Parallel SOR for Solving the Convection Diffusion Equation Using GPUs with CUDA

Yiannis Cotronis, Elias Konstantinidis,
Maria A. Louka, and Nikolaos M. Missirlis

Department of Informatics and Telecommunications,
University of Athens,
Panepistimiopolis, 15784, Athens, Greece
{cotronis,ekondis,mlouka,nmis}@di.uoa.gr

Abstract. In this paper we study a parallel form of the SOR method for the numerical solution of the Convection Diffusion equation suitable for GPUs using CUDA. To exploit the parallelism offered by GPUs we consider the fine grain parallelism model. This is achieved by considering the local relaxation version of SOR. More specifically, we use SOR with red black ordering with two sets of parameters ω_{ij} and $\omega_{ij}^{'}$. The parameter ω_{ij} is associated with each red (i+j even) grid point (ij), whereas the parameter $\omega_{ij}^{'}$ is associated with each black (i+j odd) grid point (ij). The use of a parameter for each grid point avoids the global communication required in the adaptive determination of the best value of ω and also increases the convergence rate of the SOR method [3]. We present our strategy and the results of our effort to exploit the computational capabilities of GPUs under the CUDA environment. Additionally, a program for the CPU was developed as a performance reference. Significant performance improvement was achieved with the three developed GPU kernel variations which proved to have different pros and cons.

Keywords: Iterative methods, SOR, R/B SOR, GPU computing, CUDA.
Subject classification: AMS(MOS), 65F10, 65N20, CR:5.13.

1 Introduction

Traditionally, conventional processors have been used to solve computational problems. Modern graphics processors (GPUs) have become coprocessors with significantly more computational power than general purpose processors. Their large computational potential has turned them to a special challenge for solving general-purpose problems with large computational burden. Thus, application programming environments have been developed like the proprietary CUDA (Compute Unified Development Architecture) by NVidia [16,12] and the OpenCL (Open Computing Language) [20] which is supported by many hardware vendors, including NVidia.

CUDA environment is rapidly evolving and a constantly increasing number of researchers is adopting it in order to exploit GPU capabilities. It provides an elegant way for writing GPU parallel programs, by using a kind of extended C language, without involving other graphics APIs. In this paper we use GPUs for the numerical solution of Partial Differential equations. In particular, we consider the solution of the second order convection diffusion equation

$$\varDelta u - f(x,y)\frac{\partial u}{\partial x} - g(x,y)\frac{\partial u}{\partial y} = 0 \tag{1}$$

on a domain $\varOmega = \{(x,y)\}|0 \leq x \leq 1,\ 0 \leq y \leq 1\}$, where $u = u(x,y)$ is prescribed on the boundary $\partial\varOmega$. The discretization of (1) on a rectangular grid $M_1 \times M_2 = N$ unknowns within $\varOmega$ leads to

$$\begin{gathered} u_{ij} = \ell_{ij}u_{i-1,j} + r_{ij}u_{i+1,j} + t_{ij}u_{i,j+1} + b_{ij}u_{i,j-1}, \\ i = 1,2,\dots,M_1\ ,\ j = 1,2,\dots,M_2 \end{gathered} \tag{2}$$

with

$$\begin{gathered} \ell_{ij} = \frac{k^2}{2(k^2+h^2)}(1 + \frac{1}{2}hf_{ij})\ ,\ r_{ij} = \frac{k^2}{2(k^2+h^2)}(1 - \frac{1}{2}hf_{ij}) \\ t_{ij} = \frac{h^2}{2(k^2+h^2)}(1 - \frac{1}{2}kg_{ij})\ ,\ b_{ij} = \frac{h^2}{2(k^2+h^2)}(1 + \frac{1}{2}kg_{ij}), \end{gathered} \tag{3}$$

where $h = 1/(M_1+1)$, $k = 1/(M_2+1)$, $f_{ij} = f(ih,jk)$ and $g_{ij} = g(ih,jk)$. For a particular ordering of the grid points (2) yield a large, sparse, linear system of equations of order N of the form

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

The Successive Overrelaxation (SOR) iterative method, which is given by the form

$$u_{ij}^{(n+1)} = (1-\omega)u_{i,j}^{(n)} + \omega(\ell_{ij}u_{i-1,j}^{(n+1)} + r_{ij}u_{i+1,j}^{(n)} + t_{ij}u_{i,j+1}^{(n)} + b_{ij}u_{i,j-1}^{(n+1)}) \tag{5}$$

is an important solver for large linear systems [14], [15]. It is also a robust smoother as well as an efficient solver of the coarsest grid equations in the multi-grid method. However, the SOR method is essentially sequential in its original form. Several parallel versions of the SOR method have been studied by coloring the grid points [1], [13].

In order to use a parallel form of the SOR method with fine grain parallelism we have to color the grid points red-black [1], [13] so that sets of points of the same color can be computed in parallel. However, the parameter ω which accelerates the rate of convergence of SOR is computed adaptively in terms of $u^{(n+1)}$ and $u^{(n)}$ [7]. This computation requires global communication between the processors for each iteration. To overcome this problem local relaxation methods are used [4], [5], [11]. In these methods each point in the grid has its own relaxation parameter ω_{ij} which is determined in terms of the local coefficients of the PDE. In [2], [4], [5], the local SOR (LSOR) with different formulas for the optimum values of the relaxation parameters was studied numerically and compared with the classic SOR method for the 5-point stencil. It was found that LSOR possesses better convergence rate than SOR using only local communication. However, the first theoretical results about the convergence of LSOR were presented in [11] under the assumption that the coefficient matrix is symmetric and positive definite. Following a similar approach but using two different sets of parameters ω_{ij} and ω'_{ij} for the red and black points, respectively, it was proved in [3] that the local Modified SOR method (LMSOR) possesses a better rate of convergence than LSOR for the 5-point stencil. This comparison was carried out in case the eigenvalues of the Jacobi matrix possesses either real (real case) or imaginary (imaginary case) eigenvalues.

The SOR method has been implemented on GPUs as applied to medical analysis [6] as well as to computational fluid dynamics [9] problems.

Our contribution is to explore the LMSOR method for the 5-point stencil exploiting the computational capabilities of GPUs under the CUDA environment. In our study we used three different techniques to find the one that best exploits the capabilities of the GPU.

The remainder of the paper is organized as follows. In section 2 we present a general description of the LMSOR method. In section 3 we present its implementation in GPUs, in section 4 we present our performance results and finally, in section 5, we state our remarks and conclusions.

2 The Local Modified SOR Method

The LSOR method was introduced by Ehrlich [4], [5] and Botta and Veldman [2] in an attempt to further increase the rate of convergence of SOR. The idea is based on letting the relaxation factor ω vary from equation to equation. This means that each equation of (2) has its own relaxation parameter denoted by ω_{ij}. Kuo et. al [11] combined LSOR with red black ordering and showed that is suitable for parallel implementation on mesh connected processor arrays. In [3] we generalized LSOR by letting two different sets of parameters $\omega_{ij}, \omega'_{ij}$ to be used for the red ($i+j$ even) and black ($i+j$ odd) points, respectively. An application of our method to (2) can be written as follows:

$$u_{ij}^{(n+1)} = (1-\omega_{ij})u_{ij}^{(n)} + \omega_{ij} J_{ij} u_{ij}^{(n)}, \quad \text{red points} \tag{6}$$

$$u_{ij}^{(n+1)} = (1-\omega'_{ij})u_{ij}^{(n)} + \omega'_{ij} J_{ij} u_{ij}^{(n+1)}, \quad \text{black points} \tag{7}$$

where

$$J_{ij} u_{ij}^{(n)} = l_{ij} u_{i-1,j}^{(n)} + r_{ij} u_{i+1,j}^{(n)} + t_{ij} u_{i,j+1}^{(n)} + b_{ij} u_{i,j-1}^{(n)} \tag{8}$$

and J_{ij} is called the local Jacobi operator. The parameters $\omega_{ij}, \omega'_{ij}$ are called local relaxation parameters and (6)–(8) will be referred to as the local Modified SOR (LMSOR) method. Note that if $\omega_{ij} = \omega'_{ij}$, then (6), (7) reduce to the LSOR method studied in [11]. Moreover, if $\omega_{ij} = \omega'_{ij} = \omega$ (6), (7) degenerate into the classical SOR method with red black ordering. Using Fourier analysis, Boukas and Missirlis [3] proved that the optimum values of the local relaxation parameters $\omega_{1,i,j}$ and $\omega_{2,i,j}$ for the LMSOR method in case the eigenvalues μ_{ij} of the local Jacobi operator J_{ij} are all real or all imaginary are the following.

Case 1: μ_{ij} are real. This case applies when $\ell_{ij}r_{ij} \geq 0$ and $t_{ij}b_{ij} \geq 0$. The optimum values of the LMSOR parameters are given by

$$\omega_{1,i,j} = \frac{2}{1 - \overline{\mu}_{ij}\underline{\mu}_{ij} + \sqrt{(1-\overline{\mu}_{ij}^2)(1-\underline{\mu}_{ij}^2)}} \quad \text{and} \quad \omega_{2,i,j} = \frac{2}{1 + \overline{\mu}_{ij}\underline{\mu}_{ij} + \sqrt{(1-\overline{\mu}_{ij}^2)(1-\underline{\mu}_{ij}^2)}} \tag{9}$$

where

$$\overline{\mu}_{ij} = 2\left(\sqrt{\ell_{ij}r_{ij}}\cos\pi h + \sqrt{t_{ij}b_{ij}}\cos\pi k\right) \tag{10}$$

and

$$\underline{\mu}_{ij} = 2\left(\sqrt{\ell_{ij}r_{ij}}\cos\frac{\pi(1-h)}{2} + \sqrt{t_{ij}b_{ij}}\cos\frac{\pi(1-k)}{2}\right). \tag{11}$$

Note that μ_{ij} is the spectral radius of the local Jacobi operator J_{ij} where

$$\mu_{ij} = 2\left(\sqrt{\ell_{ij}r_{ij}}\cos\frac{k_1\pi}{M_1+1} + \sqrt{t_{ij}b_{ij}}\cos\frac{k_2\pi}{M_2+1}\right), \tag{12}$$

with $k_1 = 1, 2, \ldots, M_1$, $k_2 = 1, 2, \ldots, M_2$, for periodic boundary conditions.

Case 2: μ_{ij} are imaginary. This case applies when $\ell_{ij}r_{ij} \leq 0$ and $t_{ij}b_{ij} \leq 0$. The optimum values of the LMSOR parameters are given by

$$\omega_{1,i,j} = \frac{2}{1 - \overline{\mu}_{ij}\underline{\mu}_{ij} + \sqrt{(1+\overline{\mu}_{ij}^2)(1+\underline{\mu}_{ij}^2)}} \quad \text{and} \quad \omega_{2,i,j} = \frac{2}{1 + \overline{\mu}_{ij}\underline{\mu}_{ij} + \sqrt{(1+\overline{\mu}_{ij}^2)(1+\underline{\mu}_{ij}^2)}} \tag{13}$$

where $\overline{\mu}_{ij}$ and $\underline{\mu}_{ij}$ are computed by (10) and (11), respectively.

3 Parallel Implementation

Implementations for the LMSOR method were developed for both the CPU and GPU. The CPU version is a single threaded program, used as a performance reference for our experiments. In contrast, the GPU version is a massively parallel program. The speedup that is observed between the sequential CPU program and the parallel GPU program is a sequential versus parallel program comparison, although not in the strict sense due to the GPU architectural differences.

As the solution of the Laplace equation using the red/black SOR method is memory bound [10], this problem can also be characterized as memory bound. Inspecting the computations by (6), (7) and (8) we note that, in case of good cache behavior, each element needs to access roughly 8 elements (accesses to $u_{i,j}$ count as 3, 2 reads and 1 write) per iteration (either red or black). The same computation reveals also that 11 floating point operations are needed per element. Thus, the ratio of floating point operations per accessed elements is 11/8, and considering the use of double precision arithmetic, the ratio of floating point operations per byte accesses is $11/(8 \times 8) \approx 0.17$. This ratio is quite low as the GPUs are able to handle much more compute operations than memory access operations [8]. For instance, the NVidia GTX480 has a double precision peak performance of 168 GFlops and memory throughput of 177 GB/sec. Thus, a balanced algorithm should perform at least ≈ 1 double precision floating point operation per byte accessed. It should be noted that double precision operations were applied in all developed programs of this work.

In the implementation of the LMSOR method, the parameters ω_{ij} and ω'_{ij} are precomputed on the CPU for two of the three developed kernels.

As our program implements a red/black ordering, it is beneficial to apply reordering by color strategy into separate matrices in order to optimize performance by coalescing [10], [17]. Points in a mesh are split into two different matrices, one for the red points and one for the black. This strategy can improve bandwidth utilization by improving locality and coalescing of memory accesses and mostly, by utilizing all points contained in a memory segment, which is not possible with a natural interleaved red/black ordering.

Moreover, our program utilizes 6 matrices during the computation procedure (u, ω, l, r, t and b) as formulae (6), (7) and (8) indicate, all of which feature a red/black ordering. In contrast, the solution of the Laplace equation with R/B SOR requires accessing on a single matrix [10]. As the reordering strategy can be applied on every red-black ordered matrix it is possible to apply it on all 6 matrices. This factor raises the importance of the use of point reordering by color strategy.

In order to alleviate the high memory bandwidth requirements set by the program, an alternative approach will be used. Some read-only matrices, having their elements computed during the program initialization, can be eliminated by replacing accesses on them with computations in the GPU kernel. As the GPU has very high instruction throughput capability this trade-off can be beneficial. In summary, LMSOR was implemented in three variations as three different kernels. Each variation differs by the amount of redundant computations it performs iteratively. All kernels employ the reordering by color strategy as it is expected to be beneficial. These kernels are:

Kernel #1 - No Redundant Computations. All values required in (6), (7) and (8) reside in matrices situated in the GPU device memory. Beyond employing the reordering by color strategy, this kernel is the natural outcome implementation as no extra computations are performed. Thus, the 6 aforementioned matrices are required in this scheme and about 8 element accesses per computed element. As previously shown, the ratio of floating point operations per byte accessed is 0.17, which is particularly low.

Kernel #2 - Redundant Computations of $l_{i,j}$, $r_{i,j}$, $t_{i,j}$, $b_{i,j}$. The values of the two matrices $f_{i,j}$ and $g_{i,j}$ multiplied by h, are precomputed and stored in two matrices in the device memory. Thus, instead of 4 matrices for $l_{i,j}$, $r_{i,j}$, $t_{i,j}$ and $b_{i,j}$ we just need to keep 2 matrices only in device memory. Memory requirements are lower since only 4 matrices are required to reside in device memory (for $u_{i,j}$, $\omega_{i,j}$, $f_{i,j}$ and $g_{i,j}$). However, it comes at a cost of extra operations needed to recompute the required terms for the formula on every iteration. In this case, each element requires 6 accesses and at least $11{+}4 = 15$ floating point operations, as formula (3) indicates. Now, the ratio is about $15/(6\times 8) = 0.31$ flops per byte, which is a more balanced ratio but still less than 1.0.

Kernel #3 - Redundant Computations of All Terms. In this implementation, recomputation is applied to the extreme point that all terms, excluding $u_{i,j}$, are recomputed in flight. The type of f and g functions is passed as a parameter to the kernel and all required terms are recomputed on every iteration. In this case only 3 accesses per computed element are required. The required flops are dependent on the selected $f(x, y)$ and $g(x, y)$ functions. A rough estimate is that at least $15 + 30 = 45$ flops are required plus the extra flops for the computation of $f(x, y)$ and $g(x, y)$. An approximation of the least ratio value is $45/(3 \times 8) \approx 1.9$, which clearly exceeds 1. Thus, this kernel is compute bound, as opposed to the previous kernels.

During computation only $u_{i,j}$ terms are accessed from memory and all other terms are recomputed as required. Thus, this variation has the least memory requirements of all kernels, as it requires only 1 matrix residing in device memory. The performance of the GPU version relies on the global memory cache present on Fermi GPU devices. As it has been shown [10] the global memory cache can offer the potential of high performance without the need to utilize special memory types (i.e. shared memory or texture memory). Additionally, it accomodates previously non-coalesced memory accesses with spacial locality. Therefore, the application is not expected to run efficiently on older hardware, i.e. GT-200 based GPUs. On such architectures, an alternative approach should have been chosen utilizing texture memory or shared memory of the device.

It should be noted that all implementations perform convergence checking on every iteration which raises the execution overhead. Convergence checking in the GPU kernel is implemented as a reduction of all computed maximum values. On a production environment convergence checking should be avoided, at least on most iterations, in order to attain peak performance.

The CPU version is a fairly straightforward implementation without employing any sophisticated access patterns. Elements are processed sequentially in rows and no cache blocking has been employed.

4 Performance Results

In order to test our theoretical results we considered the numerical solution of (1) with $u = 0$ on the boundary of the unit square. The initial vector was chosen as $u^{(0)}(x, y) = xy(1 - x)(1 - y)$. The solution of the problem above is zero. For the purpose of comparison we considered the application of LMSOR method with red black ordering, on CPU and GPU. In all cases the iterative process was terminated when the criterion $||u^{(n)}||_\infty \leq 10^{-6}$ was satisfied. Various functions for the coefficients $f(x, y)$ and $g(x, y)$ were chosen such that the eigenvalues μ_{ij} to be either real or imaginary. The type of eigenvalues for each case is indicated by the tuple (# real, # imaginary) in the second row of each table. The coefficients used in each problem are:

1. $f(x, y) = Re(2x - 10)^3$, $g(x, y) = Re(2y - 10)^3$
2. $f(x, y) = Re(2x - 10)$, $g(x, y) = Re(2y - 10)$
3. $f(x, y) = g(x, y) = Re \cdot 10^4$

where the Reynold operator $Re = 10^m,\ m = 0, 1, 2, 3$ and 4.

All experiments were performed on a Linux environment. The CPU implementation was compiled with GCC version 4.4.4 on a 64bit environment, with all essential optimization flags enabled (-O2 -fomit-frame-pointer -ftree-vectorize -msse2 -msse -funroll-loops -fassociative-math -fno-signed-zeros -fno-trapping-math -fno-signaling-nans). The GPU implementation was compiled using CUDA Toolkit version 4.1 and GCC version 4.1.2 on a 64bit environment. The graphics driver version was 295.53. The parameter "–use_fast_math" had been used.

The hardware used for the experimental runs was an AMD Opteron 6180 SE (2.5GHz), for the CPU executions. For the GPU executions, a Nvidia GTX-480 and a Tesla C2050 [19] were used. Both GPUs are Fermi architecture based, featuring global memory cache which is essential for the performance of our kernel.

Three different series of experimental runs were performed, each investigating the GPU and CPU implementations from a different aspect. The first series of runs were performed in order to determine the most efficient out of the 3 developed kernels applying the LMSOR method. The second series of runs was performed in order to compare the GPU version with the CPU version, for the three problems, in terms of performance, on various *Re* values. The fluctuation of *Re* values causes a varying number of required iterations to meet convergence. The last series of runs was performed to measure the performance of the GPU kernel and the CPU on one specific problem, on a wider range of mesh sizes where the CPU version execution is heavily time-consuming.

In the results that follow, two different time measurements were carried out. The first, referred as *computation time*, is the net computation time, without extra overheads like the PCI-Express data transfer time overhead and, in case of GPU kernels, the element reordering time overhead. The second, referred as *execution time*, includes all the aforementioned overhead times. The function used to measure time is the *gettimeofday()* function, which is available on Linux platform.

4.1 Three Kernel Comparison

All kernels, of both methods, were executed in solving the three aforementioned problems, on mesh size $h = k = \frac{1}{\sqrt{N}+1}$ where $\sqrt{N} = M_1 = M_2 = \{402, 2002\}$. The GPU used in this experiment was the GTX480. The results of the executions are depicted on table 1. Large matrices are more important, as the GPUs are optimized for massive parallelism and therefore suited for large array processing. Thus, it is sensible to focus on the case where $\sqrt{N} = 2002$.

Table 1. Kernel comparison in LMSOR execution on GTX480, for $\sqrt{N} = \{402, 2002\}$

		$\sqrt{N} = 402$			$\sqrt{N} = 2002$		
f,g	Experimental results	#1	#2	#3	#1	#2	#3
1	(R,I)	(0,161604)			(0,4008004)		
	Iterations	412	412	412	1998	1998	1998
	Computation time (secs)	0.0561	0.0503	0.1489	3.6108	2.9294	13.7794
	Total execution time (secs)	0.0613	0.0541	0.1507	3.6636	2.9711	13.7940
	Comp. time/iteration (msecs)	0.1361	0.1220	0.3614	1.8072	1.4662	6.8966
2	(R,I)	(161604,0)			(4008004,0)		
	Iterations	554	554	554	2704	2704	2704
	Computation time (secs)	0.0752	0.0670	0.1889	4.9431	4.0572	17.4341
	Total execution time (secs)	0.0812	0.0702	0.1905	4.9960	4.0987	17.4497
	Comp. time/iteration (msecs)	0.1358	0.1209	0.3409	1.8281	1.5004	6.4475
3	(R,I)	(0,161604)			(0,4008004)		
	Iterations	1015	1015	1015	2018	2018	2018
	Computation time (secs)	0.1372	0.1223	0.3429	3.6871	3.0273	12.9215
	Total execution time (secs)	0.1418	0.1255	0.3442	3.7400	3.0668	12.9357
	Comp. time/iteration (msecs)	0.1352	0.1204	0.3378	1.8271	1.5002	6.4031

As it is obvious from the results, kernel #3 presents the worst performance. Kernel #2 seems to be the best performing of all. Although, it executes more operations per computed element, it actually performs better ($\approx 22\%$) than the first one, revealing the memory throughput bottleneck in this program. On the C2050 the improvement of kernel #2 was even more notable ($25 - 39\%$) due to its higher double precision operation throughput.

The advantage of kernel #3 is its limited memory access requirements. Kernel #1 makes use of 6 $\sqrt{N} \times \sqrt{N}$ matrices, one for each mesh. Kernel #2 makes use of 4 matrices of the same order and kernel #3 makes use of just 1 matrix of the same order. This makes it suitable for solving a large problem when memory size is a critical limitation.

Due to the different memory access requirements of the 3 kernels, inspecting the effective bandwidth can lead to misleading conclusions about the performance of each kernel. As can be seen on table 2, some profiling data were captured during one iteration of computation of red elements, for $\sqrt{N} = 4002$. Bytes accessed by kernel were extrapolated by using the first two performance counters. For kernel #1 the achieved effective bandwidth was estimated to be almost 148GB/sec, computing about 2250 elements/sec. Kernel #2 achieved calculating near 2800 elements by utilizing about 10GB/sec less bandwidth. In contrast, kernel #3 suffers by low occupancy and instruction execution pressure.

Table 2. Profiling on one iteration of red elements calculation on the GTX480, for $\sqrt{N} = 4002$

	kernel #1	kernel #2	kernel #3
fb_subp0_read_sectors	7204214	5102010	2013049
fb_subp0_write_sectors	1001454	1001464	1102356
gputime	3549.952	2871.936	12671.392
registers/thread	25	25	63
occupancy	0.667	0.667	0.333
Bytes accessed (extrapolated)	525162752	390622336	199385920
Bandwidth (GB/sec)	147.94	136.01	15.74
MegaElements/sec	2253.55	2785.58	631.34

Each kernel is characterized by different memory bandwidth requirements and thus, it cannot be used as a direct comparison measure. Thus, pure bandwidth does not expose the actual performance of these kernels.

4.2 CPU - GPU Comparison

In this series of executions the GPU kernel #2, and the CPU program were compared, for both methods, in executions for various *Re* values. Matrix order $\sqrt{N}$ was kept constant ($\sqrt{N} = 1002$) and the program was executed for *Re*=\{*1000.0, 10000.0, 100000.0*\}. Results are depicted in table 3. It is worth to note that the GPU version is constantly achieving an over ×50 speed-up over the single threaded CPU version. The GPU shows a stable performance behavior by computing elements at a rate of less than half a millisecond per iteration.

Table 3. Kernel comparison in LMSOR execution on GTX480, for $\sqrt{N} = 1002$, for various values of *Re*, for the three problems, * indicates no convergence after 20000 iterations

f,g	Experimental results	$Re = 1000.0$		$Re = 10000.0$		$Re = 100000.0$	
		CPU	GPU	CPU	GPU	CPU	GPU
1	(R,I)	(0,1004004)		(0,1004004)		(0,1004004)	
	Iterations	2620	2620	5394	5394	6243	6243
	Computation time (secs)	56.7055	1.1146	118.1573	2.2959	133.6569	2.6551
	Total execution time (secs)	56.7055	1.1269	118.1573	2.3082	133.6569	2.6674
	Comp. time/iteration (msecs)	21.6433	0.4254	21.9053	0.4256	21.4091	0.4253
	Computation speedup	1.0000	50.8773	1.0000	51.4650	1.0000	50.3394
2	(R,I)	(0,1004004)		(0,1004004)		(0,1004004)	
	Iterations	1003	1003	1112	1112	3170	3170
	Computation time (secs)	21.0700	0.4266	23.9960	0.4728	69.4001	1.3464
	Total execution time (secs)	21.0700	0.4369	23.9960	0.4831	69.4001	1.3568
	Comp. time/iteration (msecs)	21.0070	0.4253	21.5791	0.4252	21.8928	0.4247
	Computation speedup	1.0000	49.3880	1.0000	50.7524	1.0000	51.5448
3	(R,I)	(0,1004004)		(0,1004004)		(0,1004004)	
	Iterations	5514	5514	6271	6271	7034	7034
	Computation time (secs)	118.6613	2.3459	135.8946	2.6648	154.8432	2.9918
	Total execution time (secs)	118.6613	2.3562	135.8946	2.6751	154.8432	3.0027
	Comp. time/iteration (msecs)	21.5200	0.4254	21.6703	0.4249	22.0135	0.4253
	Computation speedup	1.000	50.5827	1.0000	50.9970	1.0000	51.7567

4.3 CPU - GPU Scalability

The CPU and GPU versions were executed for a wider range of mesh sizes with $\sqrt{N} = \{402, 1002, 2002, 3002, 4002\}$, for the 2nd problem and $Re = 10.0$. The results are depicted on table 4.

The speed-up observed is further increased as $\sqrt{N}$ obtains higher values. For mesh size with $\sqrt{N} = 4002$, the speed-up exceeds $\times 110$. The GTX-480 needs just 31.16 seconds to execute 5406 iterations on that mesh which is near 150 milliseconds per iteration. This rate reaches to 2.8 Giga elements computed per second. These numbers include the time required for checking of convergence criterion.

The rate of computations of elements per second and the speed-up observed for the GPU computation times can be summarized on figure 1.

The C2050, although targeted to HPC environments it lacks the high bandwidth of the GTX480. Additionally, as the Tesla ECC protections was enabled, the memory bandwidth was further stressed roughly by 20% [18]. Thus, the performance results are lower on C2050 than on GTX-480, which does not feature ECC memories. The CPU version achieves about 25 MegaElements/sec which corresponds to $8 \times 8 \times 25 = 1600$ MB/sec bandwidth. This straightforward CPU implementation, features strided accesses (reading red or black elements) that avoid vectorization and data are used inefficiently as only half of them read in a cache line are actually used in computations.

Table 4. Various executions for the 2nd problem, (a) on CPU AMD Opteron 6180 SE, (b) on GPU NVidia GTX480 (kernel #2) and (c) on GPU NVidia Tesla C2050 (kernel #2), for mesh sizes with $\sqrt{N} = \{402, 1002, 2002, 3002, 4002\}$ and Re=10.0

Matrix $\sqrt{N} \times \sqrt{N}$	Total Iterations	(R,I)	Model	Execution time	Computation time	Mega Elements computed per second	Computation Speed-up
402 × 402	554	(161604,0)	(a)	1.54	1.54	57.55	1.00
			(b)	0.07	0.07	1329.55	23.10
			(c)	0.14	0.12	721.78	12.54
1002 × 1002	1384	(1004004,0)	(a)	28.19	28.19	49.10	1.00
			(b)	0.60	0.59	2354.79	47.96
			(c)	0.91	0.89	1555.39	31.68
2002 × 2002	2704	(4008004,0)	(a)	256.50	256.50	42.17	1.00
			(b)	4.10	4.06	2665.86	63.22
			(c)	6.02	5.91	1831.57	43.44
3002 × 3002	4055	(9012004,0)	(a)	1402.17	1402.17	26.03	1.00
			(b)	13.35	13.27	2750.96	105.69
			(c)	20.43	20.05	1820.49	84.55
4002 × 4002	5406	(16016004,0)	(a)	3473.73	3473.73	24.90	1.00
			(b)	31.30	31.16	2776.04	111.49
			(c)	46.71	46.27	1869.18	75.07

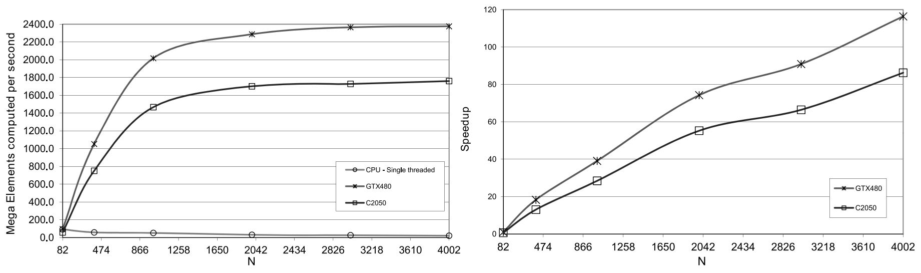

Fig. 1. Mega Elements computed per second on CPU & GPUs (left) and Computation speed-up of GPUs over CPU (right) for different matrices

5 Remarks and Conclusions

GPU is a suitable platform for massive parallel computations like those provided by the red/black ordering of iterative methods in solving systems of linear equations. In order to achieve memory coalescing, the locality of accesses must be ensured. Thereafter, the high memory bandwidth of the GPU can be exploited and attain high performance.

GPU recomputation can be beneficial in cases where memory accessing becomes a bottleneck. Instead of keeping the processing units idle, one strategy is to recompute data in order to avoid multiple memory accesses. This is a tradeoff and in many cases when a kernel is bandwidth limited, compute resources can be traded for less demand in memory bandwidth. It is applicable when a few operations at most are required for recomputation, so that computation does not turn to a bottleneck. It can provide a performance speed-up and moreover, it can release portions of device memory, allowing to solve larger problems.

Even in cases where recomputation is applied to the extreme, although performance is worsened, there can be other benefits. Recomputation leaves more available memory for other uses and thus a bigger problem is allowed to be solved. The size of the problem that is to be solved can determine the appropriate kernel to be used.

Acknowledgments. We would like to acknowledge the kind permission of the Innovative Computing Laboratory at the University of Tennessee to use their NVidia Tesla C2050 installation for the purpose of this work.

References

1. Adams, L.M., Leveque, R.J., Young, D.: Analysis of the SOR iteration for the 9-point Laplacian. SIAM J. Num. Anal. 9, 1156–1180 (1988)
2. Botta, E.F., Veldman, A.E.P.: On local relaxation methods and their application to convection-diffusion equations. J. Comput. Phys. 48, 127–149 (1981)
3. Boukas, L.A., Missirlis, N.M.: The Parallel Local Modified SOR for Nonsymmetric Linear Systems. Intern. J. Computer Math. 68, 153–174 (1998)

4. Ehrlich, L.W.: An Ad-Hoc SOR Method. J. Comput. Phys. 42, 31–45 (1981)
5. Ehrlich, L.W.: The Ad-Hoc SOR method: A local relaxation scheme, in elliptic Problem Solvers II, pp. 257–269. Academic Press, New York (1984)
6. Ha, L., Króger, J., Joshi, S., Silva, C.T.: Multiscale Unbiased Diffeomorphic Atlas Construction on Multi-GPUs. GPU Computing Gems. Emerald Edition, pp. 771–791. Morgan Kaufmann (2011)
7. Hageman, L.A., Young, D.M.: Applied Iterative Methods. Academic Press, New York (1981)
8. Kirk, D.B., Hwu, W.W.: Programming Massively Parallel Processors. Morgan Kaufmann (2009)
9. Komatsu, K., Soga, T., Egawa, R., Takizawa, H., Kobayashi, H., Takahashi, S., Sasaki, D., Nakahashi, K.: Parallel Processing of the Building-Cube Method on the GPU Platform. In: Computers & Fluids Special Issue "22nd International Conference on Parallel Computational Fluid Dynamics", vol. 45(1), pp. 122–128 (2011)
10. Konstantinidis, E., Cotronis, Y.: Accelerating the Red/Black SOR Method Using GPUs with CUDA. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2011, Part I. LNCS, vol. 7203, pp. 589–598. Springer, Heidelberg (2012)
11. Kuo, C.-C.J., Levy, B.C., Musicus, B.R.: A local relaxation method for solving elliptic PDE's on mesh-connected arrays. SIAM J. Sci. Statist. Comput. 8, 530–573 (1987)
12. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable Parallel Programming with CUDA. In: ACM SIGGRAPH 2008 Classes, vol. 16, pp. 1–14 (2008)
13. Ortega, J.M., Voight, R.G.: Solution of Partial Differential Equations on Vector and Parallel Computers. SIAM, Philadelphia (1985)
14. Varga, R.S.: Matrix Iterative Analysis. Prentice-Hall, Englewood (1962)
15. Young, D.M.: Iterative Solution of Large Linear Systems. Academic Press, New York (1971)
16. NVidia CUDA Reference Manual v. 4.0, NVidia (2011)
17. NVidia CUDA C Best Practices Guide Version 4.0, NVidia (2011)
18. Tuning CUDA Applications for Fermi, NVidia (2011)
19. Tesla C2050 And Tesla C2070 Computing Processor Board, NVidia (2011)
20. The OpenCL Specification, Khronos group (2009)