GPU Acceleration of Solving Parabolic Partial Differential Equations Using Difference Equations

David L. Foster
Electrical and Computer Engineering Department, Kettering University, Flint, MI, USA

Abstract- *Parabolic partial differential equations are often used to model systems involving heat transfer, acoustics, and electrostatics. The need for more complex models with increasing precision drives greater computational demands from processors. Since solving these types of equations is inherently parallel, GPU computing offers an attractive solution for drastically decreasing time to completion, power usage, and increasing the computation per dollar. However, since GPU computing involves a much different programming paradigm than traditional processors, techniques for optimizing solvers must still be developed. This paper presents several optimization strategies for accelerating solvers using CUDA to implement difference equations and compares their performances to a standard processor. The results demonstrate that different strategies should be used for different GPU cards, such as the C1060 and GTX 480, resulting in up to 197 times and 257 times single-precision and up to 133 and 163 times double-precision speedups respectively.*

Keywords: GPU programming, CUDA, parabolic partial differential equations, convection-diffusion-reaction equation

1 Introduction

Parabolic partial differential equations (PDEs) are useful in several problem spaces, such as heat transfer, acoustic modeling, and electrostatics. While numerical solution techniques are well-known, there are significant needs that can be addressed by solving them with GPU computing. With well-crafted algorithms, GPUs can potentially solve systems significantly faster, allowing decreased simulation times and/or increased resolution in the model. GPUs can also solve problems using an estimated one tenth to one twentieth of the power required by traditional supercomputing systems [1], thereby reducing costs.

This paper utilizes CUDA for GPU computing, which is an extension to several common programming languages that requires an NVIDIA-based video card for execution. NVIDIA GPUs are widely deployed and thus represents a very common computing platform. Additionally, NVIDIA's Tesla series specifically targets high-performance computing. For easy scalability, NVIDIA cards are designed around a generalized processing unit called a streaming multiprocessor (SM). This allows the performance of CUDA applications to scale based on the number and hardware implementation of the SMs contained on a given card.

This paper proposes a set of optimizations for solving parabolic PDEs target several issues that arise when porting code to CUDA which must be deliberately addressed for efficient use of the GPU [2, 3]. First, GPU architectures generally mitigate memory latency by using large numbers of threads and extremely fast context switching instead of deep memory caches found in CPUs, although the Fermi architecture released in 2010 did add some memory caching. This requires optimizations to exploit enough parallelism to make sufficient threads available and to carefully manage the number of memory accesses required. Second, data accesses should be formed into coalesced reads and writes. See [2] for more details on coalesced accesses. Next, each SM has a limited amount of shared memory, usually 8 kB to 48 kB, that can be leveraged to reduce the number of RAM accesses. Finally, SMs are designed to execute a group of 32 threads, called a warp, concurrently. However, all threads in the warp must be executing the same instruction. Branching code creates divergence, which can drastically lower the throughput as only a subset of the warp executes. Carefully constructed code that limits divergence can minimize this effect.

Difference equations were used to solve several model parabolic PDEs, and optimizations were developed to address the above issues. Previous work in GPU computing focused on this topic include [4] which attained a 1.1 to 11 times performance improvement on two-dimensional parabolic PDEs using double-precision floating point arithmetic. Another effort targeted one-dimensional PDEs for market making real time pricing, and risk management achieved a 25 times speedup over a well-optimized CPU implementation using a single Tesla C1060, and a 38 times improvement using two C1060s by leveraging cyclic reduction [5]. A mixed precision method has been presented to solve ill-conditioned tridiagonal systems that previously were limited to CPU solutions with a 10-fold improvement[6].

Using the techniques proposed in this paper on two-dimensional parabolic PDEs, speed-ups of 197 times and 257 times were achieved using a C1020 Tesla and a GTX 480 respectively on single-precision floating point performance compared to an Intel i7 920. Speedups up to 133 and 163 times were attained on double-precision floating point performance.

This paper is organized as follows. Section 2 outlines the general approach used to solve parabolic PDEs using CUDA. Section 3 describes three optimization strategies that were implemented. Section 4 explains the experimental setup used. Section 5 presents the results of testing these optimizations, and Section 6 concludes the paper.

2 General Approach to Solver

This work uses difference equations to solve 2-dimensional parabolic PDEs of the form

$$u_t = u_{xx} + u_{yy} + A(x,y)u_x + B(x,y)u_y + C(x,y)u + f(x,y,t).$$

The boundary equations are expressed as

$$u(0,y,t) = g_1(y,t),\ u(1,y,t) = h_1(y,t) \quad 0 < y < 1,\ t > 0$$
$$u(x,0,t) = g_2(x,t),\ u(x,1,t) = h_2(x,t) \quad 0 < x < 1,\ t > 0$$

and the system has the starting condition

$$u(x,y,0) = l(x,y)$$

where g_1, g_2, h_1, h_2, and l are determined by the system being modeled.

The two-dimensional grid is cast as a red-black array as shown in Figure 1. Red points are adjacent a black point on all four sides, and black points are adjacent to a red point of all four sides, much like a checkerboard. The red-back array is surrounded on all sides by a single row of points corresponding to the boundary conditions. The advantage to this model is that all red points may be updated in parallel using values of neighboring black points. Then the black points can be updated using the new values of the red points.

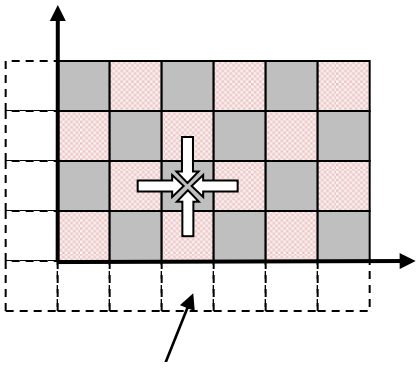

Figure 1 Red-Black Array showing an arbitrary black point's dependence on its four neighboring red points

The method to break this array into thread blocks needed to access memory efficiently and create enough thread blocks to occupy the GPU's SMs. This was accomplished by subdividing the array into sets of rows, such as 16 rows for example. Each block was assigned to 2N columns of a set in which each row contained N red points and N black points.

The general algorithm is represented by the following pseudo-code.

```
if using GPU: transfer red-black array to GPU RAM
for the required number of time steps
    update the boundary conditions
    update red points
    update black points
if using GPU: transfer red-black array to CPU RAM
```

3 Performance Optimizations

This section details the three main optimizations used.

3.1 Separated Red and Black Arrays

The first optimization split the unified red-black array into a red array and a separate black array as shown in the example in Figure 2. The separate arrays contained the same number of rows but had only half the number of columns as the original array. The separation was performed on the CPU, and the red and black arrays were passed to the CUDA kernel.

This optimization focused on two issues based on the following observation. When accessing N red points (similarly for black) in the unified array, they were interleaved with N black points. The accesses were not coalesced since the N required points were contained in a span of contiguous memory 2N points long. The default method would be to read in all 2N points and discard or save the black points for later use. With effective use of shared memory, this method can use all data read. However, writing the updated values of N red points back to the GPU card's RAM still required writing N points over a span of 2N contiguous locations with two writes. By separating the arrays, this write-back required only one write. Additionally, if a thread warp read in a set of interleaved red and black points from a unified array, the code must diverge so that the two subsets were handled differently. Using separated arrays avoided this divergence since the entire warp handled either red or black points.

3.2 Per Block Work Reduction

The second optimization slightly reduces the amount of work per thread block to reduce the number of sequential memory accesses. It was observed that to update N points,

4N+1 points must be read from memory: the N points being updated, the N points from the row above and from the row below,, and N+1 points from the adjacent points on the same row. This requires one thread to make 5 sequential memory reads while the remaining threads in the block make only 4. To eliminate the extra latency, a block of N threads was coded to update N-1 points. This requires 4N-3 points from memory taking 4 sequential reads from each thread.

Using the separated arrays from Figure 2 with N=3, suppose a block of three threads is solving for all three red points in row 3. The block would make a request using all threads for (3,1), (3,3), and (3,5) from the red array, a second request for (4,1), (4,3), and (4,5) from black, a third request for (2,1), (2,3), and (2,5) from black, a fourth request for (3,0), (3,2), and (3,4) from black, and finally a fifth request for only (3,6) from black. If the block is solving for N-1 points, only the following four requests are needed. The block would make a request using all threads for (3,1), (3,3), and (3,5) from the red array and discard (3,5), a second request for (4,1), (4,3), and (4,5) from black discarding (4,5), a third request for (2,1), (2,3), and (2,5) from black discarding (2,5), a fourth request for (3,0), (3,2), and (3,4) from black.

Note that for a thread block of 128 threads handling 256 columns, the amount of work was reduced by only 0.78%. In many cases, this small decrease would be completely masked by using unutilized threads. For example, an array 1000 points wide is spanned by four thread blocks handling either 128-points or 127-points per block.

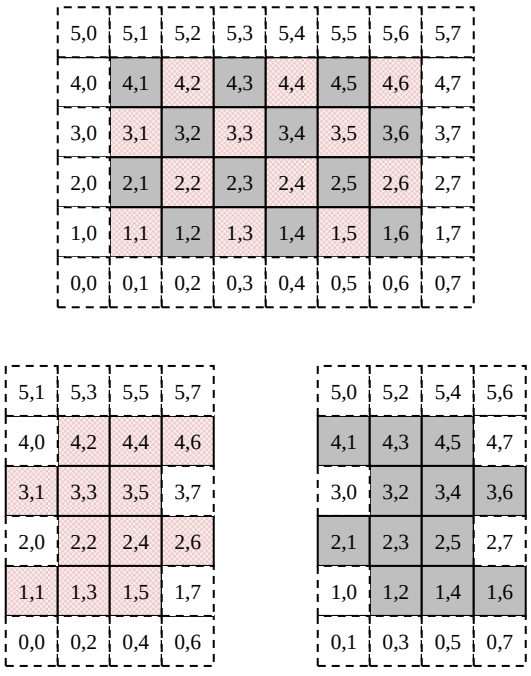

Figure 2 Dividing the Red-Black Array into separate red and black arrays. Indexes shown in the separate arrays are those from the original unified array.

3.3 Shared Memory

The final optimization reduced redundant memory reads. It can be easily seen that when updating a row of red points, for example, the black points on the same row and the black points in row below will be the upper and adjacent points respectively for updating the red points in the row below. Thus, these two rows of black points can be stored in the shared memory space, which may be faster than accessing RAM. Once the thread block loaded 4N points to update the first row of points, it only read an additional 2N points to calculate each additional row: N more red points and N more black points.

Using the separated arrays in Figure 2 as an example, suppose that a block of three threads first solved for the red points in row 4. The block would need to read row 4 from the red array and data from rows 3, 4, and 5 from the black array. If the block then solved for the red points in row 3, it would need data from row 3 of the red array and rows 2, 3, and 4 of the black array. If black rows 3 and 4 were saved in local shared memory, the block would only need to access RAM for the red row and row 2 of the black array.

4 Testing

This section details the four parabolic PDE models tested, the hardware setups, and the test parameters.

4.1 Model Equations Tested

The four models used were taken from [4] for purposes of comparison and since they have exact solutions for validation. All models are constrained by the conditions $0.0 \le x \le 1.0$, $0.0 \le y \le 1.0$, and $0.0 \le t \le 1.0$.

Model 1:

$u_t = u_{xx} + u_{yy}$

$u(0,y,t) = 0.0,\ u(1,y,t) = 0.0$	$0 < y < 1,\ t > 0$
$u(x,0,t) = 0.0,\ u(x,1,t) = 0.0$	$0 < x < 1,\ t > 0$
$u(x,y,0) = \sin(\pi x)\sin(\pi y)$	$0 < x,y < 1$

This model has the exact solution

$$u(x,y,t) = e^{-2\pi^2 t} \sin(\pi x)\sin(\pi y).$$

Model 2:

$u_t = u_{xx} + u_{yy} - u_x - u_y - u$

$u(0,y,t) = e^{-t+y},\ u(1,y,t) = e^{-t+1+y}$	$0 < y < 1,\ t > 0$
$u(x,0,t) = e^{-t+x},\ u(x,1,t) = e^{-t+1+x}$	$0 < x < 1,\ t > 0$
$u(x,y,0) = e^{x+y}$	$0 < x,y < 1$

This model has the exact solution

$u(x,y,t) = e^{-t+x+y}$.

Model 3:

$u_t = u_{xx} + u_{yy} + \sin(ax)\cos(ay)u_x - \cos(ax)\sin(ay)u_y - u$

$u(0,y,t) = 0.0$

$u(1,y,t) = e^{(-2a^2-1)t}\sin(a)\sin(ay)$ $\quad 0 < y < 1,\ t > 0$

$u(x,0,t) = 0.0$

$u(x,1,t) = e^{(-2a^2-1)t}\sin(a)\sin(ax)$ $\quad 0 < x < 1,\ t > 0$

$u(x,y,0) = \sin(ax)\sin(ay)$ $\quad 0 < x,y < 1$

This model has the exact solution

$$u(x,y,t) = e^{(-2a^2-1)t}\sin(ax)\sin(ay)$$

Model 4:

$u_t = u_{xx} + u_{yy} + u_x + u_y + u + (1 + xy)\cos(t)$
$- (1 + x)(1 + y)\sin(t)$

$u(0,y,t) = \sin(t)$, $u(1,y,t) = (1+y)\sin(t)$ $\quad 0 < y < 1,\ t > 0$

$u(x,0,t) = \sin(t)$, $u(x,1,t) = (1+x)\sin(t)$ $\quad 0 < x < 1,\ t > 0$

$u(x,y,0) = \sin(\pi x)\sin(\pi y)$ $\quad 0 < x,y < 1$

This model has the exact solution

$$u(x,y,t) = (1+xy)\sin(t).$$

4.2 Hardware Setup

The computer used for testing had the following specifications:

- Intel Core i7 920 at 2.67 GHz
- Asus P6T Deluxe motherboard
- 6 GB of 1333 MHz DDR3 RAM
- EVGA GeForce 260 GTX video card
- 750 GB 7200 RPM hard drive
- Windows 7 Professional 64-bit
- CUDA version 3.1
-

The two graphics cards used for GPU computing were an NVIDIA Tesla C1060 with 4 GB of RAM and 240 cores in 30 SMs and an EVGA GeForce 480 GT with 1.5 GB of RAM and 480 cores in 15 SMs at stock clock speeds.

4.3 Software Parameters

Thread blocks were launched using 128 threads. This was the smallest value that allowed all 1024 thread slots available in the GPUs' SMs tested to be utilized. Larger values would have left more thread slots unoccupied if register and shared memory usage prevented 8 blocks from being assigned per SM.

Each of the models was tested using square arrays from 500 points per side to 4000 points per side in increments of 500 points. The number of rows contained in a set, as explained in Section II was tested at 16, 32, and 64 rows per set.

5 Results

The four versions of the kernel tested were CPU; GPU, which contained no optimizations; GPU-RB, which separated the data into separate red and black arrays; and GPU-RBS, which contained the separation of arrays, used the per-block work reduction, and used shared memory. All data refers to tests with 16-row pitch.

Several interesting patterns were noted in all tests. First, the throughput of the CPU implementation declined quickly in all tests from 500 to about 1500 points per dimension and gradually tapered off more as the data set sizes continued to increase. The rapid increase corresponded to the size of the data sets exceeding the L3 cache of the CPU, indicating that the CPU performance was likely limited by memory bandwidth. This effect can be seen in Table 1 for the C1060's performance on Model 2 for single-precision, showing the throughput of the different versions in millions of points updated per second based on the number of points per side.

Table 1 Tesla C1060 Throughput on Model 2 with Single-Precision Floating Point

Points per Side	**CPU** (Mpts/s)	**GPU** (Mpts/s)	**GPU-RB** (Mpts/s)	**GPU-RBS** (Mpts/s)
500	65.96	1078.39	1773.51	2754.44
1000	59.07	1230.28	1772.02	3182.40
1500	44.84	1266.84	1935.90	3770.77
2000	41.49	1302.58	1956.86	3854.73
2500	37.93	1170.87	1976.39	3940.19
3000	37.50	1120.09	1976.48	3958.37
3500	35.79	1149.27	2003.34	3983.28
4000	30.67	1152.60	1978.63	3989.22

Over the same range of 500 to 1500 points per side, the throughput of the GPU versions greatly increased. This effect was related to the number of threads that were created based on the dimensions of the problem. From the previous section, a GPU block of 128 threads processed 254 columns and either 16, 32, or 64 rows. For a 16-row block and a 500 by 500 point array, this yielded only 64 blocks. Since the C1060 can hold 240 blocks of this size concurrently, the smaller data sets did not fully occupy the card, and the MPs suffered more idle time during memory accesses. Larger data sets allowed the GPUs to mitigate this latency more efficiently. It should also be noted that in addition to fully occupying the card on smaller data sets, the 16-row pitch also showed slightly better throughput by a few percent once over 32- and 64-row pitches.

Omitting the small data sets that didn't result in full occupancy of the GPU, Table 2 and Table 3 show the mean,

minimum, and maximum speeds for both GPUs on the models for 1500 to 4000 points per dimension. Full test results are not shown for space considerations. For both cards, the separated red and black arrays yielded a substantial speed increase over the basic GPU implementation with a unified red-black array. As can be seen by comparing the GPU and GPU-RB columns, this optimization often yielded speeds of around 50%.

For the Tesla, which is based on the GT200 series GPU that does not have memory caching, the use of shared memory gave significant benefits. For single-precision, the additional optimizations in GPU-RBS gave a 27.9% to 99% increase over the GPU-RB performance. For double-precision, it yielded almost double the performance for Models 1 and 2, significant increases in Model 4, and made notably little improvement in Model 3. However, since the 400-series GPUs, like that in the GTX 480, do have a memory cache, the use of shared memory required too much overhead to produce a benefit. Thus, the performance over GPU-RBS compared just GPU-RB was often about 10% slower, with around a 25% decrease for Model 2 with single precision.

It is also noteworthy that the newer GTX 480 did not dominate the C1060 as might be expected. The C1060 is based on the same GPU as the 280 GTX video cards, which were the predecessors to the GTX 480. In single-precision performance, the C1060 was comparable to the GTX 480 on Model 1, dominated on Model 2, was within 90% of the performance on Model 3, and was faster on data sets 3000 points per side and smaller on Model 4. For double-precision, the C1060 dominated the GTX 480 on Models 2 and 4, but the GTX 480 dominated on the other two. Therefore, these results show that different optimizations are beneficial based on the GPU architecture used, and that older architectures may still be more advantageous for certain problems.

6 Conclusions

This research demonstrates several advantageous techniques for accelerating difference equation solvers for two-dimensional parabolic PDEs. An important result is that optimization strategies differ based on the underlying GPU hardware, and effective GPU computing programming practices need to account for this.

7 Acknowledgments

This research was supported by an equipment donation from the NVIDIA Corporation as part of the Academic Partnership Program.

Table 2 Mean, Minimum, and Maximum Speedups on 1500 by 1500 to 4000 by 4000 point data sets on single-precision floating point numbers

	GPU	**GPU-RB**	**GPU-RBS**
Model 1 C1060	31.82, 28.13, 38.09	52.15, 42.75, 64.49	102.97, 82.33, 129.41
Model 1 GTX 480	63.85, 53.27, 79.47	101.23, 82.70, 129.27	92.77, 73.20, 119.47
Model 2 C1060	31.68, 28.25, 37.58	52.61, 43.17, 64.51	104.63, 84.09, 130.07
Model 2 GTX 480	53.15, 47.13, 64.41	74.88, 65.60, 89.47	56.81, 49.03, 68.61
Model 3 C1060	49.44, 46.44, 51.80	80.88, 70.29, 91.07	103.45, 90.74, 91.07
Model 3 GTX 480	83.50, 77.19, 95.07	110.37, 93.70, 124.70	102.68, 88.10, 115.54
Model 4 C1060	80.40, 74.83, 84.75	132.53, 108.24, 146.92	177.22, 148.76, 197.27
Model 4 GTX 480	126.66, 100.82, 164.79	189.86, 146.82, 257.16	154.45, 116.43, 216.70

Table 3 Mean, Minimum, and Maximum Speedups on 1500 by 1500 to 4000 by 4000 point data sets on double-precision floating point numbers

	GPU	**GPU-RB**	**GPU-RBS**
Model 1 C1060	26.05, 19.33, 31.85	45.04, 32.46, 55.55	89.48, 63.72, 111.64
Model 1 GTX 480	71.06, 48.97, 90.16	132.27, 91.70, 163.52	123.37, 84.71, 158.43
Model 2 C1060	26.10, 20.09, 31.60	45.32, 33.53, 55.70	90.46, 66.36, 112.43
Model 2 GTX 480	49.41, 37.96, 61.84	75.94, 55.13, 95.96	71.41, 51.21, 90.39
Model 3 C1060	19.15, 16.60, 21.15	27.11, 25.04, 28.90	27.83, 25.58, 30.04
Model 3 GTX 480	59.54, 53.72, 64.95	68.38, 60.35, 75.55	67.16, 59.44, 74.59
Model 4 C1060	39.33, 33.70, 44.01	68.90, 60.84, 75.44	117.27, 104.70, 133.84
Model 4 GTX 480	61.13, 53.58, 74.56	87.41, 76.18, 105.32	81.16, 70.65, 98.73

References

[1] NVIDIA. (2010, Nov. 23). *Tesla C2050/C2070 GPU Computing Processor Overview*.

[2] D. B. Kirk and W.-M. W. Hwu, *Programming Massively Parallel Processors: a Hands-On Approach*: Morgan Kaufmann Publishers, 2010.

[3] J. Sanders and E. Kandrot, *CUDA by Example, An Introduction to General-Purpose GPU Programming*: Addison Wesley, 2010.

[4] C.-W. Hsieh, *et al.*, "Rapid Performance of Parabolic Problems using Convection Diffusion Reacion on GPU Accelerator," presented at the PDPDA'10, Las Vegas, NV, USA, 2010.

[5] D. Egloff, "High Performance Finite Difference PDE Solvers on GPUs," QuantAlea GmbH, Zurich, Switzerland2010 2010.

[6] D. Göddeke and R. Strzodka, "Cyclic Reduction Tridiagonal Solvers on GPUs Applied to Mixed Precision Multigrid," *IEEE Transactions on Parallel and Distributed Systems* vol. 22, pp. 22-32, Jan. 2011 2011.