# Optimized Three-Dimensional Stencil Computation on Fermi and Kepler GPUs

Anamaria Vizitiu, Lucian Itu, Cosmin Niță, Constantin Suciu
Siemens Corporate Technology, SC Siemens SRL
Department of Automation and Information Technology, Transilvania University of Braşov
Braşov, Romania

***Abstract*—Stencil based algorithms are used intensively in scientific computations. Graphics Processing Units (GPU) based implementations of stencil computations speed-up the execution significantly compared to conventional CPU only systems. In this paper we focus on double precision stencil computations, which are required for meeting the high accuracy requirements, inherent for scientific computations. Starting from two baseline implementations (using two dimensional and three dimensional thread block structures respectively), we employ different optimization techniques which lead to seven kernel versions. Both Fermi and Kepler GPUs are used, to evaluate the impact of different optimization techniques for the two architectures. Overall, the GTX680 GPU card performs best for a kernel with 2D thread block structure and optimized register and shared memory usage. We show that, whereas shared memory is not essential for Fermi GPUs, it is a highly efficient optimization technique for Kepler GPUs (mainly due to the different L1 cache usage). Furthermore, we evaluate the performance of Kepler GPU cards designed for desktop PCs and notebook PCs. The results indicate that the ratio of execution time is roughly equal to the inverse of the ratio of power consumption.**

***Keywords— stencil, GPU, double precision, Kepler, Fermi, optimization***

## I. INTRODUCTION

Graphics Processing Units (GPUs) are dedicated processors, designed originally as graphic accelerators. Since CUDA (Compute Unified Device Architecture) was introduced in 2006 by NVIDIA as a graphic application programming interface (API), the GPU has been used increasingly in various areas of scientific computations due to its superior parallel performance and energy efficiency [1].

The GPU is viewed as a compute device which is able to run a very high number of threads in parallel inside a kernel (a function, written in C language, which is executed on the GPU and launched by the CPU). The threads of a kernel are organized at three levels: blocks of threads are organized in a three dimensional (3D) grid at the top level, threads are organized in 3D blocks at the middle level, and, at the lowest levels, threads are grouped into warps (groups of 32 threads formed by linearizing the 3D block structure along the x, y and z axes respectively) [2].

The GPU contains several streaming multiprocessors, each of them containing several cores. The GPU (usually also called device) contains a certain amount of global memory to/from which the CPU or host thread can write/read, and which is accessible by all multiprocessors. Furthermore, each multiprocessor also contains shared memory and registers which are split between the thread blocks and the threads, which run on the multiprocessor, respectively. With the introduction of the third and fourth generation general purpose GPU (GPGPU), the Fermi and the Kepler generations respectively [3], [4], the double precision performance has increased, and a true cache hierarchy (L1/L2) and more shared memory are available. Furthermore, the global memory bandwidth plays an important role since the performance of many kernels is bound by the peak global memory throughput: current GPUs have a bandwidth of up to 300 GB/s. The shared memory on the other side is a fast on-chip memory which can be accessed with similar throughput as the registers.

Stencil computation is a computational pattern on an *n*-dimensional grid, whereas each location is updated iteratively as a function of its neighboring locations. This pattern is found in several application domains, like image processing, computational fluid dynamics, weather prediction, quantum physics. Previous studies have shown that, if regular Cartesian grids are used, GPU based implementations are able to significantly speed up the execution compared to regular CPU based implementations [5], [6].

Research activities on stencil based computations have been reported long before the introduction of general purpose GPUs. These activities focused on the information transfer between nodes [7] and the relationship between partition shape, stencil structure and architecture [8]. Different optimization techniques have been reported more recently for GPU based stencil computations. The most often encountered optimization techniques used in the past are blocking at registers and at shared memory [9], [10]. Pre-Fermi GPUs did not have any cache memories, making the shared memory blocking technique vital for reducing memory access counts. Temporal blocking is another extensively used technique, with mixed performance improvements on GPUs [11], [12], [13]. Non-GPU architectures have also been used for stencil based computations [14].

The goal of the current work is to evaluate the performance of 3D stencil based algorithms on a series of recent GPUs. Previous research activities have focused on single precision computations. With the introduction of the Fermi and the Kepler architecture, the performance of double precision computations on NVIDIA GPU cards has increased

978-1-4799-6233-4/14/$31.00 ©2014 IEEE

substantially. To meet the high accuracy requirements, inherent for scientific computations [15], [16], in the current work we focus on double precision computations. Starting from two baseline implementations, we employ different optimization techniques which lead to seven different kernel versions. Both Fermi and Kepler GPUs are used, to evaluate the impact of different optimization techniques for the two architectures.

The paper is organized as follows. Section II first performs a brief introduction of the 3D stencil used herein. Next, the baseline implementations are introduced, followed by the different optimized approaches. Section III displays the results obtained with the different kernel versions for different Fermi and Kepler GPUs. Finally, section IV draws the conclusions.

## II. METHODS

For studying 3D stencil based algorithms implemented on graphics processing units, we consider the 3D unsteady heat conduction problem which is modeled as a second order partial differential equation describing the distribution of heat over time over a given 3D space:

$$\frac{\partial T}{\partial t} - \alpha\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2}\right) = 0 \quad (1)$$

where α is the thermal diffusivity constant and $T$ represents the temperature at any point in space $(x,y,z)$ or time $(t)$.

For the numerical solution of (1) we apply a finite difference method on a 3D grid of points. A uniform mesh of points is used and the forward difference in time and central difference in space (FTCS) method is applied, leading to a 3D 7-point stencil:

$$\frac{T_{i,j,k}^{n+1} - T_{i,j,k}^{n}}{\Delta t} = \alpha\left(\frac{T_{i+1,j,k}^{n} - 2T_{i,j,k}^{n} + T_{i-1,j,k}^{n}}{\Delta x^2} + \frac{T_{i,j+1,k}^{n} - 2T_{i,j,k}^{n} + T_{i,j-1,k}^{n}}{\Delta y^2}\right. \quad (2)$$

$$\left. + \frac{T_{i,j,k+1}^{n} - 2T_{i,j,k}^{n} + T_{i,j,k-1}^{n}}{\Delta z^2}\right) = 0,$$

which can be rewritten as:

$$T_{i,j,k}^{n+1} = T_{i,j,k}^{n} + d(T_{i+1,j,k}^{n} + T_{i-1,j,k}^{n} + T_{i,j+1,k}^{n} + T_{i,j-1,k}^{n} + T_{i,j,k+1}^{n} + T_{i,j,k-1}^{n} - 6T_{i,j,k}^{n}), \quad (3)$$

where $d = \alpha\Delta t/\Delta x^2$.

In the above equation $n$ represents the discrete time step number, $(i,j,k)$ represents the spatial index, $\Delta t$ is the time step and $\Delta x$ is the mesh spacing, which is equal in all directions. $T_{i,j,k}^{n}$ represents the temperature value at point $(i,j,k)$, at time step $n$. The numerical solution is stable if the CFL condition holds: $d = \alpha\Delta t/\Delta x^2 < 1/6$.

As can be observed in (3), the value at a grid point at time step $n$+1 is computed from the values at the previous time step, from the same grid point and from its six neighboring points, leading to a 7 point stencil computation (fig. 1).

This solution scheme is fully explicit: the computation of the new value at any grid point is fully independent from the computations at the other grid points.

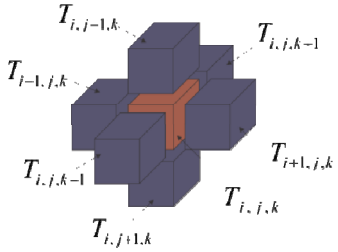


Fig. 1. 7-point stencil used for the numerical solution of the unsteady heat diffusion equation.

### A. Baseline GPU-based implementations

In the following we introduce two baseline GPU-based implementations of the unsteady heat diffusion problem. For the first baseline implementation (called in the following 3DBase) each grid point is handled by a separate thread. Two buffers are allocated, one for the values at the previous time step and one for the values at the new time step. To eliminate the memory copy requirement from one buffer to the other, the buffers are swapped at the end of each time step.

Since for the latest GPUs the execution configuration allows not only for 3D blocks of threads, but also for a 3D grid of thread blocks, the threads and the thread-blocks are organized both into 3D structures. Thus, each thread of the grid corresponds to a grid point in the 3-D computational domain. To compute the new value at a grid point each thread performs seven global memory read operations at each time step. Since global memory operations are very slow, this represents a severe limitation of the kernel performance.

In the CUDA architecture, each thread block is divided into groups of 32 threads called warps, each of which is executed in a SIMD fashion (all threads of the same warp execute the same instruction at a time). If the threads inside a warp follow different execution paths, the execution of the branches is serialized. Thus, warp divergence is another aspect which leads to loss of parallel efficiency (a minimum amount of warp divergence is required to distinguish between boundary and non-boundary nodes, so as to perform the computations only for the latter ones).

On the other hand, stencil codes can be characterized by their FLOPs per byte ratio. The baseline implementation performs 13 double-precision floating point operations per update [17]. This leads to $13 \cdot xDim \cdot yDim \cdot zDim$ operations performed at each iteration (*xDim*, *yDim* and *zDim* represent the grid dimensions). If we assume that at each time step once the old values are loaded they remain in the cache memory (which is unlikely for grid dimensions which exceed the cache size) the amount of data loaded and stored per time step is equal to $xDim \cdot yDim \cdot zDim \cdot$ sizeof(double) $\cdot$ 2. Hence the flop per DRAM byte ratio is:

$$\frac{13 \cdot xDim \cdot yDim \cdot zDim}{xDim \cdot yDim \cdot zDim \cdot sizeof(double) \cdot 2} = 0.8125. \quad (4)$$

Current GPUs, however, have a significantly higher ratio. According to this model the performance of the stencil on the GPU is therefore limited by its memory bandwidth.

To allow for a better memory usage, we also consider a more efficient approach, whereas threads and thread-blocks are organized into 2D structures. The computational grid is divided into x-y planes and the subdomains are assigned to separate thread blocks. Each 2-D slice is represented through the grid points in the *x* and *y* directions, providing for the threads the (*i*,*j*) indices of the grid points. A loop is then used to traverse the grid in the z-direction and obtain the final *k* coordinate as shown in fig. 2 (this kernel version is called in the following 2DBase).

Unlike the 3DBase implementation, for which a thread updates a single point, herein the same thread operates on several grid points. These points are placed equidistant from each other, the distance from one grid point to another is determined based on the size of the 3D domain (*xDim* · *yDim*).

### B. Optimized implementations

Next, we describe a series of optimization techniques for the two baseline implementations. We focus mainly on minimizing warp divergence and global memory accesses. Besides global memory, the GPU architecture provides fast on-chip memory, registers and shared memory, which is distributed between threads and thread blocks respectively.

*1) Three-dimensional baseline implementation with Shared Memory Usage and Data Overlap*

The starting point for the new kernel is the 3DBase implementation. Since shared memory is allocated at thread block level, threads can cooperate when populating data blocks allocated in the shared memory. If data can then be reused by different threads, global memory accesses are reduced and overall kernel performance is improved.

Shared memory arrays of size *blockXDim* · *blockYDim* · *blockZDim* are allocated (*blockXDim*, *blockYDim* and *blockZDim* represent the dimensions of the thread blocks).

Each thread within a block loads the value of the grid point it handles from global memory to shared memory. To avoid undefined behavior and incorrect results when sharing data read by different threads, a synchronization barrier is introduced. All values required for the implementation of (4) are then read from the shared memory.

With this technique, threads lying at the border of a thread block do not have access to all their neighbors and can not compute the corresponding new values. Hence, the execution configuration is designed so as to ensure block overlapping in all directions (fig. 3 - 3DShMOverL). This, however, results in global memory read redundancy: grid points lying in the overlapping regions of the blocks are read more than once for a single time step.

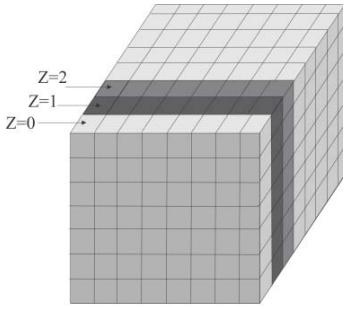


Fig. 2. 2DBase kernel: the computational grid is divided into x-y planes and a loop is then used to traverse the grid in the z-direction.

*2) Three-dimensional baseline implementation with Shared Memory Usage and No Data Overlap*

Starting again from the 3DBase implementation, a different shared memory based strategy is developed. The shared memory arrays are padded with an additional slice on each side of the 3D block leading to a total size of (*blockXDim* + 2) · (*blockYDim* + 2) · (*blockZDim* + 2), as shown in fig. 4.

First, each thread populates the value of the grid point it handles in shared memory. Next, the threads located on the boundary of the block load the remaining data slices from global memory to the shared memory (note that the corner points of the blocks are not required for the 7-point stencil). To load points located outside of the block, conditional operations are introduced which cause branch divergence.

Thus, each thread of a thread block has access to all its neighbors and is able to update the corresponding grid point (no overlapping between thread blocks is required - 3DShMNoOverL).

*3) Two-dimensional distribution of threads with additional register usage*

The 2DBase implementation can be optimized by storing redundant data in registers. Therein, the value of the current grid point for adjacent 2D slices is read from the global memory by the same thread. The same holds true for grid points which lie on the front or back sides of the 2D slices.

Because slices are iterated along the z direction, the value

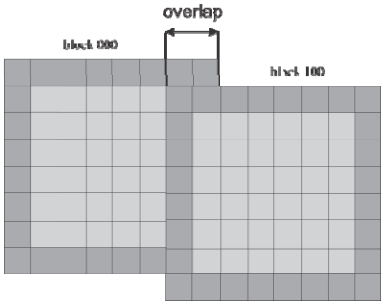


Fig. 3. 3DShMOverL kernel: the shared memory arrays have the same size as the thread blocks. Thread blocks overlap to enable the computation at all grid points.

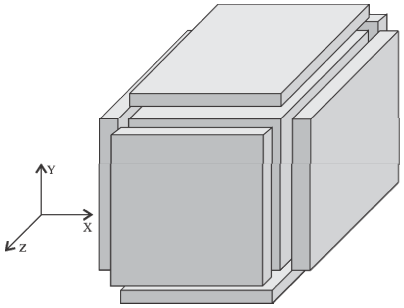


Fig. 4. 3DShMNoOverL: the shared memory arrays are padded with additional data slices loaded by the threads located at the border of the thread block.

at grid point ($i$, $j$, $k$+1) becomes the value at ($i$, $j$, $k$) at the next iteration. Similarly, the value at ($i$, $j$, $k$) becomes the value at ($i$, $j$, $k$–1). Instead of rereading these values, registers are used for caching them and two global memory accesses are saved at each iteration along the Z axis (in the following this kernel is called 2DReg).

*4) Two-dimensional distribution of threads with Shared Memory Usage*

As for the kernels with 3D thread blocks, shared memory can also be used to reduce global memory accesses for the kernels with 2D thread blocks. The size of the shared memory array chosen for this kernel version is (*blockXDim* + 2) · (*blockYDim* + 2). To allow each thread of the thread block to compute the new value of the corresponding grid point, additional slices are populated at each border of the 2D shared memory array. Hence, the size of the shared memory array used for this configuration is (*blockXDim* + 2) · (*blockYDim* + 2). Each thread first reads the value of the grid point it handles and stores it in the shared memory. Next, threads located on the boundary of the block load the remaining values (in the following this kernel is called 2DShM).

*5) Two-dimensional distribution of threads with Additional Register and Shared Memory Usage*

For the implementation version described in section II.B.4 the loading of the central section of the shared memory does not introduce any divergent branches since it is not conditioned. The loading of the slices with $y$ index equal to 0 or *blockYDim* + 2 introduces a maximum of two divergent branches, one for each half-warp, depending on the compute capability of the GPU. On the other side, the slices with $x$ index equal to 0 or *blockXDim* + 2 lead to divergent branches and only one thread of the entire half-warp performs a read operation. This aspect may be alleviated by the cache memory, but this depends on the size of the slices.

To reduce branch divergence, the shared memory array is used only for the central section and for the slices with index equal to 0 or *blockYDim* + 2, while the other values are read from the global memory and stored into registers. Only the threads lying at the left or right border perform separate global memory reads (fig. 5 - 2DShMReg), while the other values are safely read from the shared memory**.**

Hence the size of shared memory array used in this case is *blockXDim* · (*blockYDim* + 2). Each thread first reads the value of the grid point it handles and stores it in the shared memory. Next, threads located on the upper and lower boundary of the block load the remaining values.

Besides the two registers that store the values of the nodes located next to the left and right boundaries, another 2 registers are used for the optimization described in section II.B.2.

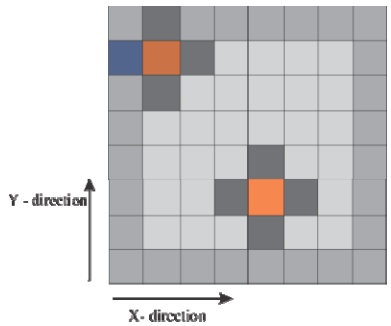


Fig. 5. 2DShMReg: Northern and southern slices are read from the shared memory, eastern and western values from the global memory.

## III. Results

To evaluate the performance of the different strategies for running 3D stencil based algorithms on GPUs, we used three different NVIDIA GPU cards: GeForce GTX 480, GeForce GTX 660M and GeForce GTX 680 (the first one is based on the Fermi architecture, while the other two are based on the Kepler architecture), and the CUDA toolkit version 5.5. The unsteady heat conduction problem was solved on a rectangular domain with Dirichlet boundary conditions, whereas the boundary values were set to 100□ C for one side of the rectangle and 0□ C for the other sides. The thermal diffusivity constant was se to $1.9 \cdot 10^{-5}$ $m^2/s$ and the computations are performed until convergence is reached. The numerical solution was obtained on a grid of 128x128x128 nodes and is displayed in fig. 6. The numerical solution was identical for all three GPU cards and for all implementation versions down to the 15$^{th}$ decimal, i.e. close to the precision of the double-type representation in computer data structures.

Table 1 displays the execution times for one time step for the three above mentioned GPU cards, obtained for the seven different kernel versions introduced in the previous section. The GTX660M card leads to the largest execution times although it has been considerably later released compared to the GTX480 card. This can be explained however by the fact that this card was specifically designed for low power consumption, so as to be used in notebook PCs (whereas the GTX480 and GTX680 were reported with a power consumption of 250W and 195W respectively, the GTX660M only required 50W). The GTX680 is the best performing card: for each of the seven implementation versions it leads to the smallest execution times. The ratio of the execution times for the GTX660M and GTX680 cards varies between 4.26 and 5.56 for different kernel versions. This roughly reflects the inverse of the power consumption ratio, which is equal to 3.9.

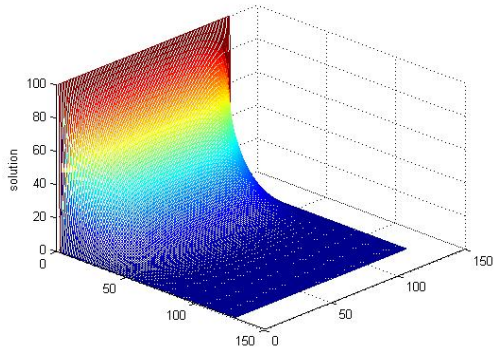


Fig. 6. Computation result for the unsteady heat conduction problem on a rectangular domain with Dirichlet boundary conditions.

TABLE I. EXECUTION TIME [MS] FOR A SINGLE TIME STEP, OBTAINED FOR THE SEVEN DIFFERENT IMPLEMENTATION VERSIONS ON THREE DIFFERENT GPU CARDS .

| Method | GTX480 | GTX 660M | GTX 680 |
|---|---|---|---|
| 3DBase | 1.7 | 3.45 | 0.62 |
| 3DShMOverL | 3.5 | 6.17 | 1.13 |
| 3DShMNoOverL | 1.8 | 3.78 | 0.73 |
| 2DBase | 1.2 | 3.09 | 0.63 |
| 2DReg | 0.9 | 2.47 | 0.58 |
| 2DShM | 1.2 | 2.87 | 0.59 |
| 2DShMReg | 1.09 | 2.32 | 0.48 |

Interestingly, whereas for the GTX660M and the GTX680 cards the 2DShMReg kernel performs best, for the GTX480 card the 2DReg kernel leads to the smallest execution time. Shared memory based optimizations were particularly important for pre-Fermi GPU cards. For the Fermi architecture these optimizations were not always leading to a better performance due to the fact that the global memory read operations were cached at L1 level. Even though the cache size is regularly small, it is efficient for algorithms based on Cartesian grids where data access patterns are regular [6]. For the Kepler architecture however the L1 cache is no longer used for caching global memory read operations, but only for register spilling [4]. Hence, for the GTX480 card (Fermi), since the L1 cache is intensively used for caching global memory read operations, the 2DReg kernel outperforms the 2DShMemReg kernel. On the other hand, for the GTX660M and the GTX680M, since the L1 cache functionality is limited to register spilling, shared memory usage became more important, illustrated by the better performance of the 2DShMReg kernel.

In the following we focus on the differences between the kernel versions for the GTX680 card, which was determined as the best performing one considered herein. Table 2 displays besides the execution time other important details of the various kernel versions.

The two baseline implementations (2DBase and 3DBase) lead to almost identical execution times. Referring first to the kernels based on a 3D thread block structure, the 3DShMOverL performs worse than the 3DBase kernel: execution time increased by 82% although the number of global accesses was reduced by 66.13%. This can be explained by the fact that a considerable amount of threads perform only load operations.

Compared to the 3DShMOverL kernel, the execution time decreased by 35.39% and the total number of read operations was reduced by 25.66% for the 3DShMNoOverL kernel. Compared to the 3DBase kernel, this implementation is compute limited instead of bandwidth limited. The main reason for the change of the limitation type lies in the number of divergent branches, which increased considerably and which in the end leads to a higher execution time than for the 3DBase kernel.

Next, we refer to the kernels based on a 2D thread block structure. The 2DReg kernel leads to a significant reduction of memory operations (28.34%) and as a result of the execution time (7.93%), compared to the 2DBase kernel. The 2DShM kernel further reduces the number of global memory load operations but execution time increases slightly, which is caused by the non-optimized register usage. Finally the 2DShMReg combines both techniques (optimized register and shared memory usage), and reduced execution time by 17.24% and the total number of read operations by 70.25% compared to the 2Dreg kernel.

Overall, the kernels with 2D thread block structure outperform the ones with 3D thread block structure for double precision computations, confirming the findings for single precision computation reported in [17].

## IV. CONCLUSIONS

In this paper, we have presented performance studies for 3D stencil based algorithms on recent NVIDIA GPUs. To our knowledge this is the first study to evaluate different implementation and optimization strategies for double precision computations. The increased accuracy obtained for double precision is required in scientific computations, which represent the main area of application for the 3D stencil based algorithms.

For the analysis we have used Fermi and Kepler architecture based cards, which represent the last two released GPU architectures from NVIDIA. Besides the shift in L1 cache usage from Fermi to Kepler, other important minor and major changes in the hardware configuration have been performed [4].

Hence, starting from two different baseline implementations (based on 3D and 2D thread block structures), we have applied different optimization strategies which have lead to different performance changes for the Fermi and Kepler cards. Overall the GTX680 GPU card (Kepler architecture) performed best for a kernel with 2D

TABLE II. KERNEL PERFORMANCE AND DETAILS FOR THE GTX680 CARD.

| Method | Execution time [ms] | Reg. per thread | Divergent branches | Shared memory per block [bytes] | Total number of 64 bit global load instr. | Total number of 64 bit global store instr. |
|---|---|---|---|---|---|---|
| 3DBase | 0.62 | 25 | 12016 | - | 14002632 | 2000376 |
| 3DShMOverL | 1.13 | 19 | 20811 | 4096 | 4741632 | 2000376 |
| 3DShMNoOverL | 0.73 | 21 | 12694 | 8000 | 3524851 | 2000376 |
| 2DBase | 0.63 | 25 | 94 | - | 14002632 | 2000376 |
| 2DReg | 0.58 | 25 | 94 | - | 10033632 | 2000376 |
| 2DShM | 0.59 | 25 | 94 | 800 | 6953688 | 2000376 |
| 2DShMReg | 0.48 | 25 | 94 | 640 | 2984688 | 2000376 |

thread block structure and optimized register and shared memory usage. Conversely, for the GTX480 GPU card (Fermi architecture) the 2D kernel, which does not use shared memory but is optimized in terms of register usage, performed best, mainly due to the different L1 cache usage in the Fermi architecture. Hence, shared memory usage has become essential for double precision stencil based computation on Kepler GPUs.

Finally, for the Kepler architecture we have evaluated the performance for a GPU designed for desktop PCs (GTX680) and for a GPU designed for notebook PCs (GTX660M). The results have indicated that the ratio of execution time is roughly equal to the inverse of the ratio of power consumption.

## ACKNOWLEDGMENT

This work is supported by the program Partnerships in Priority Domains (PN II), financed by ANCS, CNDI - UEFISCDI, under the project nr. 130/2012.

This paper is supported by the Sectoral Operational Programme Human Resources Development (SOP HRD), ID134378 financed from the European Social Fund and by the Romanian Government.

The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 600932.

We hereby acknowledge the structural funds project PRO-DD (POS-CCE, O.2.2.1., ID 123, SMIS 2637, ctr. No 11/2009) for providing the infrastructure used in this work.

## REFERENCES

[1] D. Kirk, and W.M. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, London: Elsevier, 2010.

[2] NVIDIA Corporation, "CUDA, Compute unified device architecture Programming Guide v5.5", 2013.

[3] NVIDIA Corporation, "NVIDIA's next generation CUDA compute architecture: Fermi", 2011.

[4] NVIDIA Corporation, "NVIDIA Kepler GK110 Architecture Whitepaper", 2013.

[5] E. Phillips, and M. Fatica, "Implementing the Himeno benchmark with CUDA on GPU clusters", IEEE Intern. Parallel & Distributed Processing Symposium, pp. 1-10, April 2010.

[6] T. Shimokawabe, T. Aoki, T. Takaki, T. Endo, A. Yamanaka, N. Maruyama, A. Nukada, and S. Matsuoka, "Peta-scale phase-field simulation for dendritic solidification on the TSUBAME 2.0 supercomputer", Intern. Conf. for High Performance Computing, Networking, Storage and Analysis, pp. 13-18, 2011.

[7] G.C. Fox, "Concurrent processing for scientific calculations", IEEE Computer Society International Conference, pp. 70-73, 1984.

[8] D. Reed, L. Adams, and M. Patrick, "Stencils and problem partitionings: Their influence on the performance of multiple processor systems", IEEE Trans. Comput., vol. 7, pp. 845-858, 1987.

[9] P. Micikevicius, "3D Finite difference computation on GPUs using CUDA". Workshop on General Purpose Processing on Graphics Processing Units, pp. 79-84, 2009.

[10] L. M. Itu, C. Suciu, F. Moldoveanu, and A. Postelnicu, "GPU Optimized Computation of Stencil Based Algorithms", RoEduNet Inter. Conf., pp. 1-4, June 2011.

[11] T. Grosser, A. Cohen, P. H. J. Kelly, J. Ramanujam, P. Sadayappan, and S. Verdoolaege, "Split tiling for GPUs: automatic parallelization using trapezoidal tiles", Workshop on General Purpose Processor Using Graphics Processing Units, pp. 24-31, 2013.

[12] J. Holewinski, L. N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on GPU architectures", ACM Intern. Conf. on Supercomputing, pp. 311-320, 2012.

[13] G. Zumbusch, "Vectorized higher order finite difference kernels", Lecture Notes in Computer Science, vol. 7782, pp. 343-357, 2013.

[14] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and autotuning on state-of-the-art multicore architectures", ACM/IEEE Conf. on Supercomputing, pp. 1-12, Nov. 2008.

[15] C. Niţă, L. M. Itu, and C. Suciu, "GPU Accelerated Blood Flow Computation using the Lattice Boltzmann Method", IEEE High Performance Extreme Computing Conference, pp. 1-6, Sept. 2013.

[16] P. Zaspel, M. Griebel, "Solving incompressible two-phase flows on multi-GPU clusters", Computers & Fluids 2012, vol. 80, pp. 356-364, 2013.

[17] N. Maruyama, and T. Aoki, "Optimizing stencil computations for NVIDIA Kepler GPUs", Intern. Workshop on High-Performance Stencil Computations, pp. 1-7, Jan. 2104.