CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2015; **27**:937–957
Published online 14 October 2014 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3417

SPECIAL ISSUE PAPER

Adaptation of fluid model EULAG to graphics processing unit architecture

Krzysztof Andrzej Rojek[1,*,†], Milosz Ciznicki[2], Bogdan Rosa[3], Piotr Kopta[2], Michal Kulczewski[2], Krzysztof Kurowski[2], Zbigniew Pawel Piotrowski[3], Lukasz Szustak[1], Damian Karol Wojcik[3] and Roman Wyrzykowski[1]

[1]*Czestochowa University of Technology, Czestochowa, Poland*
[2]*Poznan Supercomputing and Networking Center Applications, Poznan Wielkopolskia, Poland*
[3]*Institute of Meteorology and Water Management-National Research Institute, Warsaw, Poland*

SUMMARY

The goal of this study is to adapt the multiscale fluid solver EULerian or LAGrangian framewrok (EULAG) to future graphics processing units (GPU) platforms. The EULAG model has the proven record of successful applications, and excellent efficiency and scalability on conventional supercomputer architectures. Currently, the model is being implemented as the new dynamical core of the COSMO weather prediction framework. Within this study, two main modules of EULAG, namely the multidimensional positive definite advection transport algorithm (MPDATA) and the variational generalized conjugate residual, elliptic pressure solver Generalized Conjugate Residual (GCR) are analyzed and optimized. In this paper, a method is proposed, which ensures a comprehensive analysis of the resource consumption including registers, shared, and global memories. This method allows us to identify bottlenecks of the algorithm, including data transfers between host and global memory, global and shared memories, as well as GPU occupancy. We put the emphasis on providing a fixed memory access pattern, padding as well as organizing computation in the MPDATA algorithm. The testing and validation of the new GPU implementation have been carried out based on modeling decaying turbulence of a homogeneous incompressible fluid in a triply-periodic cube. Simulations performed using the standard version of EULAG and its new GPU implementation give similar solutions. Preliminary results show a promising increase in terms of computational efficiency. Copyright © 2014 John Wiley & Sons, Ltd.

Received 19 May 2014; Revised 20 August 2014; Accepted 14 September 2014

KEY WORDS: parallel programming; GPGPU; CUDA; EULAG; MPDATA; stencils; elliptic solver

1. INTRODUCTION

In recent years, there has been growing interest in employing heterogeneous and hybrid supercomputing architectures for modeling complex physical processes. An especially promising application for new architectures is computational fluid dynamics (CFD), and particularly, the numerical weather prediction (NWP) [1]. The adaptation of conventional CFD codes to modern supercomputing architectures offers a unique opportunity for modeling complex physical flows with accuracy greater than ever before. The new parallel computers based on multicore and many-core processors (CPUs) and graphics processing units (GPUs) enable to increase computational efficiency and reduce energy consumption [2–4]. Consequently, more computational resources, that is, processing

*Correspondence to: Krzysztof Andrzej Rojek, Czestochowa University of Technology Dabrowskiego 69 St. Czestochowa 42-201 Poland.
†E-mail: krojek@icis.pcz.pl

units and memory can be employed. This in turn allows to increase complexity of the models, so that more details that may affect the evolution of the system can be captured.

To be able to run the traditional codes efficiently on new hybrid platforms, it is necessary to redesign their structures. Nowadays, several research centers around the world are involved in various projects aimed at adapting weather forecasting models to future high-performance computing platforms. One of such projects named 'Performance On Massively Parallel Architectures' (POMPA) has been launched in 2010 by the COSMO consortium [5]. The goal of the POMPA project is to develop a prototype implementation of the current COSMO numerical weather prediction (NWP) model for modern GPU-based and CPU-based computing hardware. To-date results show a large potential of the new implementations in terms of reducing the time-to-solution. It is worth noting that the hardware cost for running the redesigned model, both on CPU-based and GPU-based machines, is significantly lower. The newly developed dynamical core is robust and capable of running COSMO-7 (horizontal resolution of 7 km) and COSMO-2 (horizontal resolution of 2.2 km), where 2 km grid resolution is nowadays common in regional weather forecast models used in operation.

The adaptation of traditional NWP codes to new machines based on GPUs allows for increasing the numerical efficiency, and enables one to take full advantage of available computational resources. This offers a unique opportunity to develop simulations with finer grid resolutions and larger computational domains. Refined grid resolutions in simulation of meso-scale and large-scale atmospheric flows for NWP and climate studies may have profound impact on improving the reliability of prognoses. This is because at the convective scales of O(1) km, flows are highly turbulent and contain a significant amount of energy [1]. In order to explicitly resolve or even admit convective processes, the grid spacing has to be fine enough. Because of computational constraints, these processes are grossly under-resolved in today's simulations.

The new methods and algorithms at work on modern architectures should allow one to dispense with a large part of convective parameterizations in global models and improve numerical weather forecasts. The aim of this study is to develop effective methods and algorithms for adapting multiscale model EULAG [6] to GPU architectures. EULAG belongs to a class of numerical models for all-scale flows in geophysics and astrophysics. The dynamical core of EULAG is based on the non-hydrostatic Euler equations, either fully compressible or anelastic. The model employs the generalized curvilinear coordinate description [7], finite-volume non-oscillatory transport algorithm multidimensional positive definite advection transport algorithm (MPDATA) [8], and advanced elliptic solver GCR [9]. Since 2008, EULAG is a candidate for the dynamical core of a very-high resolution NWP model of the COSMO consortium. The dynamical core of EULAG has considerable advantages concerning conservation properties. Moreover, the modeling of atmospheric flows with EULAG does not impose severe constraints on the maximal allowable steepness of the surface orography.

In this paper, we focus on the parallelization of the EULAG model and its porting to modern GPU architectures. For this aim, the analysis of resources usage in GPU platforms and its influence on the overall system performance are provided. We detect the bottlenecks and develop a method of efficient distribution of computation across GPU kernels. The proposed method is based on the analysis of memory transactions between GPU global and shared memories. We unveil crucial details of different strategies used to accelerate the code execution, namely stencil [10] decomposition, block decomposition (with weighting analysis between computation and communication), reduction of inter-memory communication, and register file reusing.

An efficient implementation of seven-point and 27-point stencils on high-end NVIDIA GPUs is proposed in [11]. The authors present a new method of reading the data from the global memory to the shared memory. The method avoids conditional statements; however, it extends the size of blocks stored in the shared memory. The efficient adaptation of the EULAG model to the GPU architecture requires to store many memory blocks in the shared memory, because the MPDATA algorithm requires to store six input/output matrices and some intermediate matrices for temporary results. For that reason, we propose another method that is based on reducing the GPU global memory transactions and estimating the most preferable number of GPU kernels for the algorithm.

A very common bottleneck of scientific codes is memory bandwidth. The main reason for this is a low algorithmic flop-per-byte ratio. GPUs offer an impressive computational performance exceeding a TFlop/s and a memory bandwidth that is often larger than 250 GB/s. Although these numbers seem high, modern CPU-based architectures can deliver similar performance as the GPUs [11]. The often reported 100-fold speedups with GPUs are mainly due to a use of a single CPU core and/or inefficient CPU implementations in the comparison. For many practical problems, the potential speedup of the GPUs with respect to CPU-based architectures is limited [3].

The paper is organized as follows. Related works are outlined in Section 2, while Section 3 presents the target GPU architectures, as well as the programming environment used in our research. Sections 4 and 5 describe the adaptation of MPDATA and elliptic solver to the GPU architecture, respectively. Section 6 presents the synergy between the EULAG components. A validation test case is presented in Section 7, while Section 8 gives conclusions and future work.

2. RELATED WORKS

Stencil computations on regular grids are kernels for a wide range of scientific codes. Among them are the MPDATA algorithm and elliptic solver GCR, which form the dynamical core of the EULAG model. In these computations, each point in a multidimensional grid is updated with contributions from a subset of its neighbors [12]. Reorganizing stencil calculations to take full advantage of memory hierarchies has been the subject of much investigation over the years [13–15].

Modern processor architectures tend to be inherently unbalanced concerning the relation of theoretical peak performance versus memory bandwidth [16]. To reveal performance constraints for the MPDATA algorithm and GCR solver running on hybrid architectures, we will follow the simple methodology presented in [17], where attainable performance is estimated based on the flop-per-byte ratio.

The MPDATA algorithm is among the most time-consuming calculations of the EULAG model [8, 18]. In our previous works [16, 19, 20], we proposed two decompositions of two-dimensional (2D) MPDATA computations, which provide adaptation to CPU and GPU architectures separately. The obtained performance results showed the possibility of achieving high performance both on CPU and GPU platforms. Recently [2, 21], we developed a hybrid CPU–GPU version of 2D MPDATA, to fully utilize all the available computing resources by spreading computations across the entire machine. This implementation is the starting point for our current work. However, in this paper, we focus on the three-dimensional (3D) version of the EULAG model. It requires to analyze and implement the MPDATA algorithm using different methods.

Several different techniques for porting the GCR elliptic solver to hybrid architectures have been presented in our previous study [18]. The proposed techniques relied on porting the MPI code to the hybrid MPI + OpenMP version.

Memory optimizations for stencil computations have principally focused on different decomposition strategies, like space and temporal blocking techniques [22], that attempt to exploit locality by performing operations on data blocks of a suitable size before moving on to the next block. These strategies have been used to improve the efficiency of implementing stencil codes on a variety of multicore/manycore architectures, including CPUs and GPUs (see, e.g., [4, 23–25]).

The issue of adapting the EULAG model to GPU accelerators was discussed in [26], where the PGI accelerator compiler was used for the automatic parallelization of selected parts of EULAG on NVIDIA GPUs, including the 2D MPDATA algorithm. However, the disadvantage of this approach is a complete dependency on the automatic parallelization, without any efforts to guide the parallelization process taking into account characteristics of target architectures.

A 3.5D-blocking algorithm that performs 2.5D-spatial blocking of the input grid into on-chip memory for GPUs was discussed in [25]. In present work, we also employ 2.5D blocking technique to increase data locality, but we propose an alternative solution for memory-bounded kernels, which is based on minimizing the number of global memory transactions, rather than applying 3.5D-blocking.

Quite large set of techniques of GPU optimizations including data parallelism, threads deployment, and the GPU memory hierarchy was discussed in [27], where the authors manually evaluated the best configurations of 2D stencil computations. We offer a model-based solution, which automatically configures the code, making our solution more portable.

3. GPU ARCHITECTURE AND PROGRAMMING ENVIRONMENT

All the experiments presented in this work are performed using the NVIDIA Tesla K40m GPU [28], which is based on the Kepler architecture. It includes 15 streaming multiprocessors (SMX), each consisting of 64 double precision (DP) units with configurable size of 16/32/48 KB of shared memory and 48/32/16 KB of L1 cache. It gives the total number of $15 * 64 = 960$ DP units with the clock rate of 875.5 MHz, providing the peak performance of 1.68 TFlop/s in DP. This graphics accelerator card includes 12 GB of global memory with the peak bandwidth of 288 GB/s. All the accesses to the global memory go through the L2 cache of size 1.5 MB. This GPU supports 64-bit access mode. The number of load/store unit per SMX is 32, so it gives the possibility to load/store 256 bits at once per SMX.

To manage CPU and GPU components, we take advantage of using the Compute Unified Device Architecture (CUDA) programming standard [28]. In this paper, we use CUDA V5.5.0 assuming some prior knowledge about CUDA programming and terminology. For an informative description of the crucial aspects of GPU programming, the reader is referred to work [29]. CUDA is a scalable parallel programming model and a software environment for parallel computing. It allows for the utilization of a GPU as an application accelerator, when a part of an application is executed on a standard CPU processor, while another part is assigned to the GPU, as the so-called kernel.

CUDA enable for the efficient management of GPU computing resources, beginning with GPU CUDA cores that are grouped into SMX. In the CUDA data parallel model, the same program (or kernel) runs concurrently on different pieces of data, and each invocation is called a *thread*. The threads are organized in up to three dimensions. The set of threads is called a *block*. Each block is executed on a single SMX. Threads can be synchronized within a single block. However, there is no synchronization mechanism between blocks; they are executed independently. Threads within the block are organized in warps. The warp is a vector of 32 threads that actually executes the code. The block size is a multiple of warp size, even if fewer threads are requested.

Another key feature of modern GPUs is their hierarchical memory organization. In the CUDA memory model, all the GPU threads have access to the *global memory*, relatively large but rather slow. Within a particular block, all the threads share the fast *shared memory*. It is used for communication and synchronization among threads across the block. In addition, each thread has access to its *register file*. Furthermore, the L1 and L2 caches are applied to improve the data locality for memory accesses. In particular, all the accesses to the global memory go through L2, including copies to/from the CPU host.

Stencil algorithms that are of prime interest in this paper are commonly known as memory bounded [17]. Therefore, the memory bandwidth benchmarks, with both Error Correcting Code (ECC) mode switched on and off, are performed to find the maximum achievable performance, see Table I. The memory bandwidth benchmarks are based on a method proposed in [30]. Similarly to the reduction algorithm, a large array of floats is added up; however, the results are stored in a smaller array. Each thread reads two groups of four floats using $\_\_ldg$ intrinsic to utilize the read only data cache.

Table I. Benchmarked memory bandwidth of Kepler architecture.

Platform	Maximum bandwidth	Measured bandwidth (ECC = 1)	Measured bandwidth (ECC = 0)
Kepler K20m	208 GB/s	143 GB/s	173 GB/s
Kepler K40m	288 GB/s	196 GB/s	233 GB/s

4. ADAPTATION OF ADVECTION TRANSPORT ALGORITHM MPDATA TO GPU ARCHITECTURE

In this section, we present our new approach that allows us to efficiently distribute computational tasks of MPDATA across GPU resources. The proposed technique is based on the optimization of stencil computations, and is an extension of the concept described in our previous studies [19–21, 31].

The MPDATA algorithm is a set of 17 stencils, where each stencil may depend on one or more others. It requires to load five input matrices, and returns only a single output matrix. We assume that the size of the 3D MPDATA grid determined by coordinates i, j, and k is $n \times m \times l$.

4.1. *Processing GPU kernels*

To increase the data locality within CUDA blocks, we employ a widely used method of 2.5D blocking [25], in which 2D blocks are responsible for computing $g_1 \times g_2$ data chunks, which correspond to sub-planes of a matrix, called here tiles. Between neighboring blocks, some extra computations take place on the borders. As a consequence, blocks have to be extended by adequate halo areas, both in vertical and horizontal directions.

The loop inside a GPU kernel is used to traverse the grid in the dimension l. Because the MPDATA algorithm requires to store at most $3 \times (g_1 \times g_2)$ data chunks at the same time, we use a queue of data chunks placed in registers and shared memory. In this approach, we first copy data from the GPU global memory to registers, and then, for each iteration across the dimension l, we move data between registers and shared memory. This method is illustrated in Figure 1.

The main advantage of this technique in relation to the 3D blocking is the reduction of memory requirements. We need to store only three tiles of each matrix instead of the entire column of size l to keep the same intensity of memory traffic between the global memory and shared memory or register file. It is particularly useful for GPUs, where the size of shared memory is too small to store 3D blocks of matrices.

4.2. *Analysis of 3D MPDATA with NVIDIA visual profiler*

The starting point of our considerations is when all the 17 stencils are distributed across six GPU kernels marked as A, B, C, D, E, and F (this distribution takes into account synchronization points of MPDATA). Such a number of kernels is selected for the following reasons: (1) the stencils are

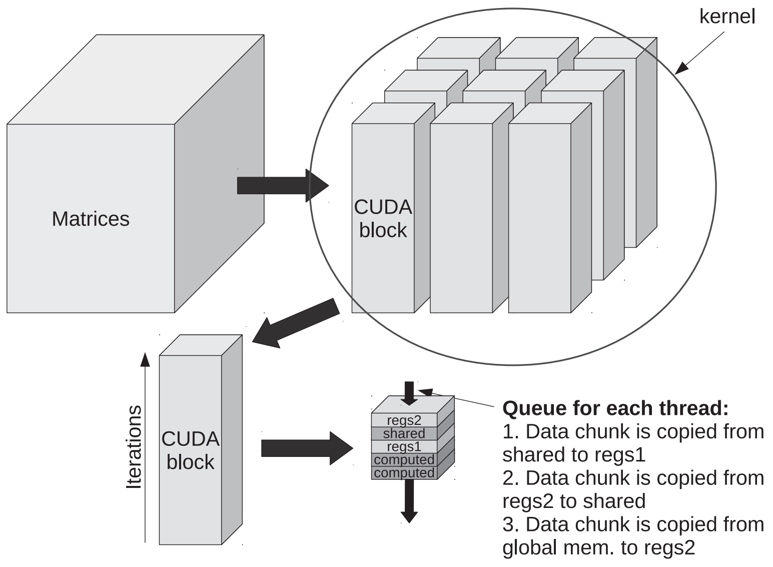
kernel
Matrices
CUDA block
Iterations
CUDA block
regs2
shared
regs1
computed
computed
Queue for each thread:
1. Data chunk is copied from shared to regs1
2. Data chunk is copied from regs2 to shared
3. Data chunk is copied from global mem. to regs2

Figure 1. Graphics processing units kernel processing.

```
for(k=1; k<l-1; ++k) {
  q1 =  fmax(0.0,v3(i,j,k+1))*x(i,j,k)
       +fmin(0.0,v3(i,j,k+1))*x(i,j,k+1);
  xP[ijk]=x[ijk]-( fmax(0.0,v1(i+1,j,k))*x(i,j,k)
  +fmin(0.0,v1(i+1,j,k))*x(i+1,j,k)-fmax(0.0,v1(i,j,k))*x(i-1,j,k)
  -fmin(0.0,v1(i,j,k))*x(i,j,k)+fmax(0.0,v2(i,j+1,k))*x(i,j,k)
  +fmin(0.0,v2(i,j+1,k))*x(i,j+1,k)-fmax(0.0,v2(i,j,k))*x(i,j-1,k)
  -fmin(0.0,v2(i,j,k))*x(i,j,k)+q1-q0 )/h[ijk];
  q0 = q1;
  ijk+=M;
}
```

Figure 2. Multidimensional positive definite advection transport algorithm stencil corresponding to kernel A (kernel F is of the same structure).

```
v1[ijk]=(0.5*fabs(v1(i,j,k))-v1(i,j,k)*v1(i,j,k)/
          (h(i-1,j,k)+h(i,j,k))
        )*(xP(i,j,k)-xP(i-1,j,k))
       -0.0625*v1(i,j,k)/(h(i-1,j,k)+h(i,j,k))*
  (
    (v2(i-1,j,k)+v2(i-1,j+1,k)+v2(i,j+1,k)+v2(i,j,k))*
    (xP(i,j+1,k)+xP(i-1,j+1,k)-xP(i,j-1,k)-xP(i-1,j-1,k))+
    (v3(i-1,j,k)+v3(i-1,j,k+1)+v3(i,j,k+1)+v3(i,j,k))*
    (xP(i,j,k+1)+xP(i-1,j,k+1)-xP(i,j,k-1)-xP(i-1,j,k-1))
  );
```

Figure 3. Stencil corresponding to computing pseudo velocity v1 (part of kernel B); kernel C of the same structure computes pseudo velocity v3 while computing pseudo velocity v2 is distributed across kernels B and C.

distributed in such a way that for each kernel, the halo area from any side of a CUDA block does not exceed 1; (2) the most memory-consuming and register-consuming stencils are implemented in kernels B and C in order to increase the GPU occupancy, defined as the number of active threads per SMX divided by the maximum number of threads supported by SMX. This version of our implementation is further referred to as the naive one. In this version, we applied the most common techniques of optimization, including the usage of the shared memory, coalesced memory access [29], and 2.5D blocking. The kernels A and F are responsible for computing the donor-cell scheme (Figure 2) of MPDATA, the kernels B and C compute pseudo velocities (Figure 3), while the kernels D and E implement the non-oscillatory option for the MPDATA algorithm [2]. Here, the variables $v1$, $v2$, and $v3$ correspond to pseudo velocities in the x, y, and z directions, respectively, while variable x is a non-negative scalar field.

Our idea of efficient adaptation of MPDATA to GPU architectures is based on the detection of bottlenecks, and allows for reducing the most notable of them. We examine the following potential bottlenecks:

- data transfers between GPU global memory and host memory;
- instructions latency (stall analysis);
- arithmetic, logic, and shared memory operations; and
- configuration of the algorithm taking into account the size of CUDA blocks, and GPU occupancy.

To overlap computation and data transfers between GPU global memory and host memory, we employ the stream processing technique [21], where each stream is responsible for performing a sequence of three activities including the following: (1) data transfer from host to GPU that occurs only once (before computations); (2) executing a sequence of six GPU kernels; (iii) data transfer from GPU to host memory (occurs after every time step). All the activities are processed synchronously within a single stream. However, all the streams are processed asynchronously. Thanks to that, the activities from one stream are overlapped by the activities from another one. Table II shows the time consumption analysis of MPDATA for 100 time steps and grid of size $392 \times 256 \times 64$. Three streams are used in the simulation. The HTOD abbreviation corresponds to data transfers from host to GPU device, while the DTOH corresponds to data transfers in the opposite direction.

Table II. Time consumption analysis of 3D MPDATA algorithm.

Activity	Time [s]	Ratio R
HTOD	0.023	0.008
DTOH	0.453	0.172
Total time of computation	3.051	1.16
Total execution time	2.631	1

3D, three dimensional; MPDATA, multidimensional positive definite advection transport algorithm.

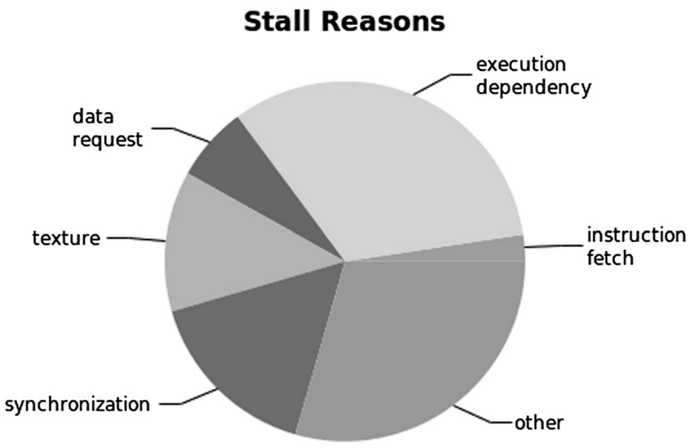

Figure 4. Analysis of stall reasons for kernels B and C.

The parameter R is computed as the ratio of the time required for a certain activity to the total execution time. It is worth noting that because of overlapping between the execution of kernels in different streams, the total execution time is shorter than the total time of computation.

Based on this analysis, we can see that the data transfer takes a relatively short time (about 18% of the total execution time). The stream processing decreases the execution time by $0.023 + 0.453 + 3.051 - 2.631 = 0.896$ s, which is about two times more than time of data transfers. We can conclude that data transfers between host and GPU are not a bottleneck of the MPDATA algorithm.

The next analysis is devoted to the stall reasons analysis. Figure 4 shows the main reasons of stalls for kernels B and C, including the analysis of execution dependency, data request, texture memory operations, synchronization, and instruction fetch.

Based on the analysis, stalls are mostly caused by the execution dependency (about 33%). This kind of stalls results from the complex structure of the MPDATA algorithm, and limits the GPU utilization. The execution dependencies can be hidden in part by increasing the GPU occupancy. However, each of the kernels B and C uses about 47 KB of shared memory for an active CUDA blocks, executing only 768 active threads per SMX (maximum is 2048 threads). It means that the GPU occupancy is only 37.5% for both kernels. So the final conclusion is that the GPU utilization is limited by the shared memory usage.

To perform the analysis of computation within kernels, we focus on the most complex part of the MPDATA algorithm. Using NVIDIA Visual Profiler tool, we select the kernels B and C as the most time-consuming ones. They take about 57% of the total computation time. Each of these kernels has five input and three output matrices, and is responsible for computing three stencils with 37 flops per each. Assuming DP data, the flop-per-byte ratio for each kernel is $37 * 3/((5 + 3) * 8) = 1.73$, while the minimum flop-per-byte ratio required by NVIDIA Tesla K40m to achieve the maximum performance is 5.2 [28]. This estimation confirms the memory-bounded nature of MPDATA stencils when implementing on GPUs.

This preliminary analysis shows also that the largest performance gain can be achieved by providing the memory optimizations. The main challenge is to find a solution where data transfers from the global to shared memory or register file are minimized.

4.3. Performance analysis based on GPU global memory transactions

Our idea of adaptation is based on an appropriate distribution of stencils across GPU kernels in order to minimize the number of memory transactions between shared and global memory. For this aim, we propose a method where a different number of kernels is considered. In each configuration, a single kernel processes a different number of stencils. We estimate the number of memory transactions for each configuration, and then select a configuration where the number of memory transactions is minimized.

The starting point for the optimization is a comprehensive analysis of data flows when executing MPDATA. The distribution of computational tasks is preceded by the estimation of the shared memory utilization, sizes of halo areas, as well as data dependencies between and within stencils. Based on such an extensive analysis, we are able to specify the most favorable number of kernels, as well as select an optimal distribution of stencils across kernels, and the sizes of CUDA blocks for each kernel. As a consequence, an efficient load balancing is preserved, and data communication is minimized and well structured.

To illustrate our method, we explain the procedure of estimating the number of kernels for the most time-consuming part of the MPDATA algorithm. In this example, we have three stencils, where the first of them is shown in Figure 3. It is a part of kernel B. The remaining two stencils are of the same algorithmic complexity.

In our approach, the following scenarios are considered:

- distribution of computation (three stencils) across two kernels B and C; and
- compression of computation within one kernel BC.

The compression of computation increases hardware requirements for CUDA blocks, and decreases the GPU occupancy. However, it allows for reducing the number of temporary matrices, and thereby, it decreases the memory traffic.

At the beginning of the analysis, we need to estimate the cost of access to a certain matrix, for each scenario. We assume that a CUDA block is of size $g_1 \times g_2$, matrices are processed according to Figure 1. Moreover, the halo areas with size 1 are located on the four boundaries of each CUDA block (Figure 5). The number of elements that need to be transferred from the GPU global memory to shared memory (or register file) is given by the following formula:

$$S_{el} = g_1 * g_2 + 2 * g_2 + 2 * g_1. \qquad (1)$$

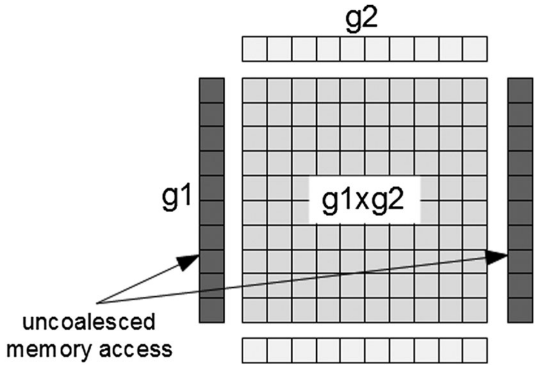

Figure 5. XY-tile of a certain matrix, assigned to a CUDA block, with halo areas.

Taking into account the size of global memory transaction (128B) [29], element of the matrix (8B), as well as parameters g_1 and g_2, we can estimate the number of memory transactions within each CUDA block, for a single XY-tile of the matrix (Figure 5):

$$S_{tr} = g_1 * top(g_2/16) + 2 * top(g_2/16) + 2 * g_1, \quad (2)$$

where $top(x)$ returns rounded up value of x, and value of 16 is the ratio of the size of global memory transaction to the size of element of the matrix. In this approach, the addresses of vertical halo areas are not coalesced.

Table III shows the cost of the memory access for the first scenario, where the XY-tile is of size 392×256. The access overhead is the ratio between the actual number of transactions required to transfer the matrix and a lower limit of memory transactions. This limit corresponds to an infinite size of shared memory, which allows for eliminating halo areas completely. For a tile of size 392×256, the lower limit of transactions is $392 * 256/16 = 6272$. Based on Table III, we conclude that the minimum access overhead is 159.95% for the first scenario, which gives the optimal number of transactions equal to $6272 * 159.95\% = 10032$. Also, this analysis allows us to estimate the most suitable size of CUDA blocks as 6×128.

A similar analysis is made for the second scenario (Table IV). Now, the most suitable size of CUDA blocks is 4×64, while the access overhead is 200%.

Finally, we estimate the cost of access to all the matrices for both scenarios. Figure 6 shows the flow diagram for the kernels B and C. There are five input matrices for the kernel B, and six input matrices for the kernel C. Additionally, there are two output matrices per each kernel. For each matrix, the access overhead is 159.95%, so the total cost of access to all the matrices is $(5 + 6 + 2 + 2) * 1.5995 = 23.99$.

Table III. Analysis of GPU global memory transactions: first scenario.

g1	g2	Blocks per tile	Transactions per block	Transactions per tile	Access overhead [%]
6	128	132	76	10032	159.95
5	128	158	66	10428	166.26
12	64	132	80	10560	168.37
11	64	144	74	10656	169.90
10	64	160	68	10880	173.47
9	64	176	62	10912	173.98
4	128	196	56	10976	175.00
8	64	196	56	10976	175.00

GPU, graphics processing unit.

Table IV. Analysis of GPU global memory transactions: second scenario.

g1	g2	Blocks per tile	Transactions per block	Transactions per tile	Access overhead [%]
4	64	392	32	12544	200.00
3	64	524	26	13624	217.22
2	128	392	36	14112	225.00
8	32	392	36	14112	225.00
7	32	448	32	14336	228.57
6	32	528	28	14784	235.71
5	32	632	24	15168	241.84
2	64	784	20	15680	250.00

GPU, graphics processing unit.

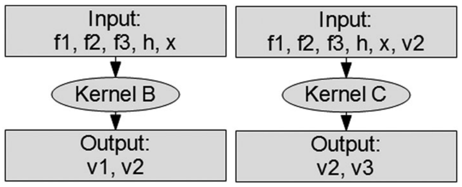

Figure 6. Flow diagram for kernels B and C.

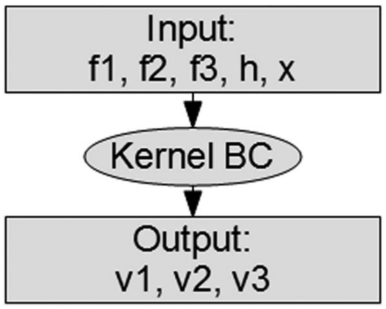

Figure 7. Flow diagram for BC kernel.

Table V. Summary of MPDATA analysis.

	Kernels B and C	Kernel BC	Ratio
Occupancy	37.50%	25.00%	1.5
Access overhead	159.95%	200.00%	0.80
# of matrices	15	8	1.85
Total cost of access	23.99	16	1.50

MPDATA, multidimensional positive definite advection transport algorithm.

Table VI. Performance results for both scenarios.

Kernel	Mflops per scenario	Time per scenario [ms]	Performance [Gflop/s]	Speedup
B and C	963.4	15.49	62.29	1
BC	847.8	10.45	80.95	1.48

The flow diagram for the second scenario is shown in Figure 7. Here, we have five input and three output matrices. The access overhead is 200%. So, the total cost of access to the all matrices is estimated as $(5 + 3) * 2.0 = 16$.

Table V shows the summary of the MPDATA analysis, taking into account the considered scenarios. It can be concluded that the number of transactions to the GPU global memory can be greatly reduced by merging two kernels into one. The expected performance speedup is about 1.5.

Table VI presents the performance results for both scenarios. In our tests, we used a single NVIDIA Tesla K40m GPU. The MPDATA algorithm is tested for the grid of size $392 \times 256 \times 64$. The achieved results are far from the GPU peak performance because of the complexity of the algorithm, intensive data traffic to the global memory, strong instruction and data dependencies, as well as the shared memory size limitation.

Based on Table VI, we conclude that our method of compressing stencils across kernels allows for increasing the MPDATA performance by about 1.48 times, for the two considered kernels. Such a speedup is very close to the expected.

A similar analysis is provided for other combinations of kernels. In consequence, the best configuration of the MPDATA algorithm is the distribution of its 17 stencils across 4 GPU kernels.

4.4. Performance results for MPDATA algorithm

To examine the performance of the MPDATA algorithm, a number of numerical experiments have been performed. We compare the parallel performance of the original CPU version of MPDATA with two GPU implementations. The CPU used in the tests is Intel Core i7-3770 with 3.4 GHz clock frequency, consisting of four cores. The input data were defined as an array of random values. The first GPU version (naive one) is not optimized for GPU global memory transactions, so the MPDATA stencils are distributed across six kernels. The second version (improved one) is based on our performance analysis, and all the stencils are compressed into four kernels.

The sizes of grid range from $16 \times 16 \times 16$ to $512 \times 512 \times 64$. The CPU tests have been performed for both a sequential configuration with a single core, and parallel version with four cores. All the tests have been conducted for 100 time steps. The performance results are listed in Table VII, which shows that our GPU implementation allows for achieving speedup of about 7 over the parallel CPU version, and about 22 over the sequential version. The speedups for all the grid sizes are shown in Figure 8. The use of GPU is profitable for the grid sizes greater than or equal to $64 \times 64 \times 16$, because computations on smaller grids do not allow us to take full advantages of GPU resources. It is worth noting that the improved GPU version gives 1.35 speedup over the naive GPU implementation.

Table VII. Execution time and speedup for MPDATA algorithm, where T_1 corresponds to MPDATA execution using only a single CPU core.

	CPU	GPU		Speedup			
Grid size	4 cores T_2 [s]	naive T_3 [s]	improved T_4 [s]	T_1/T_3	T_1/T_4	T_2/T_3	T_2/T_4
16 x 16 x 16	0.016	0.087	0.056	0.51	0.79	0.18	0.29
32 x 32 x 16	0.048	0.1	0.059	1.64	2.78	0.48	0.81
64 x 64 x 16	0.192	0.112	0.076	5.68	8.37	1.71	2.53
64 x 64 x 64	0.776	0.366	0.237	7.15	11.04	2.12	3.27
128 x 128 x 64	3.316	0.792	0.567	13.24	18.49	4.19	5.85
128 x 128 x 128	6.624	1.583	1.166	13.15	17.85	4.18	5.68
256 x 256 x 64	12.868	2.56	1.838	15.77	21.97	5.03	7.00
256 x 256 x 128	24.373	5.221	3.728			4.67	6.54
256 x 256 x 256	52.776	10.516	7.525			5.02	7.01
512 x 512 x 64	50.743	9.689	7.027			5.24	7.22

MPDATA, multidimensional positive definite advection transport algorithm.

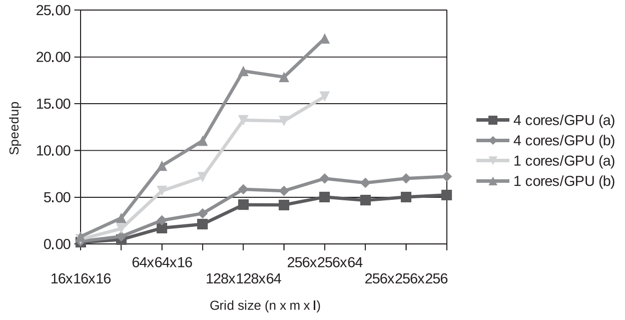

Figure 8. Speedups of graphics processing unit over central processing unit for multidimensional positive definite advection transport algorithm.

```
gcrk() {
    prforc()
    divrhs()
    precon()
    reduction()
    laplc()
    for it=1..solver_iterations {
        reduction()
        if(exit) quit_for_loops;
        precon()
        laplc()
    }
}
```

Figure 9. The body of the elliptic solver code.

5. ADAPTATION OF ELLIPTIC SOLVER COMPUTATION TO GPU ARCHITECTURE

5.1. *Main objectives*

This section describes the parallelization approach to the GCR elliptic solver. The body of the elliptic solver code consists of five major routines (Figure 9). The main routine advances the solution iteratively by calling other major computational routines. The routines *prforc* and *divrhs* initialize the solver. The former routine evaluates the first guess of the updated velocity, by combining the explicit part of the solution and the estimation of the generalized pressure gradient, while imposing an appropriate boundary condition. The latter routine evaluates the density weighted divergence of the velocity, and thus, the initial residual error of the elliptic problem for pressure is computed.

Among the most computationally intensive routine of the GCR solver is *laplc* that iteratively evaluates the generalized Laplacian operator (a combination of divergence and gradient) acting on residual errors. Another important part of the solver is the *precon* routine that accelerates the convergence of the variational scheme. By performing the direct matrix inversion in the vertical dimension of the grid, it is especially useful for large-scale simulations on thin spherical shells with grids characterized by a large anisotropy. The routine *precon* employs the sequential Thomas algorithm [32] to solve tridiagonal systems of equations with the right-hand side consisting of the horizontal divergence of the generalized horizontal gradient. This gradient is evaluated by *nablaCnablaxy*, which also belongs to the most computationally intensive routines of the GCR solver. With regard to the data access pattern, the computational loops within the elliptic solver can be simply divided into three categories: (1) reductions; (2) implicit methods of the Thomas algorithm; and (3) explicit methods of the stencils.

5.2. *Preliminary analysis of numerical performance*

To evaluate the numerical performance of the developed GPU implementation of the GCR solver, we use a standard metric for measuring the computational intensity. This metric, denoted as Q [33], is defined for each routine as the ratio of the number of arithmetic operations to the amount of required data. This amount is expressed in bytes for DP quantities, and takes into account both reading and writing operations. The constant values are not included in the metric, as they can be easily cached by the compiler in the registers.

Table VIII presents the computational intensity Q of an unoptimized GPU code for the main routines of the elliptic solver. The unoptimized GPU code is strictly based on the CPU code. In the Kepler K40m accelerator, the time needed for computing 5 DP operations is equal to the read time of 1 byte of data from the global memory. Thus, the performance of all functions in the elliptic solver is strongly bounded by the global memory bandwidth. Each of them requires more than 10 bytes of data to compute one operation.

Therefore, increasing the computational intensity is crucial for improving the performance of the elliptic solver. This can be achieved by both reducing the memory traffic and increasing the number of arithmetic operations. To address these issues, we analyze the data dependencies between

Table VIII. Computational intensity per grid element.

Platform		precon	prforce	divrhs	laplc	GCR
		Routine name				
Unoptimized GPU	# of operations	18	27	15	33	297
	# of bytes	255	289	170	604	4329
	Q_{unGPU}	0.07	0.09	0.09	0.05	0.07

GPU, graphics processing unit.

Algorithm 1 Original 2D horizontal decomposition

```
1: procedure HorizontalDecomposition($f$, $p$, $p33$, $r$, $dni$, $e$)
2:     for $k = 3..l, j = 1..mp, i = 1..np$ do
3:         $f(i,j,k) = (p33(i,j,k-1) * f(i,j,k-2) + r(i,j,k)) * dni(i,j,k)$
4:     end for
5:     for $k = l-2..1, j = 1..mp, i = 1..np$ do
6:         $p(i,j,k) = e(i,j,k) * p(i,j,k+2) + f(i,j,k)$
7:     end for
8: end procedure
```

the computational loops. Based on the analysis, we try to merge them to increase the computational intensity. The next two subsections describe the methodology of optimization, which is applied to selected routines of the elliptic solver.

5.3. *Preconditioner optimization*

The preconditioner employs the sequential Thomas algorithm [32] to solve tridiagonal systems of equations. There are two different implementations of this algorithm, depending on the parallelization method or, equivalently, on the domain decomposition scheme. The standard version is dedicated to the 2D horizontal decomposition. For implementations that use the 3D domain decomposition, the version based on the recurrence doubling approach is preferable. A detailed description of the recurrence doubling version can be found in [34].

Algorithm 1 shows the structure of the 2D horizontal decomposition where np and mp are subdomain sizes in the horizontal directions. The detailed description of all the variables can be found in [35]. The data dependencies in both loops imply that this decomposition is more suitable for the usage on a single graphic card than the 3D decomposition. The numerical scheme of our GPU implementation is shown in Algorithm 2.

To minimize the global memory traffic, two loops that compute the matrices f and p have to be join into a single CUDA kernel, where each thread operates on a vertical column of the size l. The computation of the matrix f is performed by using registers to eliminate global memory accesses caused by the reference to the $f(k-2)$ element. The shared memory could be used to cache access to the $f(k-2)$ element, but still the registers would provide a lower access latency. At the end, the f matrix is saved in the shared memory. To compute the p matrix, the f matrix is read from the shared memory, and similarly, the usage of registers eliminates the $k+2$ global memory access pattern. These optimizations improve the computational intensity of the preconditioner routine from 0.07 to 0.132.

5.4. *Stencil optimization*

Here, we demonstrate our approach to optimize the stencil computation by the example of computing the 3D Laplacian, which involves six computational loops defined as stencils. In these stencils, each element has access only to one neighboring element on both sides, for the $i-th$, $j-th$,

Algorithm 2 Optimized 2D horizontal decomposition on GPU

```
procedure HORIZONTALDECOMPOSITIONGPU($f$, $p$, $p33$, $r$, $dni$, $e$)
    $fsm(1) = freg1 = f(1); fsm(2) = freg0 = f(2)$
    for $k = 3..l$ do
        $freg2 = freg1; freg1 = freg0$
        $fsm(k) = freg0 = (p33(k-1) * f(k-2) + r(k)) * dni(k)$
    end for
    $preg1 = p(l); preg0 = p(l-1)$
    for $k = l-2..1$ do
        $preg2 = preg1; preg1 = preg0$
        $p(k) = preg0 = e(k) * preg2 + fsm(k)$
    end for
end procedure
```

and $k - th$ dimensions (Figure 10a), where the domain boundaries have to be handled specifically. To improve the computational intensity, we join these computational loops into one kernel, see Figure 10b.

This transformation creates a larger stencil with the new access pattern. In this new pattern, each element accesses a distant neighbor that is one element away from the currently updated position. The distant neighbors, similarly to stencils before the transformation, have to be accessed on both sides for the i-th, j-th, and k-th directions. The $2.5D$ blocking technique described in Section 4 efficiently caches the neighbor elements in the shared memory and reduces the global memory traffic. That is, we decompose the domain into 3D blocks, where a horizontal tile defined by the i-th and j-th directions is written to the shared memory, and the values defined by the k-th direction are written to the registers. The additional data, called halo, are written to the shared memory and register in order to compute properly elements on the 3D block boundaries. Thus, some elements are read more than once, and their number depends on the 3D block sizes. These sizes are mainly constrained by the size of the shared memory and the number of available registers. We minimize the number of the global memory transactions by selecting suitable sizes of the 3D blocks.

The memory transactions are defined at the granularity of L2 cache lines. The 3D block sizes that are close in shape to cube limit the halo size. Moreover, the 3D block sizes affect the number of active warps (Section 3). The larger number of active warps can be obtained by larger 3D block or by the higher number of smaller 3D blocks. The large number of concurrently executed warps allows us to hide a latency caused by accesses to the global memory. Therefore, in this work, a runtime method is developed to select the 3D block sizes (Algorithm 3) that optimize both the halo size and the number of active warps.

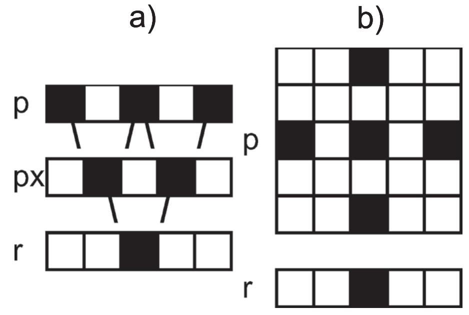

Figure 10. Access patterns for stencils before (a) and after (b) optimization: (a) two stencils computing px (partial derivative of p in x direction) and r (partial derivative of px in x direction) in the i-th direction; the same applies to j-th and k-th direction; (b) the stencil pattern in the i-th and j-th directions after transformation.

Algorithm 3 Dynamic calculation of block size

```
for $blockSize = maxThreadsPerBlock..warpSize, -warpSize$ do
    $nBlocks = maxActiveBlocksPerSMX(blockSize, sharedMemPerTh, regsPerTh)$
    $nActiveWarps = (nBlocks * blockSize)/warpSize$
    if then $activeWarps == maxActiveWarps$
        $list.push(blockSize)$
    else
        if then $activeWarps > maxActiveWarps$
            $list.clear()$
            $list.push(blockSize)$
            $maxActiveWarps = activeWarps$
        end if
    end if
end for
for $blockSize\ in\ list$ do
    $nTransactions = calcTransactions(blockSize)$
    if then $nTransactions < bestNTransactions$
        $bestNTransactions = nTransactions$
        $bestBlockSize = blockSize$
    end if
end for
```

In Algorithm 3, $maxThreadsPerBlock$ represents the maximum block size available for the target GPU architecture. For each block size, the function $maxActiveBlocksPerSMX$ estimates the SMX occupancy, based on the number of shared memory bytes, and the number of registers utilized by each thread. For the Kepler architecture, the largest occupancy can be achieved with 64 active warps per one SMX. Next, for the selected blocks, the $calcTransactions$ function calculates the number of transactions to the global memory. The largest block size with the lowest number of transactions to the global memory is selected finally. Thus, we maximize the SMX occupancy and minimize the required memory traffic. The selected sizes of blocks for the 3D Laplacian on cubic domains are shown in Table IX. The optimal sizes of blocks are determined by the brute-force search, whereas the dynamic sizes are selected by our method. Figure 11 presents the performance comparison for the 3D Laplacian using both approaches. Our method achieves no less than 90% and 95% of the optimal performance for small and large domains, respectively.

To sum up, all the described optimizations reduced the number of bytes moved to/from global memory per element by a half. In consequence, they improved the computational intensity of the *laplc* routine from 0.05 to 0.132.

5.5. Testing performance

The performance of the new GPU implementation is tested using the dual-socket Intel Xeon E5-2670 CPU with 16 cores clocked at 2.6 GHz, and Kepler K40m GPU. To compare the performance of CPU and GPU processors solely, the data transfer time between the host memory and GPU is not included into the GPU timings. During computations, all the data are available in the GPU global memory, and the full data set is asynchronously transferred to CPU after every 100 iterations.

Figure 12 shows the performance of the CPU and GPU implementations obtained in two different numerical tests. Test 1 refers to the flow in a cube with triply-periodic boundaries. Test 2 is a simulation of the flow on a sphere with non-periodic boundaries in the vertical. Additionally, test 2 utilizes the preconditioner in the horizontal direction. Both tests were conducted with one GCR iteration. It turned out that the GPU implementation is about four times faster than the CPU implementation. The speedup grows with the increase of domain sizes as the kernels utilize GPU resources more

Table IX. Selected sizes of the block for the 3D Laplacian.

	Block							
Domain	Optimal				Dynamic			
N^3	X	Y	Z	GFlop/s	X	Y	Z	GFlop/s
64^3	64	8	16	38.6	64	16	5	34.9
128^3	128	7	64	56.3	128	8	20	56.1
192^3	112	6	96	56.1	128	8	66	52.2
256^3	96	7	64	56.8	96	7	256	55.9
320^3	32	16	80	54.4	32	32	161	52.9
384^3	32	26	48	56.6	32	32	384	53.2
448^3	32	32	56	55.1	32	32	448	53.4
512^3	32	32	64	55.7	32	32	512	53.8
576^3	64	16	576	55.3	32	32	576	54.2

3D, three dimensional.

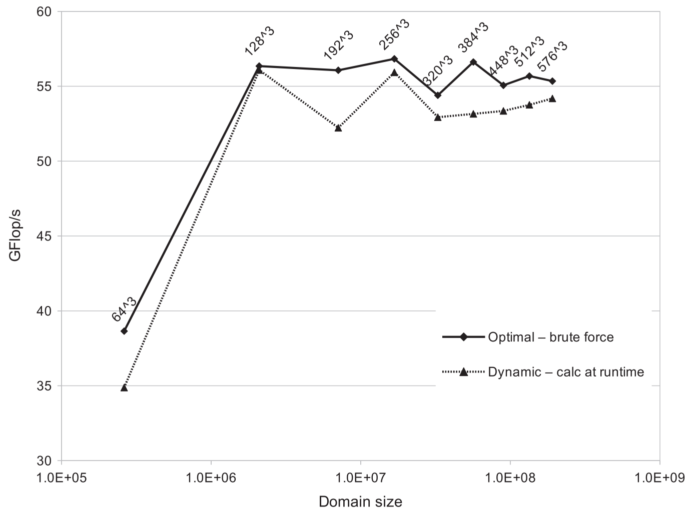

Figure 11. Performance tests for the 3D Laplacian using various sizes of the blocks.

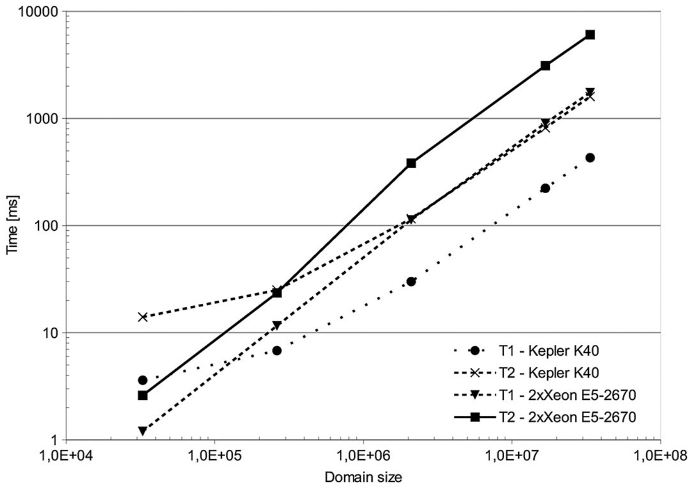

Figure 12. Performance results – test 1 and test 2.

Table X. Comparison of the computational intensity.

Version		precon	prforce	divrhs	laplc	GCR
		Routine name				
Unoptimized GPU	# of operations	18	27	15	33	297
	# of bytes	255	289	170	604	4329
	Q_{unGPU}	0.07	0.09	0.09	0.05	0.07
Optimized GPU	# of operations	22	27	17	32	291
	# of bytes	167	154	113	242	2443
	Q_{oGPU}	0.132	0.175	0.150	0.132	0.119
Q_{oGPU}/Q_{unGPU} [%]		187	188	171	242	174

efficiently. The GPU code achieves 72% of the theoretical performance for test 1, and up to 86% for test 2. The theoretical performance is defined as the product of the computational intensity and measured global memory bandwidth. The theoretical performance corresponds to the ideal situation in which data transfers and computations perfectly overlap. This occurs only if there are no memory latency effects, and when the bandwidth of the global memory is completely saturated.

Table X compares the computational intensity Q for the main routines of the elliptic solver, for both the unoptimized and optimized GPU codes. The optimized GPU implementation has a factor of 1.7 higher computational intensity than the unoptimized one. Thus, the number of required bytes of data per grid element is significantly reduced.

6. SYNERGY BETWEEN EULAG COMPONENTS

The previously described MPDATA and GCR solver algorithms are implemented in the C/C++ language, while the original EULAG code is developed in the FORTRAN 77 language. Several steps are performed to merge all the implementations. In the main EULAG loop, the MPDATA and GCR routines are called in each time step. Therefore, the wrapper routines are added to call the C routines from the FORTRAN 77 code. As FORTRAN variables, including characters and integer values, are passed by reference, they must be carefully handled. A FORTRAN subroutine call has to explicitly pass the common block variables. Another important aspect is the difference in the indexing syntax of arrays. In FORTRAN, the array index that represents elements in contiguous memory regions is written first, so we have $array(i, j, k)$, whereas in C, this index is written last $array[k][j][i]$. Our C implementation treats 3D FORTRAN arrays as one-dimensional arrays. The array elements are accessed with $array[i + j * n + k * n * m]$ syntax, where n and m are the array dimensions. The memory order in C is the same as in FORTRAN, to avoid the cost related to the data transposition.

The host compilers are used to compile the FORTRAN and C routines, whereas for the GPU code, the CUDA compiler is utilized. To link the obtained object files, the C routines that are called from FORTRAN have to be compatible with the FORTRAN naming convention. For instance, the C routine has to be declared as $gcrk\_()$, and called from FORTRAN as $call\ gcrk()$. This is performed in header file in C by the macro definition that change the routine name.

7. 3D INCOMPRESSIBLE TURBULENT FLOW: VALIDATION TEST CASE

The new GPU implementation of the GCR solver is validated using a standard benchmark test case for incompressible flow solvers. We simulate decaying turbulence of a homogeneous incompressible fluid. Here, only the simplified setup proposed by Taylor and Green [36] is considered. This problem was originally used to illustrate processes of grinding down of large eddies into smaller ones. The initial conditions for the velocity are slightly modified compared to the original work [36], namely

$$\begin{aligned} u &= sin(ax)cos(by)cos(cz) \\ v &= cos(ax)sin(by)cos(cz) \\ w &= sin(ax)cos(by) \end{aligned} \tag{3}$$

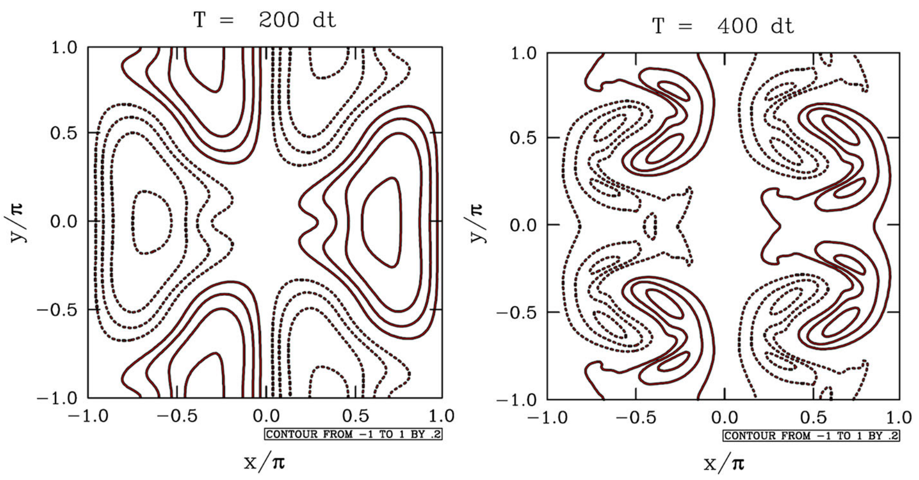

Figure 13. Time evolution of the vertical component of velocity, displayed in the bottom ($z = -\pi$) horizontal cross section through the domain. Black lines correspond to simulations with standard CPU version of EULAG. Red contours represent solution from the new GPU implementation. The dashed lines indicate negative values.

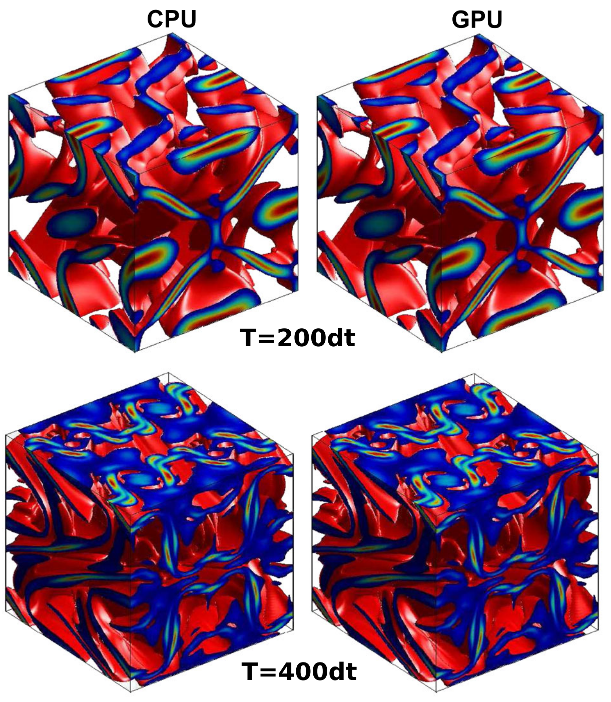

Figure 14. Isosurfaces of vorticity magnitude from simulation at grid 128^3. On the left side are results from simulations performed with the traditional (CPU) version of EULAG. On the right side are results from the new GPU implementation. The top panels present the flow after 200 time steps. The two bottom panels show the vorticity field after 400 time steps.

where $a = 2/(n-1)dx$, $b = 2/(m-1)dy$, and $c = 2/(k-1)dz$. Here m, n, and k are integers, whereas dx, dy, and dz are grid spacings in the corresponding directions. Our computational domain is a triply-periodic cube of length 2π. Grid points are uniformly distributed in each spatial direction. The size of the computational grid is 128^3. There is no external forcing so the flow is driven by the turbulent energy cascade.

We compare results from simulations performed with two versions of EULAG. The first simulation has been performed using the traditional CPU architecture and the standard version of EULAG.

Table XI. Quantitative comparison between simulations performed with CPU and GPU. The table shows L_1 and L_2, L_∞ norms computed for the vorticity fields after 200 and 400 time steps.

	L_1	L_2	L_∞
$T = 200$ dt	257.92×10^{-9}	0.3586×10^{-6}	0.0730×10^{-14}
$T = 400$ dt	587.13×10^{-9}	2.4420×10^{-6}	2.5093×10^{-14}

GPU, graphics processing unit.

To perform the second simulation, we used the new GPU implementation of the GCR solver, and the graphic card NVIDIA Kepler K40m. First, we compare the 2D velocity flow field obtained in both simulations. Figure 13 shows the vertical component of velocity in the bottom horizontal tile. There is a good agreement between solutions computed with the standard version of EULAG and the new GPU implementation. The black and red lines precisely overlap. To compare the results from simulations in the entire domain, the 3D visualization of the vorticity flow field has been prepared. The results are shown in Figure 14. Again, we confirm a good agreement between simulations performed using two different hardware platforms.

To quantify the difference between simulations performed with CPU and GPU, the L_1, L_2, and L_∞ norms of vorticity magnitude ($|\omega|$) have been computed. The norms are defined as

$$L_1(|\omega|) = \left\{ \frac{1}{N^3} \sum_{i,j,k=1}^{N} ||\omega(x_i, y_j, z_k)^{CPU}| - |\omega(x_i, y_j, z_k)^{GPU}|| \right\} \tag{4}$$

$$L_2(|\omega|) = \left\{ \frac{1}{N^3} \sum_{i,j,k=1}^{N} \left[|\omega(x_i, y_j, z_k)^{CPU}| - |\omega(x_i, y_j, z_k)^{GPU}|\right]^2 \right\}^{0.5} \tag{5}$$

$$L_\infty(|\omega|) = \left\{ \frac{1}{N^3} \max_{1\leqslant i,j,k\leqslant N} ||\omega(x_i, y_i, z_k)^{CPU}| - |\omega(x_i, y_j, z_k)^{GPU}|| \right\} \tag{6}$$

where i, j, and k are the grid indices.

Table XI shows values of these norms computed for the $|\omega|$ fields after 200 and 400 time steps. We conclude that the values are rather small. Comparing the L_1 values to the mean vorticity ($\approx 4\times10^{-3}$ at T=200 dt), we find that the differences between GPU and CPU implementations are four orders of magnitude smaller then the average $|\omega|$. As expected, because of round-off errors, resulting from using different architectures, the values of the norms increase with time. However, with respect to the average values of vorticity, the norms remain almost constant.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we reported on our efforts in adapting the multiscale model EULAG to modern GPU-based architectures. A performance model of the CFD code EULAG is proposed to analyze the hierarchy of communications at registers, shared, and global memories. The bottlenecks of the algorithm are identified. The two main modules of EULAG, namely the advection algorithm MPDATA and iterative elliptic pressure solver GCR, have been redesigned, and the new organization of computations has been implemented. It is shown that the GPU utilization is mostly limited by the number of memory transactions and latency of arithmetic operations. The GPU tuning of MPDATA (stencil optimization) is based on the experimental tuning of 'the best' number of kernels and CUDA block sizes for each kernel.

The new implementation performs better than the conventional CPU code, and can take advantage of modern heterogeneous architectures. The scalability tests were performed using the NVIDIA Tesla K40m graphic card. It was found that the speedup grows with the domain size as the kernels can utilize GPU resources more efficiently. Based on the analysis of GPU implementation of the

EULAG model, we can conclude that the GPU utilization is mostly limited by the number of memory transactions and latency of arithmetic operations. The proposed methods allow for estimating 'the best' number of kernels, as well as easy selection of CUDA block sizes for each kernel.

The analysis of GPU global memory transactions allows us to minimize data transfers from/to the GPU global memory. The proposed approach to kernel processing with queues of data chunks placed in registers and shared memory increases the data locality significantly. The achieved performance results show that the GPU implementation is profitable for mesh sizes greater or equal $64 \times 64 \times 16$, because smaller grids weakly utilize the GPU resources.

The developed GPU version of the GCR solver improves the computational intensity Q from 0.05 to 0.132. Two test problems (test 1 and test 2) are run to demonstrate the achieved speedup. It is observed that the GPU implementation is about four times faster than the CPU one. The GPU code reaches 72% of the theoretical performance for test 1 and 86% for test 2.

Adaptation of the EULAG model to modern architectures is still under development. The future work will focus on expansion of the implementation across a cluster of CPU–GPU nodes. An important aspect of future research is providing the full synergy between the MPDATA algorithm and elliptic solver GCR. The particular attention will be paid to implementation of EULAG using the OpenCL [37] standard to provide the code portability across different platforms.

Porting the multiscale model EULAG to modern architectures opens bright prospects for a further progress in fundamental research, and in applied fields that are closely related to the CFD. It is expected that the improved performance will enable us to reproduce more faithfully meteorological processes occurring in the real atmosphere. The new developments will allow for performing simulations in a larger domain, and thereby extends the range of scales what in turn may result in a more reliable operational weather forecast.

ACKNOWLEDGEMENT

This work was supported by the Polish National Science Center under grant no. UMO-2011/03/B/ST6 /03500 and by the project 'Towards peta-scale NWP for Europe' realized within the Homing Plus programme of Foundation for Polish Science, co-financed from European Union, Regional Development Fund.

The authors are grateful to Czestochowa University of Technology, Poznan Supercomputing and Networking Center, and MEGWARE Computer Vertrieb und Service GmbH, for granting access to hybrid CPU–GPU platforms.

REFERENCES

1. Skamarock WC. Evaluating mesoscale NWP models using kinetic energy spectra. *Monthly Weather Review* 2004; **132**:3019–3032.
2. Wyrzykowski R, Szustak L, Rojek K. Parallelization of 2D MPDATA EULAG algorithm on hybrid architectures with GPU accelerators. *Parallel Computing* 2014; **40**:425–447.
3. Vuduc R, Chandramowlishwaran A, Choi J, Guney M, Shringarpure A. On the limits of GPU acceleration. *HotPar10: Proceedings of the 2nd USENIX Conference on Hot Topics in Parallelism. USENIX Association*, Berkeley, CA, 2010; 13.
4. Unat D, Cai X, Baden SB. Mint: realizing CUDA performance in 3D stencil methods with annotated C. *ICS '11 Proceedings of the International Conference on Supercomputing*, Tucson, Arizona, USA, 2011; 214–224.
5. The Consortium for Small-scale Modeling. (Available from: http://www.cosmo-model.org) [Accessed on 1 August 2014].
6. Prusa JM, Smolarkiewicz PK, Wyszogrodzki AA. EULAG, a computational model for multiscale flows. *Computers & Fluids* 2008; **37**:1193–1207.
7. Smolarkiewicz PK, Kuhnlein C, Wedi NP. A consistent framework for discrete integrations of soundproof and compressible PDEs of atmospheric dynamics. *Journal of Computational Physics* 2014; **263**:185–205.
8. Smolarkiewicz PK. Multidimensional positive definite advection transport algorithm: an overview. *International Journal for Numerical Methods in Fluids* 2006; **50**:1123–1144.
9. Smolarkiewicz PK, Szmelter J. A nonhydrostatic unstructured-mesh soundproof model for simulation of internal gravity waves. *Acta Geophysica* 2011; **59**:1109–1134.
10. Schafer A, Fey D. High performance stencil code algorithms for GPGPUs. *Computer Science* 2011; **4**:2027–2036.
11. Krotkiewski M, Dabrowski M. Efficient 3D stencil computations using CUDA. *Parallel Computing* 2013; **39**: 533–548.

12. Kamil S, Husbands P, Oliker L, Shalf J, Yelick K. Impact of modern memory subsystems on cache optimizations for stencil computations. *Proceedings of the 2005 Workshop on Memory System Performance*, Chicago, IL, USA, 2005; 36–43.
13. Datta K, Kamil S, Williams S, Oliker L, Shalf J, Yelick K. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM* 2009; **51**:129–159.
14. Rivera G, Tseng CW. Tiling optimizations for 3D scientific computations. *SC'00 Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, Dallas, Texas, USA, 2000; 32.
15. Treibig J, Wellein G, Hager G. Efficient multicore-aware parallelization strategies for iterative stencil computations. *Journal of Computational Science* 2011; **2**:130–137.
16. Szustak L, Rojek K, Gepner P. Using Intel Xeon Phi coprocessor to accelerate computations in MPDATA algorithm. *Proceedings of the PPAM 2013 Conference Lecture Notes in Computer Sciences*, Warsaw, Poland, 2013; 582–592.
17. Hager A, Wellein G. *Introduction to high performance computing for science and engineers*. CRC Press: Boca Raton, Florida, USA, 2011.
18. Ciznicki M, Kopta P, Kulczewski M, Kurowski K, Gepner G. Elliptic solver performance evaluation on modern hardware architectures. *Proceedings of the PPAM 2013 Conference Lecture Notes in Computer Sciences*, Warsaw, Poland, 2013; 155–165.
19. Rojek K, Szustak L. Parallelization of EULAG Model on multicore architectures with GPU accelerators. *Lecture Notes in Computer Science* 2012; **7204**:391–400.
20. Wyrzykowski R, Rojek K, Szustak L. Using Blue Gene/P and GPUs to accelerate computations in the EULAG model. *Lecture Notes in Computer Science* 2012; **7116**:662–670.
21. Wyrzykowski R, Szustak L, Rojek K, Tomas A. Towards efficient decomposition and parallelization of MPDATA on hybrid CPU-GPU cluster. *Proceedings of the LSSC 2013 Conference Lecture Notes in Computer Sciences*, Sofia, Bulgaria, 2013; 457–464.
22. De la Cruz R, Araya-Polo M, Cela JM. *Introducing the semi-stencil algorithm*. Springer Berlin Heidelberg, 2010.
23. Datta K, Murphy M, Volkov V, Williams S, Carter J, Oliker L, Patterson D, Shalf J, Yelick K. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. *SC'08 Int. Conf. on High Performance Computing, Networking, Storage and Analysis*, Austion, USA, 2008; 1–12.
24. Venkatasubramanian S, Vuduc R. Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems. *ICS'09 Proceedings of the 23rd International Conference on Supercomputing*, Yorktown Heights, NY, USA, 2009; 244–255.
25. Nguyen A, Satish N, Chhugani J, Changkyu K, Dubey P. 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE Computer Society, New Orleans, USA, 2010; 1–13.
26. Kurowski K, Kulczewski M, Dobski M. Parallel and GPU based strategies for selected CFD and climate modeling models. *Environmental Science and Engineering* 2011; **3**:735–747.
27. Cecilia JM, Garcia JM, Ujaldon M. CUDA 2D stencil computations for the Jacobi method. *Lecture Notes in Computer Science* 2012; **7133**:173–183.
28. NVIDIA Kepler Compute Architecture. (Available from: http://www.nvidia.com/object/nvidia-kepler.html) [Accessed on 1 August 2014].
29. Best practices guide: CUDA toolkit documentation. (Available from: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide) [Accessed on 1 August 2014].
30. Duguet F. Kepler vs Xeon Phi: our benchmark [source code included], 2013. (Available from: http://www.hpcmagazine.eu/hpc-labs/kepler-vs-xeon-phi-our-benchmark-source-code-included/).
31. Wyrzykowski R, Rojek K, Szustak L. Performance analysis for stencil based 3D MPDATA algorithm on GPU architecture. *Lecture Notes in Computer Science (in print);*.
32. Strikwerda J. Finite difference schemes and partial differential equations. *Society for Industrial and Applied Mathematics* 2004; **ch. 3.5.**:88–91.
33. Williams S, Waterman A, Patterson D. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM* 2013; **52**(4):65–76.
34. Wyszogrodzki AA, Piotrowski ZP, Grabowski WW. Parallel implementation and scalability of cloud resolving EULAG model. *Lecture Notes in Computer Science* 2012; **7204**:252–261.
35. Piotrowski ZP, Wyszogrodzki AA, Smolarkiewicz PK. Towards petascale simulation of atmospheric circulations with soundproof equations. *Acta Geophysica* 2011; **59**:1294–311.
36. Taylor G, Green A. Mechanism of the production of small eddies from large ones. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, Vol. 158, 1937; 499–521.
37. Munshi A, Gaster BR, Mattson TG, Fung J, Ginsburg D. *OpenCL - Programming Guide*. Addison-Wesley, 2011.