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
{chema,jmgarcia}@ditec.um.es
[2] Computer Architecture Department, University of Malaga, Spain
ujaldon@uma.es

Abstract. We are witnessing the consolidation of the GPUs streaming paradigm in parallel computing. 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 [11]. Whereas hybrid clusters of SMPs where once the realm only of costly supercomputers, GPUs now turn nodes with this added complexity into a commodity. This fact has attracted GPUs to researchers in many fields [7], 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 normally 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 is rearranged to become the input to the next iteration, which prevents from parallelizing consecutive iterations.

K. Jónasson (Ed.): PARA 2010, Part I, LNCS 7133, pp. 173–183, 2012.

In order to exploit SIMD parallelism in general-purpose computing, both Nvidia and AMD have released software components which provide simpler access to GPU computing power than that realized by treating the GPU as a traditional graphics processor. CUDA [3] is Nvidia's solution as a simple block-based API for SIMD programming; AMD's solution is called Stream Computing. We choose CUDA to program the GPU for being more popular and providing more mechanisms to optimize general-purpose applications. Nevertheless, both models are expected to converge in OpenCL [8] as a higher level standard shared by a wide set of GPU models. Among them, we have some high-end graphics cards aimed specifically at the scientific General Purpose GPU (GPGPU) computing market: the Tesla products [12] are from NVIDIA, and Firestream [6] is AMD's product line.

Stencil computations are those in which each computing node in a multidimensional grid is updated with weighted values contributed by neighboring nodes. These neighbors 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 in CUDA: threads deployment, different uses of the shared memory, the effect of larger 2D stencils, the floating-point accuracy for single and double precision, and finally the scalability for our solution versus a multicore CPU. Our best version reaches a speed-up factor of 3-4x over a non-optimized GPU version, also showing great scalability when moving towards a more sophisticated GPU architecture and/or demanding problem sizes.

The rest of the paper is organized as follows. Section 2 explains the Jacobi method. Section 3 briefly introduces the specifics of the GPU programming with CUDA. Section 4 outlines our CUDA implementation, exposes the execution times and analyzes the results. Finally, Section 5 reviews some related work and Section 6 concludes.

2 The Jacobi Method

Jacobi [9] is a popular algorithm for solving Laplace's differential equation on a square domain, regularly discretized [5]. 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 at all of the five stencil nodes (see Figure 1).

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 because the computation at each particle

```
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]; }
```

Fig. 1. Jacobi's solver pseudocode

Table 1. GPU features for the Tesla cards used during our experimental analysis

Hardware feature	**Tesla C870**	**Tesla C1060**
Compute Capabilities	1.0	1.3
Number of streaming processor	128	240
Frequency of streaming processors	1.35 GHz	1.30 GHz
Global memory size and type	1.5 Gbytes GDDR3	4 Gbytes GDDR3
Global memory width and speed	384 bits @ 2x800 MHz	512 bits @ 2x800 MHz
Global memory bandwidth	76.8 Gbytes/sc.	102 Gbytes/sc.

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, the Laplace equation, once discretized, leads to our Jacobi kernel. This 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 $O(k \cdot N^2)$.

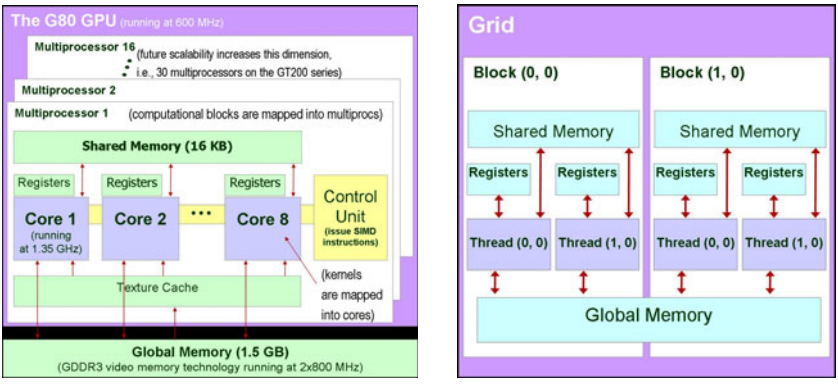
3 CUDA

CUDA (Compute Unified Device Architecture) [3] is a programming interface and set of supported hardware to enable general purpose computation on Nvidia GPUs and leverage special hardware features not visible to more traditional graphics-based GPU programming, such as small cache memories, explicit massive parallelism and lightweight context switch between threads.

The Tesla C870 and C1060 GPUs are respectively based on the G80 and GT200 architectures, whose major features are depicted in Table 1.

Each GPU multiprocessor can run a variable number of threads, and the local resources are shared among them. In any given cycle, each core in a multiprocessor executes the same instruction on different data based on its `threadId`, and communication between multiprocessors is performed through global memory (see Figure 2).

In the CUDA programming model, a program is decomposed into **blocks**, groups of threads runnning in parallel within a single multiprocessor where they share registers, memory and other multiprocessor's resources (see Figure 2.a).



(a) Hardware interface (b) Programming model

Fig. 2. CUDA highlights for the G80 core used in the Tesla C870 board

Table 2. Major hardware and software limitations with CUDA. Constraints are listed for the G80 and GT200 GPUs, the ones inside our Teslas C870 and C1060 boards.

	G80	**GT200**
CUDA Compute Capabilities	1.0 and 1.1	1.2 and 1.3
Multiprocessors per GPU	16	30
Processors / Multiprocessor	8	8
32-bit registers / Multiprocessor	8192	16384
Shared Memory / Multiprocessor	16 KB	16 KB.
Threads / Warp	32	32
Thread Blocks / Multiprocessor	8	8
Threads / Block	512	512
Threads / Multiprocessor	768	1024

A **kernel** is a code function compiled to the instruction set of the device and executed by all of its threads. Threads run on different processors of the multiprocessors sharing the same executable and global address space, though they may not follow exactly the same path of execution. A kernel is organized into a **grid** as a set of **thread blocks** explicitly defined by the application developer and executed on a single multiprocessor.

Threads placed in different blocks from the same grid cannot communicate. This tradeoff between parallelism and thread resources must be wisely solved by the programmer to maximize execution efficiency on a certain architecture given its limitations. These limitations are listed for the case of our Tesla boards in Table 2.

3.1 Memory Optimizations

During our CUDA implementation, attention must be paid to how threads access the 16 banks of shared memory, since only when the data resides in different banks can all of the available ALU bandwidth truly be used. Each bank only

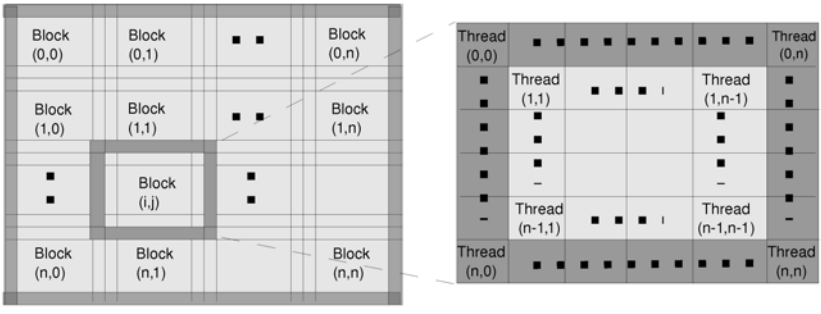


Fig. 3. Threads deployment for the CUDA parallelization strategy

Table 3. Execution times (in seconds) for our Jacobi baseline implementation

Matrix size	(threads deployment per 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 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

supports one memory access at a time; simultaneous memory bank accesses are serialized, stalling the rest of the multiprocessor's running threads until their operands arrive. The use of shared memory is explicit within a thread, which allows the developer to solve bank conflicts wisely.

Another critical issue related to memory performance is *data coalescing*. A coalesced access involves a contiguous region of global memory where the starting address must be a multiple of region size and the k^{th} thread in a half-warp must access the k^{th} element in a block being read. This way, the hardware can serve completely two coalesced accesses per clock cycle, which maximizes memory bandwidth. It is programmer's responsability to organize memory accesses in such a way.

4 Implementation

4.1 Optimal Threads Deployment

Figure 3 shows the threads deployment for the parallelization of the Jacobi method using CUDA. 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 preliminary survey. Table 3 shows that 16x16 constitutes the optimal number of threads per block, with a penalty around 5-10% in terms of the execution time for the other three cases. All remaining squared alternatives for the matrix of threads led to worse results.

4.2 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 neighbors 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, because 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. This leads to a more homogeneous behavior of threads, which is exploited to omit certain __syncthreads() calls at block level, and also enables data prefetching as a positive side-effect. Nevertheless, these enhancements behave similarly on CCC 1.0 and CCC 1.3, and consequently are translated into minor improvements in the overall execution time.
3. The third optimization reduces the relative amount of shared memory used by each thread so that a block of threads can work with a greater data area. In other words, the block uses the same amount of shared memory, say 16x16, but with those data now is capable of managing a tile of 16x32 data in two iterations. Doing so, we can maximize the parallelism allowed on each CUDA platform. 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 amount of shared memory used by each block was reduced until 4120 bytes, so that we can assign three blocks of 256 threads to each multiprocessor. In the second case, we reduced this size even more until we could assign four blocks of 256 threads, which increases parallelism leading to slightly better results.

Table 4.a shows the execution times for all these versions on a Tesla C870 and Table 4.b does the same for the Tesla C1060 GPU. An average speed-up factor of 3.5x is roughly attained.

Table 4. 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.

Matrix	Baseline:	Optimiz. 1	Optimizs. 1+2	Optimizs. 1+2+3
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

Matrix	Baseline:	Optimiz. 1	Optimizs. 1+2	Optimizs. 1+2+3
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

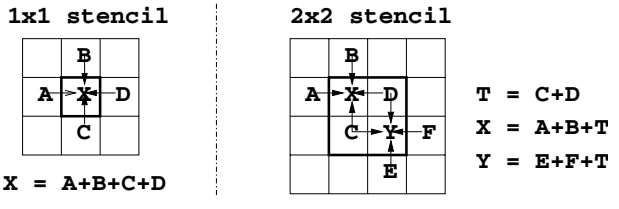


Fig. 4. Increasing the stencil size: some redundant operations may be saved

4.3 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 4), 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 5 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. The execution is slowed down 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: using a 1x1 stencil we require 341x341 calls with thread blocks, whereas using a 2x2 stencil, we just need 157x157 calls.

Table 5. 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.

Threads deployment	Stencil size 1x1	Stencil size 2x2	Slowdown factor
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 6. Execution times (in seconds) and empirical streaming bandwidth (in GB/s.) 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, where theoretical peak bandwidth is 102 GB/s. We run out of memory for the 8192^2 case on double precision. Threads deployment is 16x16 for all cases.

Matrix size	Single Precision		Double Precision		Slowdown factor
	Exec. time	Bandwidth	Exec. time	Bandwidth	
1024^2	0.90	104.25 GB/sec	1.70	56.69 GB/sec	89%
2048^2	3.36	114.49 GB/sec	6.55	14.70 GB/sec	95%
4096^2	13.41	117.00 GB/sec	26.33	3.66 GB/sec	96%

4.4 Floating-Point Accuracy and Performance

We now evaluate floating-point performance by comparing the GPU power on single and double precision versions of our optimal Jacobi implementation. According to Table 1, peak performance on the Tesla C1060 is 933 GFLOPS in single precision and 78 GFLOPS in double precision. However, Table 6 shows that times roughly 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.

Table 6 also includes bandwidth data attained by our implementation. In certain cases, we reached values exceeding the theoretical peak bandwidth provided by the global memory (GDDR3 video RAM in our GPU), which can justify the presence of small cache areas in the underlying architecture, a feature which is often deliberately undisclosed by hardware vendors. Nevertheless, we believe these bandwidth numbers reflect a solid validation of outstanding performance for the kernels developed throughout this work.

4.5 GPU versus CPU Multi-core Performance

In our final experiment, we want to compare the many-core GPU with a multi-core CPU in terms of scalability, parallel performance and the influence of the

Table 7. Execution times (in seconds) for different architectures and implementations

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	3.88	0.63	12.30	6.04	3.08	3.91	4.26
2048^2	14.71	2.42	50.05	53.13	61.10	61.17	59.02
4096^2	57.45	9.66	200.56	220.02	252.92	251.84	251.44
8192^2	241.81	40.29	807.87	876.00	1003.01	1009.11	1000.43

memory hierarchy. Table 7 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 than 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.

5 Related Work

APIs such as OpenMP are able to tile stencil loops at run-time and execute the tiles in parallel [10]. 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 [4], GPUs [13] and the Cell [2]. Those techniques are usually based on the concept of ghost zones, which enlarge the tile with a perimeter overlapping neighboring tiles by multiple *halo* regions to reduce communications by replicating computation.

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 [4] 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. [13] 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 [13] was developed in parallel to ours with a similar methodology. Our implementation sacrifices two idle threads on each half-warp by replicating data placed at block boundaries, which helps us to succeed on a more homogeneous access to the device memory. This is rewarded on conflicts-free accesses to memory banks and particularly on coalesced accesses, overall in CUDA Compute Capabilities 1.3, where conditions for non-coalesced accessed were widely relaxed.

In all CUDA implementations, a crucial parameter for attaining the best performance is the thread block size. In [13], a 1.7x slowdown factor is reported when moving from a 64x8 to a 16x8 block size on a Tesla C1060 GPU using a 8x8 thread block. Though we agree on how sensitive the execution time is to the thread deployment, our best performance is achieved on a 16x16 thread block, with a clear benefit on lighter threads. This way, our key guidelines to the optimal Jacobi implementation were to reduce to the minimum (1) the number of target elements computed by each thread, and (2) the number of tiles computed by each thread block.

6 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.

Acknowledgments. This work has been jointly supported by the Fundación Séneca (Agencia Regional de Ciencia y Tecnología, Región de Murcia) under grant 00001/CS/2007, by the Junta de Andaluciía under project P06-TIC-02109, and also by the Spanish MEC and European Commission FEDER funds under grants Consolider Ingenio-2010 CSD2006-00046 and TIN2009-14475-C04-02.

References

1. Amorim, R., Haase, G., Liebmann, M., Weber dos Santos, R.: Comparing CUDA and OpenGL Implementations for a Jacobi Iteration. Technical Report, Graz University of Technology (December 2008)
2. Christen, M., Schenk, O., Neufeld, E., Messmer, P., Burkhart, H.: Parallel Data-Locality Aware Stencil Computations on Modern Micro-Architectures. In: Procs. IEEE Intl. Parallel and Distributed Processing Symposium, Rome (May 2009)
3. CUDA: `http://developer.nvidia.com/object/cuda.html` (accessed September 15, 2010)
4. Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D.A., Shalf, J., Yelick, K.: Stencil Computation Optimization and Auto-Tuning on State-of-the-art Multicore Architectures. In: Proceedings ACM/IEEE Supercomputing 2008, Austin, TX, USA, pp. 1–12 (November 2008)
5. Demmel, J.: Applied Numerical Linear Algebra. SIAM, Philadelphia (1997)
6. Firestream: AMD Stream Computing, `http://www.amd.com/us/products/workstation/firestream/Pages/firestream.aspx` (accessed September 15, 2010)
7. GPGPU: General-Purpose Computation Using Graphics Hardware (2010), `http://www.gpgpu.org`
8. The Khronos Group: The OpenCL Core API Specification. Headers and documentation, `http://www.khronos.org/registry/cl` (accessed September 15, 2010)
9. Lester, B.: The Art of Parallel Programming. Prentice Hall, Engl. Cliffs (1993)
10. OpenMP: The OpenMP API (2010), `http://www.openmp.org`
11. Owens, J., Luebke, D., Govindaraju, Harris, M., Kruger, J., Lefohn, A., Purcell, T.: A Survey of General-Purpose Computation on Graphics Hardware. Journal Computer Graphics Forum 26(1), 80–113 (2007)
12. Tesla: Nvidia Tesla GPU computing solutions for HPC, `http://www.nvidia.com/object/personal_supercomputing.html` (accessed September 15, 2010)
13. Venkatasubramanian, S., Vuduc, R.W.: Tuned and Wildly Asynchronous Stencil Kernels for Hybrid CPU/GPU Systems. In: Proceedings ACM Intl. Conference on Supercomputing, New York, USA (June 2009)