blocks. Ghost regions, i.e. the data adjacent to the boundaries of the square data block, are copied by separate threads. This data is copied from the global memory into four arrays located in the shared memory of each thread block (Figure 5); 2) perform cross-pattern calculation on the elements and get partial results, using elements from the ghost regions as needed (Figure 6); 3) copy the square data blocks from the shared memory back to global memory (Figure 7).

This method maximizes the independence of kernel execution in each data block and the data can be read from (Figure 5) and be written back to the global memory in a coalesced manner (Figure 7).

5. RESULTS

In this section, we present the results and performance analysis of our GPU algorithms, and compare them with serial implementation of previous work by Raim et al. [21].

5.1 Ground Truth

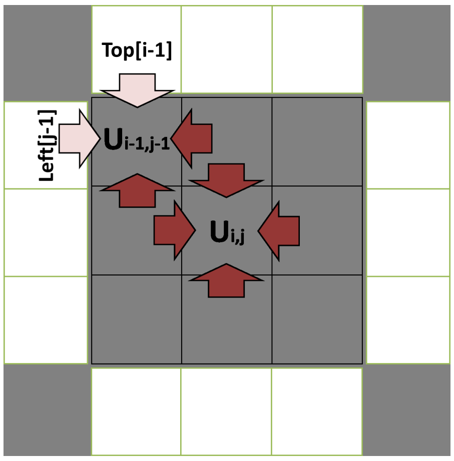

Figure 6: Calculate partial results in each block: calculating one element in the resulting vector requires four values, center element, and its four neighbors (top, bottom, left and right). $U_{i,j}$ uses four adjacent elements within the square block (indicated by dark red arrows); $U_{i-1,j-1}$ uses two adjacent elements in the square block (dark red) and two from the "ghost region" (pink)

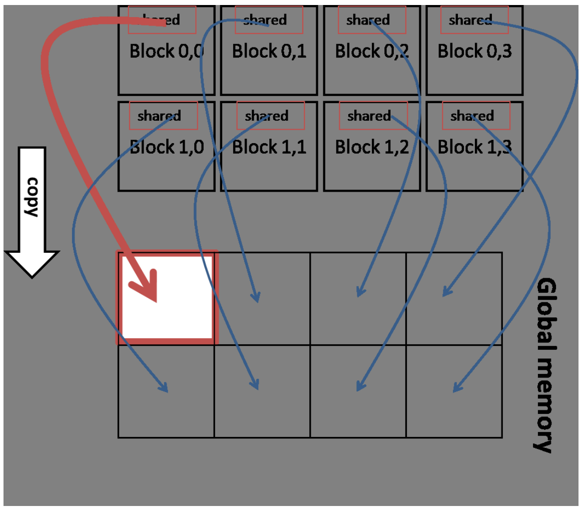

Figure 7: Data copied from shared memory/block back to global memory: only the elements in the square are copied back from the shared memory after calculation

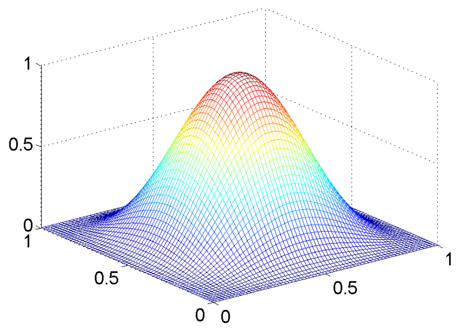

Figure 8: Numerical result u_h, $N = 64$

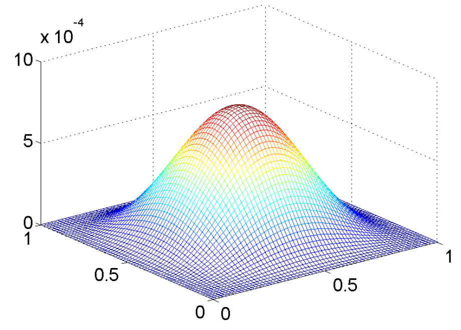

Figure 9: Error: $|u - u_h|$

In order to verify the results are numerically correct, we compare to the exact solution to the elliptic test problem as described in Section 3.2. The numerical result shown in Figure 8 is the numerical result u_h from solving the test problem with 64×64 unknowns, and Figure 9 is the error, $|u - u_h|$, where u is the true result. The maximum error is 7.7811×10^{-4}.

Since we are merely changing the time complexity of the matrix-vector multiplication in the un-preconditioned conjugate gradient method, the numerical results of the various algorithms we have discussed in the paper should converge at the same rate with the same numerical results. Our experiments verified this to be true. The results from our CUDA kernels using double precision floating points are identical to our test MPI runs.

5.2 Performance Analysis

Figure 10 is a log plot of execution time of a serial implementation and our CUDA code. The serial code ran on one compute node of a cluster with one process running. The compute node has two quad core Intel Nehalem X5550 processors (2.66 GHz, 8192 KB cache), 24 GB of memory, and a 120 GB local hard drive. Our CUDA algorithms are executed on an NVIDIA Fermi C2070 GPU with 6G memory. In order to prove the scalability of our solution, Table 1 presents execution time of the algorithm with direct access to global memory using double precision floating point numbers.

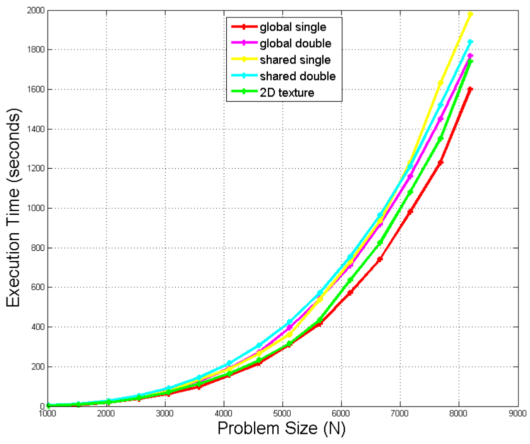

Figure 10: Execution Time for our CUDA algorithms

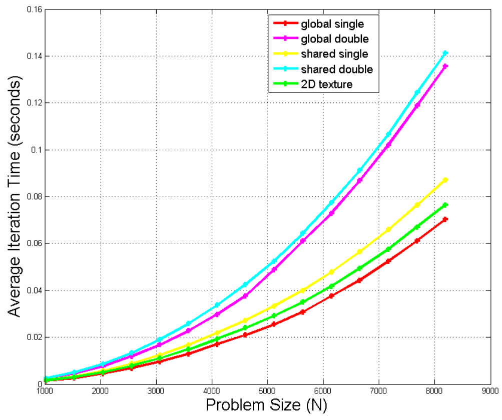

Figure 11: Average Iteration Time

Given we are performing experiments using floating point numbers with both single and double precision, the number of iterations in each CUDA algorithm differs given the same tolerance, i.e. 10^{-6} is used in our paper. We also calculated the average execution time of one iteration for each algorithm. (We timed the execution time for the kernel and divide this number by the number of iterations when the solution converges.) The average iteration time is presented in Figure 11.

From the experimental results, all CUDA kernels variants delivered approximately a 10× speedup over the serial code. The direct global memory/coalesced read algorithm with single precision floating point delivered best performance.

N	**DOF**	L_{inf}	**#iter**	**Execution Time**
32	1024	3.0128E-003	48	0.19
64	4096	7.7811E-004	96	0.21
128	16384	1.9765E-004	192	0.24
256	65536	4.9797E-005	387	0.33
512	262144	1.2494E-005	783	0.74
1024	1048576	3.1266E-006	1581	3.52
2048	4194304	7.8019E-007	3192	24.2
4096	16777216	1.9366E-007	6452	192
8192	67108864	4.7402E-008	13033	1770

Table 1: Execution Time Using Global Memory Double Precision Floating Point Numbers

Although the data are located in global memory, by grouping threads that perform read and write operations on consecutive memory addresses, the read and write operations are coalesced for optimal speed; by launching a large amount of parallel threads at the same time, data fetching latency from the global memory is hidden.

The texture memory is located in the global memory as one chunk of cached data. Data must be bound as a texture for use, and unbound to access as regular global memory. When we bind the data to the texture memory, parallel threads' read and write operation can benefit from the data spatial locality. The binding operation happens at the beginning of every iteration, which can be expensive for larger problems. Graphics hardware is optimized for 2D texture accesses, and this is supported by our data, where the 2D texture kernel performs better than the 1D texture kernel.

Although shared memory is located on-chip, data must be fetched to it before the calculations can benefit from its register-speed memory performance. As shown in Figure 5, 6 and 7, data block and "ghost region" copy operations need additional condition checking, because every element on the data block boundary requires a different set of data elements from the global memory to be in its ghost region. This added complexity is one factor in the relatively poor performance of the shared memory algorithm.

In addition, shared memory induces a limit on total thread count. The size of shared memory available in each thread block is limited. Current GPUs with compute capability of 2.0 and above have up to 48KB shared memory per multiprocessor, split between the thread blocks sharing that multiprocessor. In our shared memory CUDA kernel, two vectors with size of P^2 are allocated to store the data block, and the vector of partial results; another four vectors of size P to store the ghost regions. With just one thread block per multiprocessor, we have

$$(2P^2 + 4P) * sizeof(float) \leq 48KB \qquad (7)$$

This gives $P \leq 77$ (effectively $P = 64$) for single precision or $P \leq 54$ (effectively $P = 32$) for double precision.

6. CONCLUSION

In this paper, we present a GPU-based solution to the classical elliptic test problem of a Poisson equation with Dirichlet boundary conditions in two spatial dimensions. Using a matrix-free implementation of the iterative Conjugate Gradient method relieves some memory concern, which is the major bottle neck of modern graphics processing units for GPGPU. We performed a performance study on three options for GPU memory access. From this study, we have shown that coalesced direct access to global memory, even with repetitive accesses to the same global memory locations, performs best on this type of problem. Next best is using 2D texture memory access, though this limits the computation to single precision. Somewhat surprisingly, shared memory performed the worst in this problem space.

7. ACKNOWLEDGMENTS

The first author acknowledges partial financial support from the University of Maryland, Baltimore County (UMBC) High Performance Computing Facility (HPCF) and Maryland Industrial Partnerships / Sensics. NVIDIA hardware was provided by NVIDIA Research through the NVIDIA CUDA Teaching Center program. The high performance cluster used in the computational studies (serial MPI code) is part of the HPCF. The facility is supported by the U.S. National Science Foundation through the MRI program (grant no. CNS-0821258) and the SCREMS program (grant no. DMS-0821311), with additional substantial support from the UMBC.

8. REFERENCES

[1] M. Ament, G. Knittel, D. Weiskopf, and W. Strasser. A Parallel Preconditioned Conjugate Gradient Solver for the Poisson Problem on a Multi-GPU Platform. In *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, PDP '10, pages 583–592, Washington, DC, USA, 2010. IEEE Computer Society.

[2] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: a portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, Supercomputing '90, pages 2–11, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[3] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Trans. Comput.*, 38:1526–1538, November 1989.

[4] G. E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39:85–97, March 1996.

[5] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *ACM TRANSACTIONS ON GRAPHICS*, 22:917–924, 2003.

[6] L. Brandon, C. Boyd, and N. Govindaraju. Fast Computation of General Fourier Transforms on GPUs. In *Multimedia and Expo, 2008 IEEE*

International Conference on, pages 5 –8, June 23 2008-April 26 2008 2008.

[7] A. Camargos, R. Batalha, C. Martins, E. Silva, and G. Soares. Superlinear Speedup in a 3-D Parallel Conjugate Gradient Solver. *Magnetics, IEEE Transactions on*, 45(3):1602 –1605, March 2009.

[8] R. Carvalho, C. Martins, R. Batalha, and A. Camargos. 3D Parallel Conjugate Gradient Solver Optimized for GPUs. In *Electromagnetic Field Computation (CEFC), 2010 14th Biennial IEEE Conference on*, page 1, May 2010.

[9] B. He, N. K. Govindaraju, Q. Luo, and B. Smith. Efficient gather and scatter operations on graphics processors. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 46:1–46:12, New York, NY, USA, 2007. ACM.

[10] M. Hestenes and E. Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards*, 49:6, December 1952.

[11] P. Hudak, J. Hughes, S. P. Jones, and P. Wadler. A history of Haskell: Being lazy with class. In *In Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages (HOPL-III)*, pages 1–55. ACM Press, 2007.

[12] K. E. Iverson. *A programming language*. John Wiley & Sons, Inc., New York, NY, USA, 1962.

[13] G. Knittel. A CG-based Poisson solver on a GPU-cluster. In *High Performance Computing (HiPC), 2010 International Conference on*, pages 1 –10, Dec. 2010.

[14] B. Larsen. Simple optimizations for an applicative array language for graphics processors. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, DAMP '11, pages 25–34, New York, NY, USA, 2011. ACM.

[15] A. E. Lefohn, S. Sengupta, J. Kniss, R. Strzodka, and J. D. Owens. Glift: Generic, efficient, random-access GPU data structures. *ACM Trans. Graph.*, 25:60–99, January 2006.

[16] Y. Li, J. Dongarra, and S. Tomov. A note on auto-tuning GEMM for GPUs. In *Proceedings of the 9th International Conference on Computational Science: Part I*, ICCS '09, pages 884–892, Berlin, Heidelberg, 2009. Springer-Verlag.

[17] R. Nath, S. Tomov, T. T. Dong, and J. Dongarra. Optimizing symmetric dense matrix-vector multiplication on GPUs. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 6:1–6:10, New York, NY, USA, 2011. ACM.

[18] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide Version 1.0*, June 2007.

[19] NVIDIA. *NVIDIA cuBLAS Library*, 4.0 edition, 2011.

[20] NVIDIA. *NVIDIA CUDA Occupancy Table*. NVIDIA, Dec 2011.

[21] A. M. Raim and M. K. Gobbert. Parallel Performance Studies for an Elliptic Test Problem on the Cluster Tara. Technical Report HPCF–2010–2, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2010.

[22] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan Primitives for GPU computing. In *GRAPHICS HARDWARE 2007*, pages 97–106. Association for Computing Machinery, 2007.

[23] B. J. Smith. R package magma: Matrix Algebra on GPU and Multicore Architectures, version 0.2.2, August 27, 2010. [On-line] http://cran.r-project.org/package=magma.

[24] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 31:1–31:11, Piscataway, NJ, USA, 2008. IEEE Press.