CUDA 2D Stencil Computations for the Jacobi Method

José María Cecilia[1], José Manuel García[1], and Manuel Ujaldón[2]

[1]*Computer Engineering and Technology Department, University of Murcia, Spain*
[2]*Computer Architecture Department, University of Malaga, Spain*

Abstract

This paper explores stencil operations in CUDA to optimize on GPUs the Jacobi method for solving Laplace's differential equation. The code keeps constant the access pattern through a large number of loop iterations, that way being representative of a wide set of iterative linear algebra algorithms. Optimizations are focused on data parallelism, threads deployment and the GPU memory hierarchy, whose management is explicit by the CUDA programmer. Experimental results are shown on Nvidia Teslas C870 and C1060 GPUs and compared to a counterpart version optimized on a quadcore Intel CPU. The speed-up factor for our set of GPU optimizations reaches 3-4x and the execution times defeat those of the CPU by a wide margin, also showing great scalability when moving towards a more sophisticated GPU architecture and/or more demanding problem sizes.

Keywords CUDA, GPGPU, Stencil Computation, Parallel Numerical Algorithms

1 Introduction

The newest versions of programmable GPUs provide a compelling alternative to traditional CPUs, delivering extremely high floating point performance for scientific applications which fit their architectural idiosyncrasies [8]. This fact has attracted GPUs to researchers in many fields [5], among which numerical methods constitute one of the most prolific ones.

A large number of numerical computing techniques use large multidimensional arrays as its primary data structure, which bring us a good opportunity to benefit from Single Instruction Multiple Data (SIMD) parallelism. In addition, such algorithms use to have an iterative nature, that is, they tend to converge through a number of steps towards the final solution until certain condition is fulfilled. Usually, parallelism is exploited within an iteration, where each processor can work on a different subsection of the global data to produce an output which is partially communicated to other processors. Then, data are rearranged to become the input to the next iteration, which prevents from parallelizing consecutive iterations.

Stencil computations are those in which each computing node in a multi-dimensional grid is updated with weighted values contributed by neighboring nodes. These neighbours comprise the stencil, and multiple iterations across the array are usually required to achieve convergence or to simulate time steps. Among those stencil codes, our work focuses on the Jacobi method to solve Laplace's differential equation, which is a priori not an ideal partner for GPUs due to its low arithmetic intensity. We overcome this drawback by exploring a wide set of optimizations paths, which try to illustrate the strength of CUDA for high-performance computing.

The rest of the paper is organized as follows. Section 2 explains the Jacobi method. Section 3 outlines our CUDA implementation, exposes the execution times and analyzes the results. Finally, Section 4 reviews some related work and Section 5 concludes.

2 The Jacobi Method

Jacobi [6] is a popular algorithm for solving Laplace's differential equation on a square domain, regularly discretized [4]. The kernel (see Figure 1) is based on the following idea: Let us consider a body represented by a 2D array of particles, each with an initial value of temperature. This body is in contact with a fixed value of temperature on the four boundaries, and Laplace's equation is solved for all internal points to determine their temperature as the average of the four neighboring particles.

Taking this task as the computational core, a number of iterations are performed over the data to recompute average temperatures repeatedly, and the values gradually converge to a finer solution until the desired accuracy is reached. For experimental purposes, we consider a constant number of 4096 iterations. Note that iterations have to be serialized due to carried-loop dependencies, but parallelism is enabled within iterations for the computation at each particle is independent. Thus, the workload depends more on the number of iterations, whereas the amount of parallelism that can be extracted from the code relies more on the size of the 2D input matrix.

At the end, our Jacobi kernel consists of three nested loops, with the two innermost being of length N (which is the matrix dimension), and the outermost being of length k (the number of iterations) - see Figure 1. The algorithm complexity can be expressed as $O(k \cdot N^2)$.

```
for (k=0; k<4096; k++) {
 for (i=0; i<N; i++)
  for (j=0; j<N j++)
   T[i][j] = 0.2*(A[i][j]+A[i-1][j]+
              A[i+1][j]+A[i][j-1]+A[i][j+1]);
 for (i=0; i<N; i++)
  for (j=0; j<N j++)
   A[i][j] = T[i][j]; }
```

Figure 1: Jacobi's solver pseudocode.

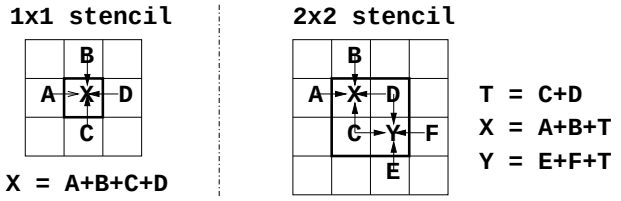

Figure 2: Benefits of increasing the stencil size: Some redundant operations may be saved.

3 Implementation

3.1 Optimal threads deployment

Blocks and threads are deployed following a 2D layout to balance the decomposition of the computational domain on each matrix dimension. Adjacent blocks share data placed on boundaries, and each thread within a block is responsible for updating a single element on each iteration.

Among all possibilities concerning an input matrix of size NxN and a squared block of BxB threads, we have selected $N = 1024, 2048, 4096, 8192$ and $B = 14, 16, 18, 20$ for representing good choices after a preliminar survey. Table 1 shows that 16x16 constitutes the optimal number of threads per block, with a penalty around 5-10% for the other three cases. All remaining squared alternatives for the matrix of threads led to worse results.

3.2 Shared memory optimizations

Our CUDA baseline implementation does not use shared memory. All threads access the device memory to read an element together with its four matrix neighbours and later update its value with the average. From this departure point, three optimizations were incrementally developed:

1. Each input element read from device memory is stored into shared memory by the owner thread prior to the actual computation, and the output result is written back into device memory. The kernel length increases from 34 to 78 instructions, but this variant notably reduces the pressure on device memory, just requiring 18 GB/s of memory bandwidth compared to 122 GB/s in our baseline version.

   On the Tesla C870, 99.68% of the memory accesses to device memory are non-coalesced when running the code using CUDA Compute Capabilities 1.0 (CCC 1.0). On the Tesla C1060, things are very different, for this device uses coalescing rules based on CCC 1.3, leading to a 100% of coalesced accesses. Benefits are therefore larger on the Tesla C1060 GPU.

2. Our second optimization uses an internal register as substitute of the shared memory cell on each thread, eliminates unnecessary synchronization barriers between threads at block level, and enables data prefetching. These enhancements behave similarly on CCC 1.0 and 1.3, and are translated into minor improvements in the overall execution time.

3. The third optimization reduces the tile size to decrease the use of shared memory. In CCC 1.0, the maximum number of threads assigned to a multiprocessor is 768, whereas in CCC 1.3 this number reaches 1024. In the first case, the tile size is decreased to reduce the amount of shared memory used (4120 bytes) so that we can assign three blocks of 256 threads to each multiprocessor. In the second case, the tile size is reduced even more until we can assign four blocks of 256 threads, which increases parallelism leading to slightly better results.

Table 2.a shows the execution times for all these versions on a Tesla C870 and Table 2.b does the same for the Tesla C1060 GPU. An average speed-up factor of 3.5x is roughly attained.

3.3 The effect of larger 2D stencils

Our next alternative kernel tries to evaluate the effect of changing the 2D stencil size, which imposes a coarser granularity on SIMD parallelism. Instead of a single element, a 2x2 matrix of elements was assigned to every thread. Using this new stencil, partial sums on diagonal elements of the matrix can be reused for computing the output elements on the other diagonal (see Figure 2), saving two arithmetic operations and four memory accesses on each thread at the expense of using two registers for storing auxiliary values.

Execution times are shown in Table 3 on a Tesla C1060 GPU for different threads deployment (depicted on rows). The input matrix size is 4096^2 and our kernel uses shared memory without further optimizations. Times slowdown 30-40% on average with respect to the case in which each thread computes a single element, proving that context switch is free in CUDA and the block startup is not: Using a 1x1 stencil we require 341x341 block calls, whereas using a 2x2 stencil, we just need 157x157 block calls.

3.4 Floating-point accuracy and performance

Peak performance on the Tesla C1060 is 933 GFLOPS in single precision and 78 GFLOPS in double precision. However, Table 4 shows that times barely double when switching from single to double precision. This basically means that our application is bandwidth limited and the fact that double precision numbers occupy 64 bits versus 32 bits for single precision explains the slowdown factor.

Matrix size	(threads deployment per CUDA block) (14x14)	(16x16)	(18x18)	(20x20)
1024^2	13.50	**13.16**	13.70	14.13
2048^2	52.73	**50.74**	52.57	52.43
4096^2	206.99	**203.35**	207.06	211.28
8192^2	843.55	**850.18**	899.46	852.26

(a) Tesla C870.

Matrix size	(threads deployment per CUDA block) (14x14)	(16x16)	(18x18)	(20x20)
1024^2	3.27	**2.34**	3.24	3.071
2048^2	12.73	**8.72**	11.88	11.594
4096^2	50.36	**34.60**	46.28	44.402
8192^2	211.03	**144.02**	211.16	177.795

(b) Tesla C1060.

Table 1: Execution times (in seconds) for our Jacobi baseline implementation.

Input matrix size	**Baseline: No shared memory used**	**Optimization 1: Using shared memory**	**Optimizations 1+2: Shared memory + coalescing**	**Optimizations 1+2+3: Also solving banks conflicts**
1024^2	13.16	3.77 (3.49x)	3.76 (3.50x)	3.88 (3.39x)
2048^2	50.74	14.49 (3.50x)	14.45 (3.51x)	14.71 (3.45x)
4096^2	203.35	55.60 (3.65x)	55.59 (3.65x)	57.45 (3.54x)
8192^2	850.18	243.00 (3.50x)	241.81 (3.51x)	241.81 (3.51x)

(a) Tesla C870.

Input matrix size	**Baseline: No shared memory used**	**Optimization 1: Using shared memory**	**Optimizations 1+2: Shared memory + coalescing**	**Optimizations 1+2+3: Also solving banks conflicts**
1024^2	2.34	0.73 (3.20x)	0.65 (3.60x)	0.63 (3.71x)
2048^2	8.72	2.79 (3.12x)	2.47 (3.53x)	2.42 (3.60x)
4096^2	34.60	11.45 (3.02x)	9.93 (3.48x)	9.66 (3.58x)
8192^2	144.02	45.70 (3.15x)	40.35 (3.57x)	40.29 (3.57x)

(b) Tesla C1060.

Table 2: Execution times (in seconds) for our Jacobi implementation using different optimizations. Between parenthesis, we show the speed-up factor versus the baseline implementation on the same platform. Threads deployment is 16x16 for all cases.

Threads deployment	**Stencil size**		**Slowdown factor**
	One	**2x2**	
14x14	13.91	19.31	38%
16x16	11.45	15.43	34%
18x18	13.27	18.16	36%
20x20	13.83	18.40	33%

Table 3: Execution times (in seconds) on a Tesla C1060 GPU for different threads deployment (depicted on rows). The input matrix size is 4096^2 and the code version uses shared memory without further optimizations. The stencil size is the number of elements computed by each thread.

Matrix size	**Single Precision**	**Double Precision**	**Slowdown factor**
1024^2	0.90	1.70	89%
2048^2	3.36	6.55	95%
4096^2	13.41	26.33	96%

Table 4: Execution times (in seconds) for the optimal version of our Jacobi implementation (that is, optimizations 1, 2 and 3 performed), using single and double precision in our Tesla C1060 GPU. We run out of memory for the 8192^2 case on double precision. Threads deployment is 16x16 for all cases.

3.5 GPU versus CPU multi-core performance

Table 5 presents the execution times that we have obtained parallelizing the Jacobi method on the GPU using CUDA and the CPU using pthreads. For the multithreaded CPU version, the best performance was obtained by assigning entire rows to each CPU core as data partition.

We can see that the Tesla C1060 GPU is unbeatable, and the C870 is also more effective that the quad-core in most of the cases, overall when working with large matrices. It can also be seen how the CPU times are poorly scalable when the working set exceeds the L2 cache size (12 MB in our case), that is, from the 2048^2 case on. In other words, the CPU cores have to rely on caches to become effective, and the Jacobi method becomes even more bandwidth limited when running on multicore CPUs.

Input matrix size	On a Tesla GPU		On an Intel Core 2 Quad Q9450 CPU				
	C870	C1060	1 core 1 thread	2 cores 2 threads	4 cores 4 threads	4 cores 8 threads	4 cores 16 threads
1024^2	13.16	2.34	12.30	6.04	3.08	3.91	4.26
2048^2	50.74	8.72	50.05	53.13	61.10	61.17	59.02
4096^2	203.35	34.60	200.56	220.02	252.92	251.84	251.44
8192^2	850.18	144.02	807.87	876.00	1003.01	1009.11	1000.43

Table 5: Execution times (in seconds) for different architectures and implementations.

4 Related Work

APIs such as OpenMP are able to tile stencil loops at runtime and execute the tiles in parallel [7]. Researchers have investigated the best combination of tiling strategies that optimizes both cache locality and parallelism, and even propose automatic tuning for tiling stencil computations on multicores [3], GPUs [9] and the Cell [2].

Stencil kernels on GPUs have recently gained attention by the scientist community. Listed in order of affinity with our work, we may select the following four contributions: Datta et al [3] tune a benchmark of 3D stencil kernels on GPUs and multicores, Christen et al. [2] consider a 7-point stencil kernel to be implemented on GPUs and the Cell BE, Amorim et al. [1] perform a comparison of the Jacobi method between a GPU parallelization using OpenGL and CUDA, and finally, Venkatasubramanian et at. [9] also implement the Jacobi method on GPUs and hybrid CPU/GPU systems.

Focusing on the work performed specifically on Jacobi method, Amorim et al. [1] use diagonal matrices and a different access pattern than ours to compare results against a CPU implementation on a quad-core AMD Phenom processor, obtaining a 78x speed-up factor. On the other hand, the work in [9] was developed in parallel to ours with a similar methodology. Our implementation sacrifices two idle threads on each half-warp to be rewarded on coalesced and conflicts-free accesses to memory banks, since memory bandwidth is more a bottleneck than the availability of computing cores within the GPU. Also, padding is more profitable in our coalesced case because it allows us to take advantage of remarkable improvements introduced in CUDA Compute Capabilities 1.3.

5 Summary and Conclusions

This paper explores CUDA on GPUs to optimize stencil computations using as benchmark the Jacobi method for solving Laplace's differential equation. Optimization paths are focused on data parallelism, threads deployment and the GPU memory hierarchy, with a clear influence of the stencil access pattern.

Experimental results show great success for our techniques on Teslas C870 and C1060 GPUs, achieving great scalability and good performance versus a quad-core Intel CPU. The speed-up factor for our set of GPU optimizations reaches 3-4x and the execution times defeat those of the CPU by a wide margin, also showing great scalability when moving towards a more sophisticated GPU architecture and/or more demanding problem sizes.

Streaming and arithmetic intensive kernels produce higher performance on the GPU to reach two orders of magnitude gain factors with respect to a multicore CPU. However, our kernel for Jacobi is bandwidth limited, preventing us from further optimizations. This behavior is also confirmed when comparing single to double precision performance, as the peak computational power is theoretically more than an order of magnitude higher for the single precision case and the execution time barely gets better by a factor of two.

References

[1] R. Amorim, G. Haase, M. Liebmann, and R. Weber dos Santos. Comparing CUDA and OpenGL Implementations for a Jacobi Iteration. Technical report, Graz University of Technology, December 2008.

[2] M. Christen, O. Schenk, E. Neufeld, P. Messmer, and H. Burkhart. Parallel Data-Locality Aware Stencil Computations on Modern Micro-Architectures. In *Proceedings IEEE Intl. Parallel and Distributed Processing Symposium*, Rome, Italy, May 2009.

[3] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. A. Patterson, J. Shalf, and K. Yelick. Stencil Computation Optimization and Auto-Tuning on State-of-the-art Multicore Architectures. In *Proceedings ACM/IEEE Supercomputing 2008*, pages 1–12, Austin, TX, USA, November 2008.

[4] J. Demmel. *Applied Numerical Linear Algebra*. SIAM, Philadelphia, PA, USA, 1997.

[5] GPGPU. General-Purpose Computation Using Graphics Hardware, 2009.

[6] B. Lester. *The Art of Parallel Programming*. Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[7] OpenMP. The OpenMP API, 2009.

[8] J. Owens, D. Luebke, Govindaraju, M. Harris, J. Kruger, A. Lefohn, and T. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Journal Computer Graphics Forum*, 26(1):80–113, Mar. 2007.

[9] S. Venkatasubramanian and R. W. Vuduc. Tuned and Wildly Asynchronous Stencil Kernels for Hybrid CPU/GPU Systems. In *Proceedings ACM Intl. Conference on Supercomputing*, New York, USA, June 2009.