Parallel Computing 40 (2014) 425–447

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Parallelization of 2D MPDATA EULAG algorithm on hybrid architectures with GPU accelerators

Roman Wyrzykowski *, Lukasz Szustak, Krzysztof Rojek

Institute of Computer and Information Sciences, Czestochowa University of Technology, Poland

ARTICLE INFO

Article history:
Received 4 June 2013
Received in revised form 13 January 2014
Accepted 25 April 2014
Available online 15 May 2014

Keywords:
MPDATA advection algorithm
Stencil computation
GPU accelerators
Hybrid CPU–GPU architectures
Hierarchical decomposition
Autotuning

ABSTRACT

EULAG (Eulerian/semi-Lagrangian fluid solver) is an established computational model developed for simulating thermo-fluid flows across a wide range of scales and physical scenarios. The dynamic core of EULAG includes the multidimensional positive definite advection transport algorithm (MPDATA) and elliptic solver. In this work we investigate aspects of an optimal parallel version of the 2D MPDATA algorithm on modern hybrid architectures with GPU accelerators, where computations are distributed across both GPU and CPU components.

Using the hybrid OpenMP–OpenCL model of parallel programming opens the way to harness the power of CPU–GPU platforms in a portable way. In order to better utilize features of such computing platforms, comprehensive adaptations of MPDATA computations to hybrid architectures are proposed. These adaptations are based on efficient strategies for memory and computing resource management, which allow us to ease memory and communication bounds, and better exploit the theoretical floating point efficiency of CPU–GPU platforms. The main contributions of the paper are:

- method for the decomposition of the 2D MPDATA algorithm as a tool to adapt MPDATA computations to hybrid architectures with GPU accelerators by minimizing communication and synchronization between CPU and GPU components at the cost of additional computations;
- method for the adaptation of 2D MPDATA computations to multicore CPU platforms, based on space and temporal blocking techniques;
- method for the adaptation of the 2D MPDATA algorithm to GPU architectures, based on a hierarchical decomposition strategy across data and computation domains, with support provided by the developed GPU task scheduler allowing for the flexible management of available resources;
- approach to the parametric optimization of 2D MPDATA computations on GPUs using the autotuning technique, which allows us to provide a portable implementation methodology across a variety of GPUs.

Hybrid platforms tested in this study contain different numbers of CPUs and GPUs – from solutions consisting of a single CPU and a single GPU to the most elaborate configuration containing two CPUs and two GPUs. Processors of different vendors are employed in these systems – both Intel and AMD CPUs, as well as GPUs from NVIDIA and AMD. For all the grid sizes and for all the tested platforms, the hybrid version with computations spread across CPU and GPU components allows us to achieve the highest performance. In particular, for the largest MPDATA grids used in our experiments, the speedups of the

* Corresponding author. Tel./fax: +48 343250589.
E-mail address: roman@icis.pcz.pl (R. Wyrzykowski).

http://dx.doi.org/10.1016/j.parco.2014.04.009

hybrid versions over GPU and CPU versions vary from 1.30 to 1.69, and from 1.95 to 2.25, respectively.

1. Introduction

Recent activities of major chip manufacturers (NVIDIA, Intel, AMD, IBM) make it evident [20] that future designs of microprocessors and HPC systems will be hybrid and heterogeneous in nature. These heterogeneous solutions rely on integration of two major types of components in various proportion to speed up computation intensive applications: (i) multicore CPU technology, and (ii) special purpose hardware and massively parallel accelerators. In such a heterogeneous co-processing model the gap with classical multiprocessor architecture is so huge that simply enhancing existing solutions is not an option. As a result, adapting applications to hybrid architectures with accelerators requires revisiting the algorithmic and software stacks. The challenge is [2] to build a parallel application which would be spread across the entire machine, as opposed to simply being offloaded to accelerators.

Currently, the most widespread accelerators are Graphics Processing Units (GPUs) – fast and inexpensive, manycore architectures, which have the computing power of several TFlops [12,20,27]. GPUs allow for creating many thousands of threads, which operate in a SIMD fashion. A tremendous step towards a wider acceptance of GPUs in general purpose computations was development of efficient and easy-to-use software environments such as CUDA and OpenCL [1,10,12,18,25,42]. In practice, however, it is still difficult to achieve a good sustained performance on systems with GPU accelerators. Among the main obstacles are: performance/volume constraints of access to the GPU hierarchical memory, and bottlenecks for communication between GPU and CPU.

EULAG (Eulerian/semi-Lagrangian fluid solver) [31–34] is an established computational model developed by the group headed by Piotr Smolarkiewicz for simulating thermo-fluid flows across a wide range of scales and physical scenarios, such as numerical weather and climate prediction, simulation of urban flows, areas of turbulence, and ocean currents, etc. EULAG is a representative of the class of anelastic hydrodynamic models. The dynamic core of the EULAG model includes the multidimensional positive definite advection transport algorithm (MPDATA) and elliptic solver.

The MPI parallelization of EULAG computations on massively parallel systems and x86-based clusters was thoroughly studied in [26,41], using tens of thousands of cores, or even more than 100 K cores in the case of IBM Blue Gene/Q [45]. Such large numbers of cores are necessary to provide the required execution time, taking into account performance limitations of conventional CPU-based nodes of these systems without using any accelerators. Besides high hardware costs, one of the negative consequences of utilizing large numbers of CPU cores is a large energy consumption [35].

Rewriting the EULAG dynamic core and replacing conventional HPC systems with heterogeneous clusters using accelerators such as GPUs was proposed in [19,29,44], to reduce the hardware cost and energy consumption. In particular, preliminary studies of porting anelastic numerical models to GPU architectures were carried out in [29,44]. As a result, selected parts of EULAG, including 2D MPDATA, were ported to NVIDIA Tesla C1060 and ATI Radeon HD 5870 GPU cards. Also, some directions of parallelization of 2D MPDATA on modern CPU architectures were proposed to improve efficiency of CPU multicore processing, vectorization, and cache reuse. These directions are based on the block decomposition and loop tiling techniques for a sequence of stencil computations performed within the MPDATA algorithm. At the same time, the problem of distributing computations across both CPU and GPU resources of the entire machine has not been studied at all. The issue of adapting the EULAG model to clusters with GPU accelerators was discussed in [19], where the PGI Accelerator compiler was used for the automatic parallization of selected parts of EULAG on NVIDIA GPUs, including the 2D MPDATA algorithm. Apart from not addressing the problem of spreading computations across both CPU and GPU components, another disadvantage of this approach is relaying entirely on the automatic parallization, without any efforts to guide the parallelization process taking into account characteristics of both the underlying algorithms and target architectures.

In this work we focus on investigating aspects of an optimal parallel version of the 2D MPDATA algorithm on modern hybrid architectures with GPU accelerators, where computations are distributed across both GPU and CPU components. In order to better utilize features of such computing platforms, comprehensive adaptations of MPDATA computations to hybrid architectures are proposed. These adaptations are based on efficient strategies for memory and computing resource management, which allow us to ease memory and communication bounds, and better exploit the theoretical floating point efficiency of CPU–GPU platforms. The main contributions of the paper are:

- method for the decomposition of the 2D MPDATA algorithm as a tool to adapt MPDATA computations to hybrid architectures with GPU accelerators by minimizing the number of communication and synchronization requests between CPU and GPU components;
- method for the adaptation of 2D MPDATA computations to multicore CPU platforms, based on space and temporal blocking techniques;

- method for the adaptation of the 2D MPDATA algorithm to GPU architectures, based on a hierarchical decomposition strategy across data and computation domains, with support provided by the developed GPU task scheduler allowing for the flexible management of available resources;
- approach to the parametric optimization of 2D MPDATA computations on GPUs using the autotuning concept, which allows us to provide a portable implementation methodology across a variety of GPUs.

This paper is organized as follows. Related works are outlined in Section 2, while Section 3 presents the target hybrid architectures, as well as the hybrid programming environment used in our research. The basics of the 2D MPDATA algorithm, including characterization of data dependencies, are presented in Section 4, which concludes with exposure of performance limitations for MPDATA on hybrid architectures. Section 5 outlines the MPDATA adaptation to hybrid CPU–GPU architectures, followed by a MPDATA parallelization method on CPUs, described in Section 6. Our MPDATA parallelization approach on GPUs is presented in Section 7, which also describes management of GPU resources using the developed scheduler. Section 8 introduces the automatic tunning of MPDATA on GPUs, including implementation and experimental verification. Performance results are presented in Section 9, while Section 10 gives conclusions and future work.

2. Related works

Stencil computations on regular grids are kernels for a wide range of scientific codes. In these computations each point in a multidimensional grid is updated with contributions from a subset of its neighbors [15], where updating all points in the grid according to a fixed rule is called a sweep. Reorganizing stencil calculations to take full advantage of memory hierarchies has been the subject of much investigation over the years [28,15,7,8,38,6,22,24,36,40,9,5]. This is motivated both by the importance of these kernels, and their poor performance in comparison to machine peak as data can not be transferred from the main memory fast enough to avoid stalling the computational units. Memory optimizations for stencil computations have principally focused on different decomposition strategies, like space and temporal blocking techniques [6,9], 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 multi-/manycore architectures, including CPUs and GPUs (see, e.g., [7,38,24,37]). Another popular optimization strategy [5,11] is based on using halos, also known as ghost regions or overlapped tilling, for reducing communication. In particular, the ghost cell extension method [11] has been applied in [39] to improve the efficiency of a multi-GPU shallow-water simulation. At the same time, the issue of adapting stencil computations to hybrid CPU–GPU platforms, with computations spread across both CPU and GPU components, has been tackled in [38] only.

The underlying assumption for using the temporal blocking and ghost cell expansion method is that no other computation need to be performed between consecutive stencil sweeps. This assumption has been aggressively used in [38] to improve the efficiency of implementing 2D stencil codes on hybrid CPU–GPU platforms by removing or delaying synchronization between iterations. This is not the case for EULAG computations where the MPDATA algorithm is interleaved with the elliptic solver, in each iteration (or time step). In our paper we exploit the fact that each iteration of MPDATA consists of a series of stencil sweeps. This allow us to merge all the sweeps within a series, to develop decomposition strategies which aim at alleviating memory and communication bounds, and better exploit the theoretical floating point efficiency of hybrid CPU–GPU architectures. In particular, it becomes possible to reduce traffic over the PCIe bus connecting GPU to CPU, which is the primary bottleneck for an efficient utilization of hybrid platforms.

The complexity of modern multicore and accelerator architectures makes it extremely difficult to adapt demanding numerical algorithms to these architectures [43]. An efficient way to solve this problem is software automatic tuning (autotuning in short), which is a paradigm enabling software adaptation to a variety of computational conditions [30,16,4]. Originating from the stream of research works on HPC, it is considered to be one of the most promising approaches to the required performance advancements on the next generation supercomputing platforms.

Autotuning has been used intensively on CPUs [12,21] to automatically generate near optimal numerical libraries. For example, ATLAS and PHiPAC used it to generate a highly optimized BLAS. Work [21] on autotuning CUDA kernels for NVIDIA GPUs has shown that this technique is a very practical approach to port existing algorithmic solutions on quickly evolving GPU architectures, and to substantially speed up even highly tuned hand-written kernels.

The main approach to autotuning is based [20] on empirical optimization techniques. Namely, these are techniques to generate a large number of parameterized code variants for a given algorithm, and run these variants on a given platform to discover the best one using the feedback loop. This approach could select near-optimum parameters for a given machine. However, the autotuning method is just a general concept of optimization of computations. Each algorithm requires a specific analysis and deployment of this concept for a particular class of computer architectures.

The empirical methods are usually based on a search space, which need to be pruned by an autotuning mechanism. Some of such mechanisms are clearly described by K. Sato et al. in [30], where an approach similar to ours is proposed for CUDA stencil codes. However, their approach differs from ours, since we take into account not only a configuration of CTA (Cooperative Thread Array, which is called the work-group in our work) for different grid sizes, but also parameters responsible for the GPU global memory management, and overlapping data transfers with computations. Moreover, we take into consideration portability of code across AMD and NVIDIA GPUs. Several methods of defining and evaluating the search space for

stencil codes are described in [5]. Some of them are very general, and can be simply adopted to our approach (exhaustive search, pruning a search space). The difference between general approaches to the autotuning and our mechanism derives from the fact that our mechanism takes into account specificity of the MPDATA algorithm, and can be applied mainly to a group of forward-in-time algorithms, where a single execution of the algorithm take a relatively short time, and empirical tests are not so expensive.

3. Hybrid CPU–GPU architectures and programming environment

3.1. Target hybrid platforms

A summary of key features of hybrid platforms used in this study is shown in Table 1. These platforms contain different numbers of CPUs and GPUs – from two solutions consisting of a single CPU and single GPU, through a platform with two CPUs and one GPU, to the most elaborate configuration containing two CPUs and two GPUs. Processors of different vendors are employed by manufacturers of these systems – both Intel and AMD CPUs, as well as GPUs from NVIDIA and AMD. It should be noted that values of peak performance shown in Table 1 are given for double precision arithmetic, without taking into account MAD or FMA operations since the MPDATA algorithm does not enable for utilizing them. Moreover, in the case of CPUs these values assume the usage of SIMD vectorization based on either the AVX extension (for Intel Sandy Bridge-EP), or SSE extension (for others).

3.2. Hybrid programming environment and OpenCL standard for GPU programming

To harness the power of CPU–GPU platforms, a hybrid programming environment is assumed in this research. To manage CPU and GPU components, we take advantage of using the OpenMP and OpenCL standards, respectively. The first one supports multi-platform shared memory parallel programming on most CPU architectures, while the main reason for selecting OpenCL is the requirement to design a performance portable adaptation of the MPDATA algorithm across both NVIDIA and AMD GPUs [1,10,25]. In principle, OpenCL can be used even for the CPU programming. However, after performance tests with simple MPDATA stencils, using OpenMP v.3.0 and OpenCL v.1.2, we decided to select OpenMP for CPUs, as it was more performance efficient than OpenCL.

OpenCL is an open standard for parallel programming of heterogeneous computing systems, maintained by the non-profit Khronos Group [18]. The OpenCL software architecture [10,23] 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. Each data item used in the GPU needs to be copied from the CPU host memory to the GPU memory; each of these transfers is a source of latency which affects the resulting performance negatively [23]. These performance overheads can be reduced using asynchronous command queues. It allows for overlapping GPU computations with data transfers between the main memory and the GPU global memory.

The OpenCL specification [23] enables the efficient management of GPU computing resources, beginning with *processing elements*, which correspond to GPU cores. Processing elements are grouped into *compute units* corresponding to GPU

Table 1
Specification of test platforms.

Setup	2CPUs + 2GPUs	2CPUs + GPU	CPU + GPU	CPU + GPU
CPU	Interlagos AMD Opteron 6234	Westmere Intel Xeon E5649	Sandy Bridge-EP Intel Xeon E5-2670	Thuban AMD Phenom X6 1090T
Frequency [GHz]	2.4	2.53	2.6	3.2
Number of cores	2×12	2×6	8	6
Memory size [GB]	48	48	48	8
Memory band. [GB/s]	$2 \times 2 \times 21.3$	2×25.6	42.6	21.3
L3 cache size [MB]	$2 \times 2 \times 8$	2×12	20	6
Peak perf. [GFlop/s]	2×57.6	2×30.36	83.2	38.4
GPU	Fermi NVIDIA Tesla M2050	Fermi NVIDIA Tesla M2070Q	Kepler NVIDIA Tesla K20	Evergreen ATI Radeon HD 5870
Frequency [GHz]	1150	1150	706	850
Number of cores	448	448	2496	1600
Memory size [GB]	3	6	5	2
Memory band. [GB/s]	148.4	150.3	208	153.6
Shared mem. size [KB]	48	48	48	32
Peak perf. [GFlop/s]	2×257.6	257.6	585	540
PCIe band. [GB/s]	8	8	16	8
Total peak performance [GFlop/s]	630.4	318.32	668.2	578.4

streaming multiprocessors. Finally, a collection of compute units creates a *compute device*, which accounts for the whole GPU processor. In the OpenCL data parallel model, the same program (or kernel) runs concurrently on different pieces of data, and each invocation is called a *work-item*. The work-items (or threads) are organized in up to three dimensions. The set of work-items is called a *work-group*. Each work-group is executed on a single compute unit. Work-items can be synchronized within a single work-group. However, there is no synchronization mechanism between work-groups; they are executed independently.

Another key feature of modern GPUs is their hierarchical memory organization. In the OpenCL memory model, all the GPU threads have access to the *global memory*, relatively large but rather slow. Within a particular work-group, all the work-items share the fast *local memory*. It is used for communication and synchronization among work-items across the work-group. Finally, each work-item has access to its *private memory*. The OpenCL memory model is mapped onto the physical memory organization of a specific GPU architecture. In particular, for the NVIDIA Fermi architecture [25], the private memory is mostly implemented as a pool of registers, while the local memory corresponds to the *shared memory* assigned to each streaming multiprocessor. The size of shared memory space is limited to 48 KB, while the global memory provides up to 6 GB of GDDR5 memory (for a single-GPU architecture).

4. Basics of 2D MPDATA computations

4.1. Introduction to MPDATA algorithm with nonoscillatory option

The MPDATA algorithm belongs to the group of nonoscillatory forward-in-time algorithms [31]. The two-dimensional MPDATA [34,32] is based on the two-dimensional advection equation:

$$\frac{\partial \Psi}{\partial t} = -\frac{\partial}{\partial x}(u\Psi) - \frac{\partial}{\partial y}(v\Psi), \tag{1}$$

where x and y are space coordinates, t is time, $u, v = const$ are flow velocities, and Ψ is a nonnegative scalar field. Eqn. (1) is approximated according to the donor-cell scheme, which for the $(n+1)$th time step $(n = 0, 1, 2, \ldots)$ gives the following equation:

$$\Psi^*_{i,j} = \Psi^n_{i,j} - \left[F\left(\Psi^n_{i,j}, \Psi^n_{i+1,j}, U_{i+1/2,j}\right) - F\left(\Psi^n_{i-1,j}, \Psi^n_{i,j}, U_{i-1/2,j}\right)\right] - \left[F\left(\Psi^n_{i,j}, \Psi^n_{i,j+1}, V_{i,j+1/2}\right) - F\left(\Psi^n_{i,j-1}, \Psi^n_{i,j}, V_{i,j-1/2}\right)\right]. \tag{2}$$

Here the function F is defined in terms of the local Courant number U:

$$F(\Psi_L, \Psi_R, U) \equiv [U]^+ \Psi_L + [U]^- \Psi_R, \tag{3}$$

$$U \equiv \frac{u\delta t}{\delta x}; \quad [U]^+ \equiv 0.5(U + |U|); \quad [U]^- \equiv 0.5(U - |U|). \tag{4}$$

The same definition is true for the local Courant number V.

The first-order accurate advection equation can be approximated to the second-order in $\delta x, \delta y$ and δt, defining the advection–diffusion equation:

$$\frac{\partial \Psi}{\partial t} = -\frac{\partial}{\partial x}(u\Psi) + \frac{(\delta x)^2}{2\delta t}(|U| - U^2)\frac{\partial^2 \Psi}{\partial x^2} - \frac{\partial}{\partial y}(v\Psi) + \frac{(\delta y)^2}{2\delta t}(|V| - V^2)\frac{\partial^2 \Psi}{\partial y^2} - \frac{UV\delta x\delta y}{\delta t}\frac{\partial^2 \Psi}{\partial x \partial y}. \tag{5}$$

The antidiffusive pseudo velocities $\tilde{u}$ and $\tilde{v}$ in respectively x and y directions are defined according to the following equations:

$$\tilde{u} = \frac{(\delta x)^2}{2\delta t}(|U| - U^2)\frac{1}{\Psi}\frac{\partial \Psi}{\partial x} - \frac{UV\delta x\delta y}{2\delta t}\frac{1}{\Psi}\frac{\partial \Psi}{\partial y}, \tag{6}$$

$$\tilde{v} = \frac{(\delta y)^2}{2\delta t}(|V| - V^2)\frac{1}{\Psi}\frac{\partial \Psi}{\partial y} - \frac{UV\delta x\delta y}{2\delta t}\frac{1}{\Psi}\frac{\partial \Psi}{\partial x}. \tag{7}$$

Therefore, in order to compensate the first-order error of Eqn. (1), once again the donor-cell scheme is used but with the antidiffusive velocity $\tilde{u} = -u_d$ in place of u, and with the value of Ψ^* already updated in Eqn. (2) in place of Ψ^n. It allows us to compute values of Ψ for the $(n+1)$th time step.

The described algorithm does not preserve the monotonicity of the transported variables and, in general, the solutions are not free of spurious extrema [31]. However, when required MPDATA can be made fully monotone by adapting the flux-corrected transport (FCT) formalism to limit the pseudo velocities. This method is called a nonoscillatory option of MPDATA. The FCT-limited antidiffusive velocity for the jth dimension of grid ($j = 1, 2, \ldots, M$ for an M-dimensional grid) can be written as:

$$\widetilde{U}^j_{i+1/2e^j} = \min\left(1, \beta^{MIN}_i, \beta^{MAX}_{i+e^j}\right)\left[\widetilde{U}^j_{i+1/2e^j}\right]^+ + \min\left(1, \beta^{MAX}_i, \beta^{MIN}_{i+e^j}\right)\left[\widetilde{U}^j_{i+1/2e^j}\right]^-, \tag{8}$$

where U^j is the local Courant number for the jth dimension, and e^j corresponds to the neighbor element in the jth dimension. The limiting coefficients β^{MAX} and β^{MIN} are as follows:

$$\beta_i^{MAX} = \frac{\Psi_i^{MAX} - \Psi_i^*}{\sum_{j=1}^{M} \frac{\delta t}{\delta X^j} \left(\left[\widetilde{U}^j_{i-1/2e^j} \right]^+ \Psi^*_{i-e^j} - \left[\widetilde{U}^j_{i+1/2e^j} \right]^- \Psi^*_{i+e^j} \right) + \varepsilon}, \tag{9}$$

$$\beta_i^{MIN} = \frac{\Psi_i^* - \Psi_i^{MIN}}{\sum_{j=1}^{M} \frac{\delta t}{\delta X^j} \left(\left[\widetilde{U}^j_{i+1/2e^j} \right]^+ \Psi^*_{i} - \left[\widetilde{U}^j_{i-1/2e^j} \right]^- \Psi^*_{i} \right) + \varepsilon}, \tag{10}$$

where δX^j is the spacing between grid points in the jth dimension, Ψ^* is computed according to Eqn. (2), and Ψ^{MAX} as well as Ψ^{MIN} are computed based on the following equations:

$$\Psi_i^{MAX} = \max_{j=1,M} \left(\Psi^n_{i-e^j}, \Psi^n_i, \Psi^n_{i+e^j}, \Psi^*_{i-e^j}, \Psi^*_i, \Psi^*_{i+e^j} \right), \tag{11}$$

$$\Psi_i^{MIN} = \min_{j=1,M} \left(\Psi^n_{i-e^j}, \Psi^n_i, \Psi^n_{i+e^j}, \Psi^*_{i-e^j}, \Psi^*_i, \Psi^*_{i+e^j} \right). \tag{12}$$

4.2. Data dependencies in MPDATA

Depending on a type of simulated physical phenomenon, the number of time steps for MPDATA can exceed a few millions [26,33]. Considering the MPDATA algorithm as a part of the EULAG model, it should be noted that there are other algorithms (e.g., elliptic solver) executed in consecutive time steps in combination with MPDATA. Therefore, our method of MPDATA parallelization is constrained to a single time step.

The MPDATA computations in each step are decomposed into a set of sixteen stencil sweeps, called here stages, with data dependencies between stages described by a data dependency graph. Fig. 1 shows a part of MPDATA implementation, illustrating dependencies between stages. Each stage is responsible for calculating elements of a certain matrix, based on the corresponding stencil. The 3rd stage depends on the 1st and 2nd stages, where computations in each grid point for the 3rd stage require two neighboring elements in i-dimension, calculated at the 1st stage, and two neighboring elements in j-dimension,

```
#define fdim(a, b) ( (a>b) ? (a-b):(0.0) )
#define donor(y1, y2, a) ( fdim(a, 0.0) * (y1) - fdim(0.0, a) * (y2) )
//stage 1
for( ... ) // i - dimension
  for( ... ) // j - dimension
    f1[i][j] = donor( xIn[i-1][j], xIn[i][j], u1[i][j] );
/*...*/
//stage 3
for( ... ) // i - dimension
  for( ... ) // j - dimension
    x[i][j] = xIn[i][j] - ( f1[i+1][j]-f1[i][j]+f2[i][j+1]-f2[i][j] )/h[i][j];
//stage 4
for( ... ) // i - dimension
  for( ... ) { // j - dimension
      mx[i,j]=max( xIn[i][j-1], xIn[i][j], xIn[i][j+1], xIn[i+1][j], xIn[i-1][j] );
      mn[i,j]=min( xIn[i][j-1], xIn[i][j], xIn[i][j+1], xIn[i+1][j], xIn[i-1][j] );
//stage 5
for( ... ) // i - dimension
  for( ... ) // j - dimension
    v1[i][j] = vdyf(x[i-1][j], x[i][j],u1[i][j], 0.5*(h[i-1][j]+h[i][j])) +
               vcorr(u1[i][j], u2[i-1][j]+u2[i-1][j+1]+u2[i][j+1]+u2[i][j],
                     x[i-1][j-1], x[i][j-1], x[i-1][j+1], x[i][j+1],
                     0.5*(h[i-1][j]+h[i][j]));
/*...*/
//stage 7
for( ... ) // i - dimension
  for( ... ) { // j - dimension
      mx[i,j]=max( mx[i,j], x[i][j-1], x[i][j], x[i][j+1], x[i+1][j], x[i-1][j] );
      mn[i,j]=min( mn[i,j], x[i][j-1], x[i][j], x[i][j+1], x[i+1][j], x[i-1][j] );
/*...*/
```

Fig. 1. Part of MPDATA implementation.

derived at the 2nd stage. The stages from 1 to 3 are sufficient to compute the 5th and 6th stages, while the 7th stage requires all the stages from 1 to 4. Here the matrix x corresponds to the field Ψ, the matrices $u1$ and $u2$ are flow velocities, while the matrices $v1$ and $v2$ are pseudo velocities; the matrices $f1$, $f2$, mx, and mn contain some temporary values. Obviously, the input matrix xIn for stages from 1 to 4 is calculated in the previous time step.

The data dependencies occurring between all the stages create a data dependency graph shown in Fig. 2a. The first three stages correspond to the donor-cell scheme given by Eqn. (2). The 4th, 7th, 10th, and 11th stages are necessary to apply the nonoscillatory option corresponding to Eqn. (8), while the rest of stages provide the second-order accuracy of the MPDATA algorithm. Almost every stage depends on more than one stage. For each stage, its output data are input ones for next stages, and finally the matrix x is calculated at the last stage as the output of each MPDATA time step. This matrix is then input data for the next time step. As a result, the order of executing MPDATA stages is constrained by the data dependency graph shown in Fig. 2a. Our approach to parallelization of MPDATA on hybrid architectures does not affect the structure of the MPDATA algorithm given by this graph. This requirement provides compatibility with the standard implementation of the EULAG model [34]. Another requirement is to perform all computations in double precision.

Stencils shown in Fig. 2b - 2e illustrate in detail how these dependencies are created. For example, an element (i,j) of $v1$-matrix to be calculated at stage 5 requires six elements $(i-1,\ j-1)$, $(i-1,j)$, $(i-1,j+1)$, $(i,j-1)$, (i,j), $(i,j+1)$ of x-matrix computed at the 3rd stage.

5. Adaptation of MPDATA to CPU–GPU architecture

Stencil algorithms are commonly known as memory bounded [13]. When mapping the MPDATA algorithm on hybrid CPU–GPU architectures with computation spread across the entire machine, the most important performance limitation becomes the relatively low bandwidth B_{PCIe} of the PCIe bus connecting GPU to CPU. In fact, for our test platforms we have at most $B_{PCIe} = 2$ GWord/s (double precision data), while for CPU and GPU the memory bandwidth is respectively $B_{CPU} = 5.33$ GWord/s (Intel Xeon E5-2670) and $B_{GPU} = 13$ GWord/s (NVIDIA Tesla K20). Therefore, in the case of hybrid CPU–GPU architectures any MPDATA code optimization should first aim at reducing traffic between both components (CPU and GPU). Only the second priority is to reduce data transfers, and make them as efficient as possible for each component.

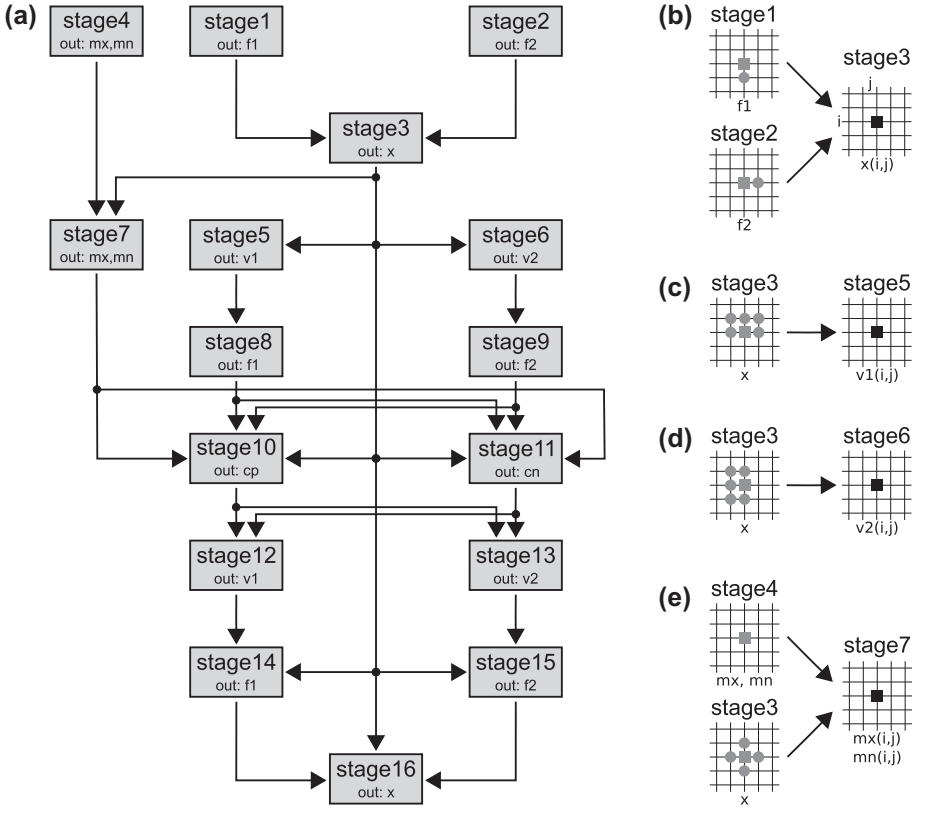

Fig. 2. Data dependency graph for MPDATA (a), and examples of stencils for different stages (b–e).

In this section, we consider adaptation of the 2D MPDATA algorithm to the hybrid CPU–GPU architecture, following the first priority. The main goal is to minimize the idleness of computing resources for both CPU and GPU processors, taking into account properties of the MPDATA algorithm, as well as performance constraints of the underlying architecture.

A special challenge associated with such an adaptation is the decomposition of MPDATA grid between CPU and GPU. The basic strategy of grid partitioning assigns separate stripes of grid rows to CPU and GPU. As a result, CPU–GPU data transfers are necessary only for computing elements placed around the border between CPU and GPU. These elements belong to the so-called halo areas, corresponding to grid rows. Moreover, CPU–GPU communications take place for all the stages. This prohibits the efficient utilization of available resources.

CPU–GPU transfers of halo areas can be avoided at the cost of extra computations (Fig. 3b), based on the ghost cell expansion method [11]. These extra computation result from extending halo areas by extra rows, for both the CPU and GPU domains. The CPU has now to compute more rows than before, since some rows, which originally were assigned to the GPU domain only, are now duplicated in the CPU domain, and vice versa.

Fig. 3 explains by the example of stages 1, 3, and 6 how to recursively extend halo areas, based on dependencies shown in Fig. 2. Firstly based on dependencies between stages 3 and 6, to remove CPU–GPU communications when executing these stages, we extend the CPU domain by a row, which corresponds to a CPU halo area with the size of $ihB = 1$. Similarly, the GPU domain has to be extended by a single row, that gives a GPU halo area with the size of $ihT = 1$. Continuing this analysis for dependencies between stages 1 and 3, we conclude that the CPU halo area has to be further extended by an additional row, that gives the CPU halo area of size $ihB = 2$, for stage 1. At the same time, due to the properties of dependencies between stages 1 and 3, there is no need to extend the GPU halo area determined previously, so finally $ihT = 1$, for stage 1.

A similar analysis can be applied for dependencies between all the stages of MPDATA. It allows us to determine sizes of halo areas corresponding to extra computations performed by CPU and GPU at every stage. The resulting values of *ihB* and *ihT* are presented in Table 2.

The proposed approach allows for avoiding communication between CPU and GPU domains within each time step of the MPDATA algorithm. Thus, CPU and GPU components can compute their domains independently, but each of the two components needs to perform extra computations, in accordance with dependencies between MPDATA stages. As is shown in Fig. 4, the CPU–GPU cooperation, including communication and synchronization, is now required only after each time step.

To implement the scheme presented in Fig. 4, we use the OpenMP–OpenCL hybrid programming environment shown in Fig. 5. It allows us to perform MPDATA computations on both CPU and GPU resources simultaneously. One of OpenMP threads is responsible for the GPU management, using the OpenCL Host API, while other OpenMP threads, as well as OpenCL work-items, provide parallel computations. The synchronization and communication between CPU and GPU, required after each time step, are performed via access to the main memory. This access is implemented by both the OpenMP and OpenCL Host API mechanisms, which are synchronized by a common OpenMP *barrier* construct.

When adapting MPDATA to the hybrid CPU–GPU architecture, the next challenge is to provide high performance for each component, taking into account their heterogeneity. Hence, two different adaptations of the MPDATA algorithm to CPU and GPU processors are required. Each of these adaptations should primarily take into account constraints for the memory bandwidth. The key idea to alleviate these limitations is to reduce saturation of memory traffic within CPU, as well as GPU. For CPU, this goal can be achieved by taking advantage of cache memory reuse, as high as possible. For the graphics processor, the presence of fast and relatively large GPU global memory allows us to decrease the intensity of access to the main memory, since results of GPU computations performed within a single time step can be stored in GPU, without sending them to the main memory. As a result, performance restrictions due to the memory bandwidth saturation can be minimized, and the high density of computing resources should be better utilized.

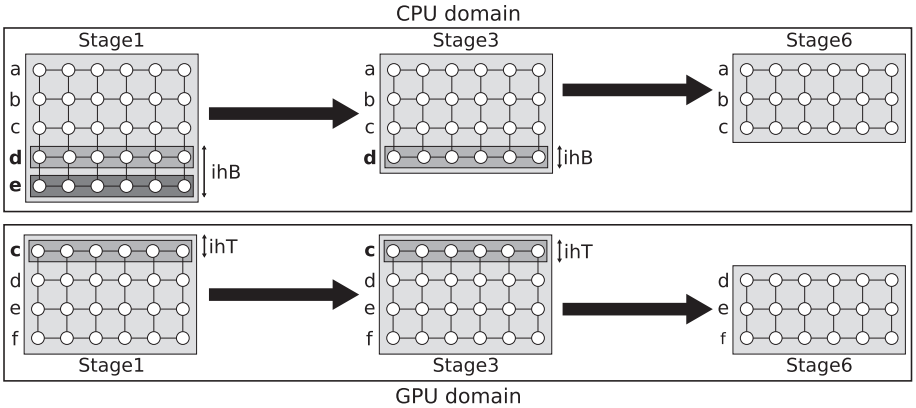

Fig. 3. Idea of decomposition of MPDATA grid across CPU and GPU to avoid communication between components at the cost of additional computations.

Table 2
Sizes of halo areas in vertical direction, for different MPDATA stages.

Stage	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16
ihT	2	2	2	1	1	1	1	1	1	1	1	0	0	0	0	0
ihB	3	2	2	1	2	1	1	2	1	1	1	1	0	1	0	0

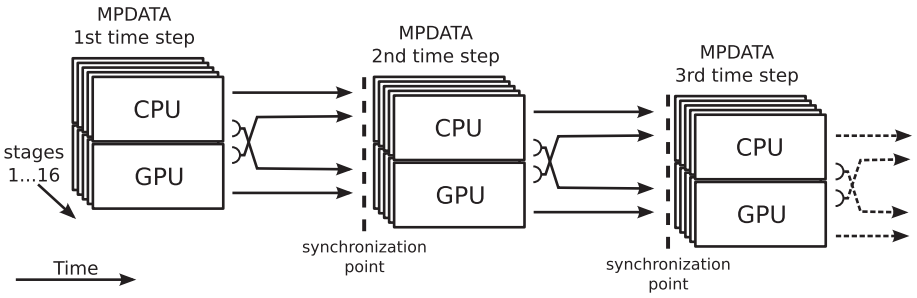

Fig. 4. Scheme of cooperation between CPU and GPU components running MPDATA algorithm.

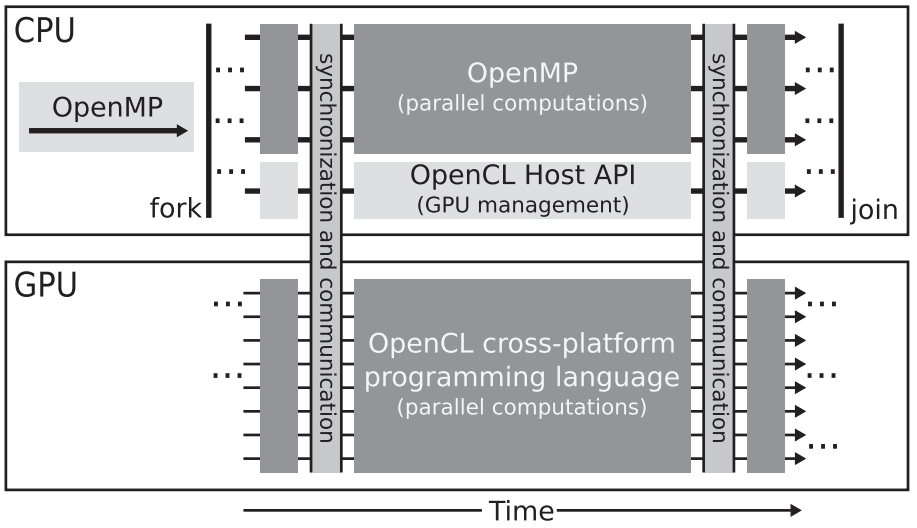

Fig. 5. Hybrid programming environment.

6. Parallelization of MPDATA on CPU

The aim of this section is to develop a parallel mapping of MPDATA computation on multicore CPU architectures. The starting point is a block decomposition of MPDATA, whose efficiency is then improved by eliminating a part of extra computation corresponding to halo areas. In the resulting mapping, one block is processed at a time by different CPU cores. A refinement of this mapping is finally proposed to take into account properties of a specific CPU platform.

6.1. CPU block decomposition of MPDATA

Shows that the main memory traffic is a bottleneck when implementing MPDATA on multicore CPUs. By considering the execution of all the MPDATA stages within a single time step, we obtain the possibility to reduce memory traffic after applying a block decomposition strategy shown in Fig. 6. In consequence, the intermediate results of computations performed at all but the last stage can be placed in the cache memory, while only the final results should be returned to the main memory. Such an approach to cache reuse is commonly called temporal blocking [24,36].

The proposed block decomposition combines both space and temporal blocking techniques. The basic requirement is to keep in cache all the data required for MPDATA computations for each block, within the whole time step. Taking into account data dependencies between all 16 stages of MPDATA, each block requires some extra calculations to be performed for every

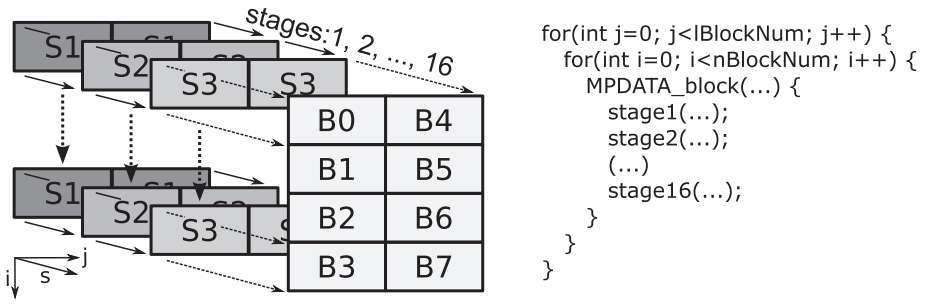

Fig. 6. CPU block decomposition of MPDATA.

stage, in order to ensure the correct results of the whole algorithm. Extra computations take place on the borders between neighboring blocks. As a consequence, blocks have to be extended by adequate halo areas, both in the vertical and horizontal directions. For each stage, sizes of halo areas are given by parameters ihT, ihB, ihL, and ihR (top, bottom, left, and right).

The sizes of halo areas in the vertical direction are already determined in Table 2. Based on the same analysis as in Section 5, it is possible to determine halo areas in the horizontal direction as well (see Table 3). By the example of dependencies between stages 1, 2, 3, and 6 (Fig. 7a), we explain how to recursively expand halo areas in both directions (Fig. 7b and 7c).

Initially, in the proposed approach all the blocks are executed independently, within each time step. It reduces the saturation of main memory traffic at the cost of extra computations. In addition, such a CPU block decomposition of MPDATA is fully compatible with the hybrid CPU–GPU decomposition proposed in the previous section.

The amount of all the data required for MPDATA computations within each block cannot exceed the cache size. Furthermore, the smaller block size the more extra calculations are necessary. So the efficiency of our approach increases with increasing the size of blocks, which is limited by the size of available cache.

Assuming blocks of size $nB \times lB$, the total consumption C_T of cache memory can be estimated by the following equation:

$$C_T = \sum_{i=1}^{22} [(ihB_i + nB + ihT_i) * (ihL_i + lB + ihR_i)] \ [Word], \tag{13}$$

where the first 16 components in the above sum correspond to MPDATA stages. The last 6 components take into account the necessity to store blocks of input matrices, which are also extended by appropriate halo areas. Table 4 shows estimations of C_T calculated for different values of nB and lB.

6.2. Improving efficiency of block decomposition

In general, each block requires extra computations for all the four halo areas (top, bottom, left, and right); these computations are duplicated by adjacent blocks. In order to increase the efficiency of CPU block decomposition, we introduce a method that allows us to minimize this overhead, without increasing the cache consumption. In this method, extra computations in the top and bottom halo areas can be avoided by leaving partial results in the cache memory. So after performing computations within a certain block, the required partial results for all the MPDATA stages are left in cache for executing the next block. When implementing this method, the key requirement is to order computations by columns of blocks. This method requires also to provide an appropriate mapping of partial results onto the cache space based on the round robin technique described in [36]. Hence, the arrays that store partial results for all the MPDATA stages are appropriately shifted when executing successive blocks.

The advantage of the method is making the amount of extra computations independent of the block size nB. The total amount E_{new} of extra elements of MPDATA matrices computed in this method depends only on lB size, and sizes n, l of grid:

$$E_{new} = \frac{n * (l - lB)}{lB} \sum_{i=1}^{16} [(ihL_i + ihR_i)]. \tag{14}$$

For the grid of size 2048×2048, Table 5 presents values of percentage $E^{\%}_{new}$ of the amount E_{new} to the total number $E = 16 * n * l$ of all the elements of matrices computed by the MPDATA algorithm. In the best case, when $lB = l$, there are no extra computations at all. Moreover, we have possibility to choose such a value of size nB that the required cache

Table 3
Size of halo areas in horizontal direction, for all MPDATA stages.

No. stages	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16
ihL	2	2	2	1	1	1	1	1	1	1	1	0	0	0	0	0
ihR	2	3	2	1	1	2	1	1	2	1	1	0	1	0	1	0

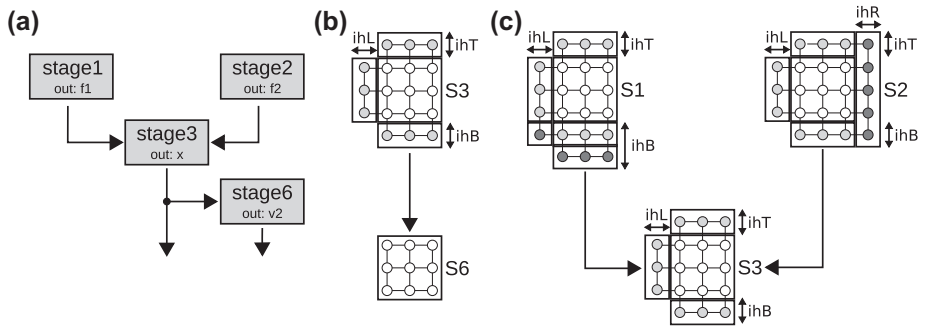

Fig. 7. Building halo areas for CPU blocking.

Table 4
Cache consumption [KB] for CPU block decomposition, assuming different values of *nB* and *lB*.

nB\lB	2	4	8	16	32	64	128	256	512	1024	2048
2	4	6	9	15	28	54	105	207	412	822	1641
4	6	8	13	22	40	77	151	298	593	1183	2363
8	9	13	20	35	65	124	243	480	955	1906	3806
16	15	22	35	61	113	218	427	844	1680	3351	6693
32	28	40	65	113	210	405	794	1572	3129	6242	12,468
64	54	77	124	218	405	780	1529	3029	6027	12,023	24,016
128	105	151	243	427	794	1529	3000	5941	11,823	23,586	47,114
256	207	298	480	844	1572	3029	5941	11,765	23,414	46,712	93,308
512	412	593	955	1680	3129	6027	11,823	23,414	46,598	92,964	185,698
1024	822	1183	1906	3351	6242	12,023	23,586	46,712	92,964	185,468	370,477
2048	1641	2363	3806	6693	12,468	24,016	47,114	93,308	185,698	370,477	740,034

Table 5
Values of percentage $E^{\%}_{new}$ for different values of size *lB*.

nB \lB	2	4	8	16	32	64	128	256	512	1024	2048
2–2048	102.7	51.3	25.6	12.7	6.3	3.1	1.5	0.7	0.3	0.1	0.0

consumption (Table 4) does not exceed the available size of cache. For the tested CPU platforms, it is possible even for AMD Phenom X6 equipped with 6 MB of L3 cache memory. In this case, based on Tables 4 and 5, we can choose the final block sizes $nB \times lB$ among the following three options: 2×2048, 4×2048, and 8×2048.

6.3. Mapping MPDATA on resources of CPU platforms

The basic scheme of mapping MPDATA on CPU-based computing resources is shown in Fig. 8a. For each block, computations are distributed across n_T threads, which are mapped onto the available CPU cores. Thus, within a certain stage s, where $s = 1, 2, \ldots, 16$, each block of size $nB^*_s \times lB^*_s$ (where $nB^*_s = ihB_s + nB + ihT_s$, and $lB^*_s = ihL_s + lB + ihR_s$) is divided into chunks of size $nB^*_s \times (lB^*_s/n_T)$, and these chunks are assigned to threads separately. Another level of parallelization is vectorization applied inside each thread, so the resulting SIMDification is performed within rows of each chunk. In consequence, values of lB^*_s have to be adjusted to a vector size.

At the same time, for a fixed MPDATA block, a sequence of stages is executed, satisfying the dependency graph. Due to the data dependencies of MPDATA, appropriate synchronizations between MPDATA stages are necessary. Finally, different MPDATA blocks are processed sequentially, following the order proposed for the CPU block decomposition in the previous subsection.

Although the block decomposition of MPDATA ensures an acceptable usage of memory bandwidth, it still does not guarantee a satisfying utilization of a specific CPU platform. In fact, such platforms contain groups of cores, which correspond to different processors (Fig. 8b) or even different dies within a single processor (Fig. 8c). The latter takes place, e.g., in the case of AMD Interlagos CPUs. These groups are interconnected either by Intel QPI or AMD HyperTransport (HT) links. Thus, each group of cores has direct access to its own cache memory, and indirect access to caches of other groups.

To alleviate overheads due to inter-cache communications between groups of cores, it is possible to adapt the basic mapping scheme to properties of a specific CPU platform. For this aim, we use exactly the same grid decomposition as in the case of adaptation of MPDATA to CPU–GPU architecture (see Section 5). This decomposition permits for reducing

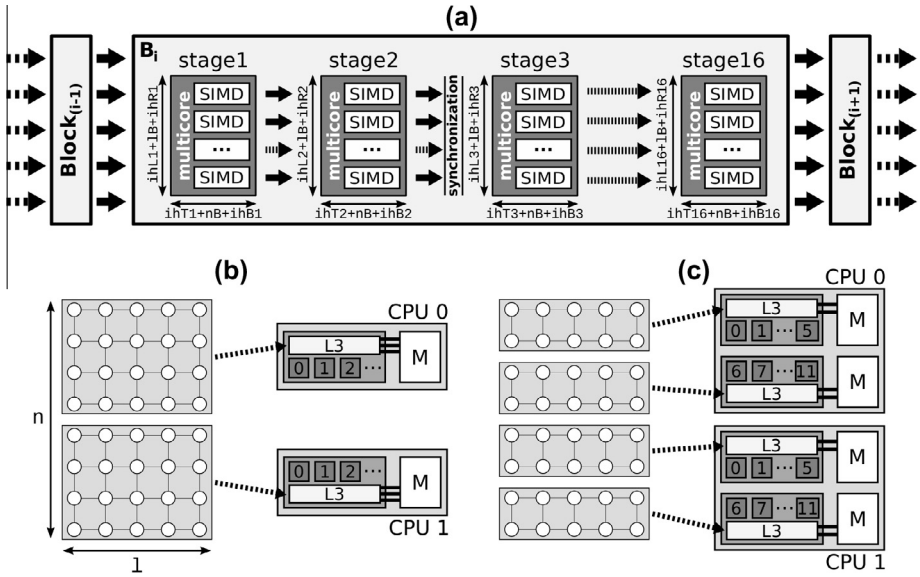

Fig. 8. Mapping MPDATA on resources of CPU platforms: (a) basic scheme, (b) Intel Xeon, and (c) AMD Opteron Interlagos platforms.

inter-cache communications across either two Intel Xeon processors (Fig. 8b), or four dies within two AMD Interlagos CPUs (Fig. 8c). Another advantage of this approach is possibility to apply the NUMA "first-touch" policy more efficiently.

The standard OpenMP barriers are used for the synchronization of computation between time steps (Fig. 5). Additionally, within each time step, the synchronization of threads is required between selected stages, as shown in Fig. 8a. Since groups of cores execute computation independently from each other, these inter-stage synchronizations should be provided within each group. Therefore, the group-level barriers using the OpenMP atomic directive are implemented to carry out the mutual independence of groups.

Remark 1. When developing the basic scheme of mapping MPDATA on CPU resources (Fig. 8a), we took into account different strategies of partitioning computation onto cores - by chunks of columns or rows, as well as a mixture of them. In some cases, the first strategy, which is used in this paper, gave a better performance, while for others the winner was the second or third strategy, depending on the configuration, size of block, number of cores, etc. However, the performance difference for these strategies was below 3% for all the tests.

Remark 2. The grid decomposition illustrated in Fig. 8b and Fig. 8c allows us to reduce inter-chip communications and save some synchronizations, but requires some extra computation. Therefore, the proposed strategy is perfectly suited to reduce overheads associated with inter-chip communications between groups of cores, since the cost of extra computation is largely amortized by decreasing the communication cost introduced by relatively slow Intel QPI or AMD HT links. However, this strategy does not give the desired effect at the core level with intra-chip communications taking place between cores connected to each other in the last level cache. In this case, the cost of extra computation is not sufficiently amortized by reducing the communication cost, due to much faster communications within the last level cache than communications across QPI or HT links.

6.4. CPU performance results

Table 6 presents the execution time of the 2D MPDATA algorithm achieved for 500 time steps and grid of size 2048×2048, on Intel Xeon E5649 (Westmere) and AMD Opteron 6234 (Interlagos) CPUs. These performance results are obtained for two versions of MPDATA: serial and block ones, using the gcc compiler (version 4.4 or 4.6 depending on the platform) with optimization flag -O3. The serial version is a non-blocking implementation of MPDATA, without the SIMD vectorization. The block version is based on the proposed CPU block decomposition of MPDATA. For the grid of size 2048×2048, the best performance is achieved using the block of size 8×2048.

The presented results confirm a significant improvement in the MPDATA performance due to the proposed CPU block decomposition. The performance gain is reported even for a single core implementation of MPDATA (see the first and second rows of Table 6). The highest benefit of this approach is noticed when all the available computing resources are utilized. In this case, the speedup over the serial version is 15.21 and 39.82 for two Intel Westmere CPUs (12 cores in total) and two AMD

Table 6
Execution time (in seconds) of 2D MPDATA on Intel and AMD CPUs, for 500 time steps and grid of size 2048 × 2048.

Version	CPU resources	SIMD	# Groups	Intel Westmere Xeon E5649	AMD Interlagos Opteron 6234
Serial	1core	–	–	224.71	384.68
Block	1core	–	–	186.10	259.28
Block	1core	+	–	118.60	-
Block	1CPU	+	1	28.66	28.64
Block	1CPU	+	2	–	18.96
Block	2CPUs	+	1	19.52	30.45
Block	2CPUs	+	2	14.77	14.25
Block	2CPUs	+	4	–	9.66
Block	2CPUs	–	2/4*	20.94	17.07

* 2 Groups for two Intel CPUs, and 4 groups for two AMD CPUs.

Interlagos CPUs (24 cores), respectively. These results confirm that the performance gain of the block version is higher for CPU platforms characterized by a higher computing performance and lower memory bandwidth.

The mapping proposed in Section 6.2 is the main strategy to increase the benefit of using multiprocessor platforms, since it allows for reducing inter-cache communications. This effect is particularly noticeable for the platform with two AMD Opteron 6234 CPUs, where four groups of threads (6 threads per group) are used to provide the highest performance. This solution allows us to accelerate MPDATA computations 3.15 times, in comparison to a single group of threads. In the case of two Intel Xeon E5649 CPUs, two groups of threads (with 6 threads in each group) give the speedup of 1.32 over a single group of threads. Finally, by comparing execution times in the last three rows of Table 6 we conclude that the utilization of SSE vectorization improves the performance by 1.41 and 1.76 times for Intel Westmere and AMD Interlagos, respectively.

7. Parallelization of MPDATA on GPU

7.1. Levels of parallelization

The increasing complexity and diversity of hardware platforms have severe implications for the design of parallel algorithms. Our research in the area of GPU parallelization of MPDATA includes several key issues, such as alleviating bandwidth constraints on CPU–GPU communications, optimizing access to the GPU hierarchical memory, as well as maximizing the utilization of GPU computing resources. The proposed method of parallelization is based on three levels of GPU parallel hierarchy (Fig. 9):

1. overlapping data transfers between the host memory and GPU global memory with GPU computations;
2. parallel computations across threads running on GPU cores;
3. dynamic vectorization within a GPU thread.

The first level requires us to apply an appropriate decomposition of data domain into streams, in order to use the streams processing mechanism. It allows us to alleviate bandwidth constraints of PCIe connection between CPU and GPU.

The second level concerns parallel processing of GPU threads, which are assembled into work-groups. The proposed approach assumes the usage of 1- or 2-dimensional work-groups. The lack of synchronization between work-groups executed within a single GPU kernel implies the necessity of using one of the two synchronization strategies. They assume either the usage of threads synchronization within a work-group, which requires additional computations for halo areas in each work-group, or data synchronization at the level of kernel execution. In our research, we decided to use the second strategy, which allows us to avoid additional computations across halos within a work-group by providing the decomposition of MPDATA calculations into series of kernels. Each kernel corresponds to a certain stage in the data dependency graph shown in Fig. 2.

The last level of parallelization is based on vectorization of computations within a single GPU thread. The proposed approach gives the possibility of using different sizes of vectors for various GPU kernels. The vector data types are hardware supported by AMD graphics processors only. However, the usage of this technique on NVIDIA GPUs is also worth considering. By increasing the amount of computations within a single GPU thread, this technique allows for reducing overheads of access to GPU global and local memories.

7.2. Decomposition of MPDATA for GPU

Aiming at utilizing efficiently all the three levels of parallel GPU hierarchy, we apply the algorithm decomposition strategy across data and computation domains (Fig. 10). Our strategy is based on the partitioning of MPDATA into asynchronously executed S_c streams, where each stream consists of b synchronously processed data blocks, each of size $S_b \times l$. Each stream allocates a separate area of GPU global memory, which is reused by all the blocks of the stream. Every block is executed by

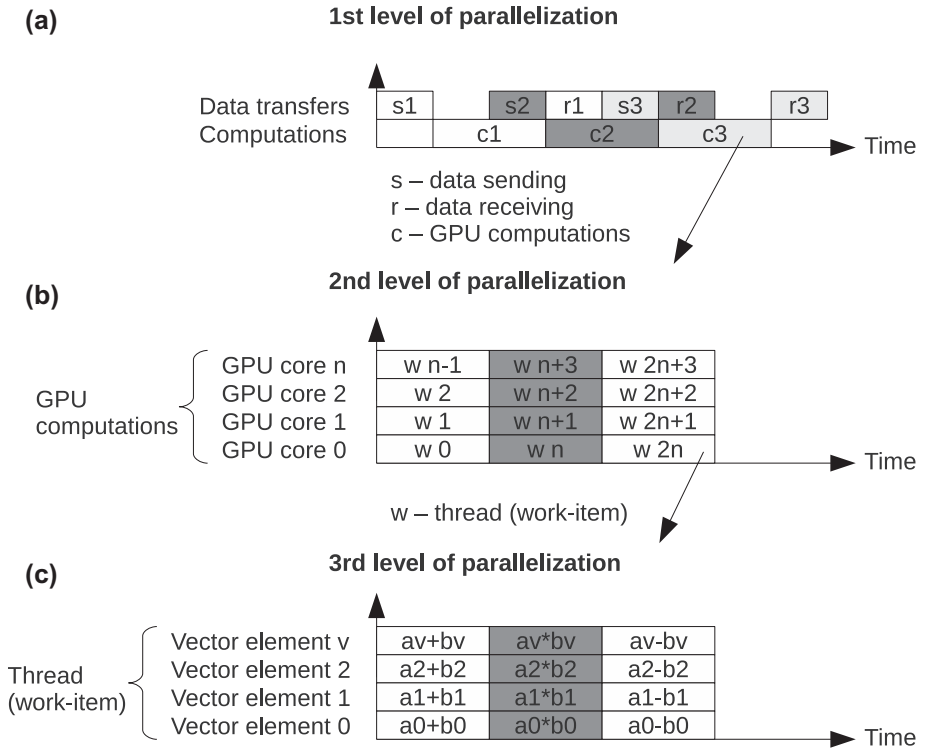

Fig. 9. Levels of parallelization for MPDATA algorithm.

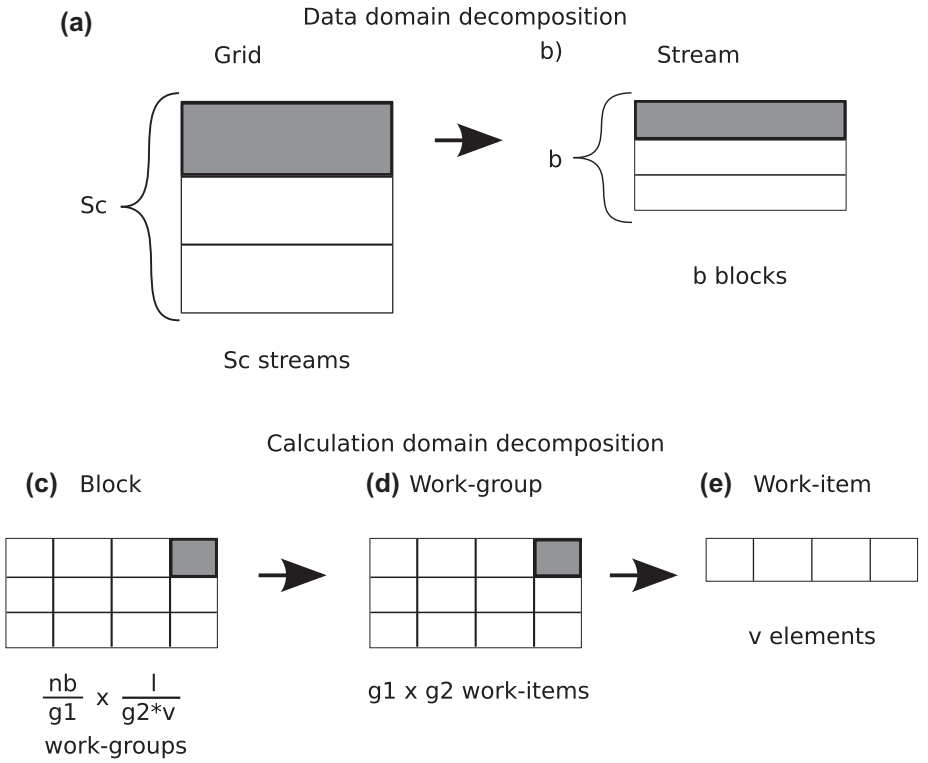

Fig. 10. Decomposition in data domain (a–b) and computation domain (c–e).

groups of work-items (GPU threads). Finally, each thread computes v elements in the SIMD fashion, where v is the vector size.

The idea of decomposition in the data domain aims at preserving all the data dependencies given by the dependency graph (Fig. 2), what is required to provide the data integrity of all 16 stages. This decomposition is applied to the grid of size $n \times l$. Concerning communications between the main memory and GPU global memory, the data domain of MPDATA is recursively partitioned in i-dimension into streams and blocks, where each block is of size $S_b \times l$. The size S_b is given by the following equation:

$$S_b = ihT + n_b + ihB, \tag{15}$$

where $n_b = n/(S_c * b)$ is the number of rows in each block (without halo areas), while ihT and ihB are numbers of halo rows required to compute all 16 stages. Using an 1D decomposition here has several advantages. In particular, it allows for avoiding halos on the left and right sides of the grid, and makes it easy to transfer data blocks, which are always continuous in the memory space. Moreover, modern GPUs have enough memory (a few GB) to store all the necessary blocks for the 2D MPDATA algorithm.

The computation domain is partitioned in both dimensions, where each data block is computed by a series of $\frac{n_b}{g_1} \times \frac{l}{g_2 * v}$ work-groups of size $g_1 \times g_2$ each. To increase the level of data locality, a work-group makes a copy of data from the GPU global memory to the local memory. For example, for stage 3 of MPDATA the requirements to the amount of GPU local memory can be expressed as:

$$M_{local} = (2 * g_1 * g_2 + (g_1 + 1) * g_2 + g_1 * (g_2 + 1)) * v * S_{el}, \tag{16}$$

where S_{el} is size of a single element of matrices (float or double). By using the local memory for this kernel, the number of loads from the GPU global memory is reduced by $2 * g_1 * g_2$.

Each block represents a part (submatrix) of all the matrices in the MPDATA algorithm, and is computed by a single GPU task. Every task is a sequence of GPU kernels which compute different stages of the algorithm. Each GPU task is decomposed into 16 kernels, based on synchronization points and data dependencies. These kernels are executed in a FIFO order with respect to the dependency graph expressing data dependencies between stages. Each kernel can be configured in an individual way considering the following parameters: (i) sizes g_1 and g_2 of work-group; (ii) size v of vectors. The estimation of these parameters is based on the autotuning mechanism, which is also responsible for the estimation of values of the number S_c of streams and the number b of blocks, that are common to all the kernels. It is worth noting that once being estimated, the values of parameters g_1, g_2, v, S_c, and b, together with sizes n and l of the grid, unambiguously determine the total number of work-items for each kernel.

7.3. Management of GPU resources using scheduler

The MPDATA code is characterized by a set of parameters, which can be configured in different ways. This implies the necessity to develop a tool which allows for management of GPU resources transparently, based on a given configuration. For this purpose, a GPU task scheduler is proposed, which is responsible for: (i) creating streams; (ii) dividing stream into tasks, where one task operates on a single block of every matrix of MPDATA; (iii) allocating data buffers for data blocks (a single buffer is allocated for all the blocks within a stream); (iv) running tasks in accordance to the dependency graph; (v) managing data transfers between the host memory and GPU global memory.

The input parameters for the GPU task scheduler are:

- size $n \times l$ of the grid;
- number S_c of streams;
- number b of blocks per stream;
- parameters ihT and ihB for each kernel of MPDATA.

The scheduler output is a series of GPU tasks. A separate FIFO queue is assigned for each stream, so tasks within a stream are executed synchronously, while different streams works asynchronously. Streams are managed by asynchronous OpenCL command queues - one queue per stream [25]. Therefore, each task is described by the following main parameters:

- stream *id*, which a GPU task belongs to;
- task *id*, which determines location of the task in a FIFO queue;
- size of data block corresponding to the task;
- size of data buffer allocated in the GPU global memory;
- address of the data block within each MPDATA matrix when allocated in the host memory.

All tasks are generated by the scheduler in two steps. The first one is used to create streams and divide them into data blocks without considering data dependencies between kernels. The second step extends the data blocks by halo areas, based on the data dependency graph. The sizes of halos are characterized by ihT and ihB parameters. Then, the scheduler allocates

data buffers in the GPU global memory. Since a single buffer is allocated per the whole stream, the buffer size corresponds to a maximum size of blocks within the stream.

The proposed scheduler is responsible for management of computations and data transfers, in order to support the stream processing mechanism providing overlapping communications with computations. For each task, the scheduler calls a sequence of the following operations: (i) data reorganization within data buffers (beginning from the second task in every stream); (ii) sending data blocks from the host memory to the GPU global memory; (iii) computations performed by a sequence of 16 kernels. The data reorganization within a data buffer is responsible for copying its last $ihT + ihB$ rows, which are placed at the beginning of the buffer. As all the kernels within a stream are executed in the FIFO order, the copied chunk of data can be reused by the next block within the stream, without recomputing halo areas. Assuming an ideal case without communication delays and additional calculations, we conclude that the more number of streams the more performance gain can be achieved in this way. But in practice, the performance benefits are counterbalanced by losses which are results of latencies of data transfers, as well as increasing amount of computations for halo areas between different streams. So it is desirable to have a mechanism to determine the optimal number of streams which allows us to take the full advantage of data level parallelism provided by asynchronously operated FIFO queues assigned to different streams.

8. Automatic tuning of MPDATA on GPU

One of the main goals of this work is to provide both code and performance portability across a variety of GPU platforms, including different GPU vendors (NVIDIA, AMD). Our approach requires to develop a configurable code with a set of parameters, which allow us to manage the code execution on a particular GPU architecture. The configuration of parameters is fully automatic by enabling a mechanism for the on-demand mapping of the MPDATA algorithm onto GPU architectures. The manual setting of code parameters is impractical because of a large space of possible solutions. To provide the performance portability across various GPUs, a distinctive configuration is required for each GPU platform. In this section, we propose an autotuning mechanism, which is responsible for generating and evaluating "the best" solution for a given GPU.

8.1. Concept of MPDATA autotuning for GPU

The proposed mechanism is based on both the online and offline approaches, which correspond respectively to mathematical optimizations at runtime, and empirical optimizations when installing MPDATA. A disadvantage of empirical optimizations is the time cost of searching for "the best" code variant, which is usually proportional to the number of variants generated and evaluated. For this reason, our approach combines the empirical search with a performance-model-driven method that would limit the search space.

The concept of our autotuning mechanism is based on four modules (Fig. 11), which jointly generate the final configuration of the algorithm. The proposed mechanism aims at selecting a configuration that provides the shortest execution time of the algorithm. The first module accounts for the generation of the search space corresponding to a set of admissible values for the algorithm parameters. The others modules are responsible for evaluating and searching for "the best" values of these

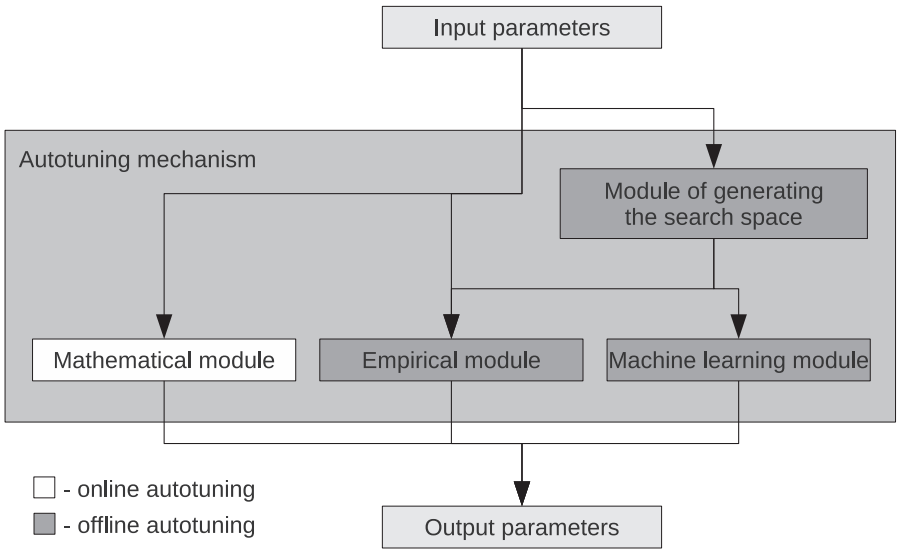

Fig. 11. Concept of our autotuning mechanism.

parameters in an efficient way. The second and third modules are called empirical and machine learning ones, respectively. These modules are the most time consuming parts of the proposed mechanism, because they require to perform time measurements while testing configurations of the algorithm. However, once having been executed for a given GPU platform, these modules have not to be activated any more for subsequent MPDATA runs on this platform. The fourth module, called the mathematical one, is based on the online autotuning methodology, but it has no significant impact on the MPDATA performance.

8.2. Autotuning implementation

The ranges of admissible values of the considered parameters determine the size of the search space; they depend on properties of a particular GPU architecture, the data dependency graph, as well as the MPDATA grid size. The input parameters of the autotuning mechanism are:

- size M_G of GPU global memory;
- size M_L of GPU local memory;
- maximum size g_{max} of work-group, supported by a given GPU architecture;
- maximum sizes g_1^{max} and g_2^{max} of work-groups, in both dimensions;
- sizes *ihT* and *ihB* for each kernel of the algorithm;
- sizes n and l of the MPDATA grid.

There are two groups of output parameters. The first group contains parameters which are applied individually for each of 16 kernels, creating the local search space for each kernel. These parameters include the sizes g_1, g_2 of work-group, and the size v of vectors. All of them are evaluated by the empirical module. The evaluation of work-group sizes is of particular importance for the autotuning mechanism. The range of g_1 and g_2 depends on a GPU platform. For tested GPUs, we identify the following constraints:

$$g_1 \in [1,512] \wedge g_2 \in [1,512], \quad g = g_1 * g_2 \wedge g \leqslant 512. \tag{17}$$

For each kernel, it gives 3280 possible combinations of g_1 and g_2. As a result, the evaluation of work-group sizes is the most time-consuming among all the parameters. The range of vector size v is defined based on vector sizes supported by the OpenCL standard, whence $v \in \{1,2,4,8,16\}$.

The second group consists of parameters applicable to the entire algorithm, such as the number S_c of streams, evaluated by the machine learning module, and the number b of blocks per stream, assessed by the mathematical module. The considered values for S_c are from 1 to 20, which allows us to find the near-optimum solution, while the parameter b is evaluated by the online autotuning, so it does not expand the size of search space for the offline autotuning.

The important parameters here are modified sizes n^* and l^* of the MPDATA grid, which are aligned to the evaluated work-group sizes g_1 and g_2. In fact, the output MPDATA configuration is created for the finite set G of predefined grid sizes given by $n \in 2^i \wedge l \in 2^i$, for $i \in [4,13]$, which gives 100 distinctive combinations of values n and l. Then taking into account the actual grid sizes, the nearest configuration given by n and l from the set G is selected at runtime. For example, when the user runs the MPDATA algorithm with the grid of the actual sizes 900×900, the autotuning mechanism firstly selects the nearest configuration from the set G, so the configuration for the grid of sizes 1024×1024 is selected. Then the actual grid sizes are aligned to the work-group sizes evaluated for the selected configuration, so if the work-group sizes are evaluated as 16×1, the grid sizes are modified to 912×900 at runtime.

Fig. 12 illustrates the global search space for offline autotuning, defined by ranges of all the considered parameters. Its size is given by the sum of sizes of local search spaces for each MPDATA kernel multiplied by ranges of parameters applicable to the entire algorithm: $S_{global} = 16 * 5 * 3280 * 100 * 20 * 1 = 524,800,000$.

Testing empirically all the configurations from the global search space is very expensive. For that reasons, we provide a group of methods which allow us to reduce the search space radically. We focus here on describing the reduction of local search spaces corresponding to each kernel, as the most important. The reduction procedure, which is implemented by the empirical module, applies the following constraints to sizes of work-group:

- $g_2 \geqslant g_1$ – improves the coalesced memory access for the row-mayor matrices used in MPDATA;
- $g_2 mod 16 = 0$, where the value of 16 corresponds to the number of GPU load/store units, and is a divisor of warp size (32 for NVIDIA GPUs) and wavefront size (32 or 64 for AMD GPUs);
- the size of local memory required by a work-group can not exceed the size of local memory available in a particular GPU;
- sizes g_1 and g_2 of work-group can not exceed sizes n and l of MPDATA grid.

For AMD GPUs, by considering only those vector sizes which improve utilization of GPU cores, we restrict ourselves to $v \geqslant 2$ (for double precision).

For instance, the proposed constraints allow us to prune the size of search space for the 3rd kernel from $S_{local} = 5 * 3280 = 16,400$ to $S_{local} = 151$ for the ATI Radeon HD 5870 GPU, and to $S_{local} = 302$ for NVIDIA M2070Q GPU,

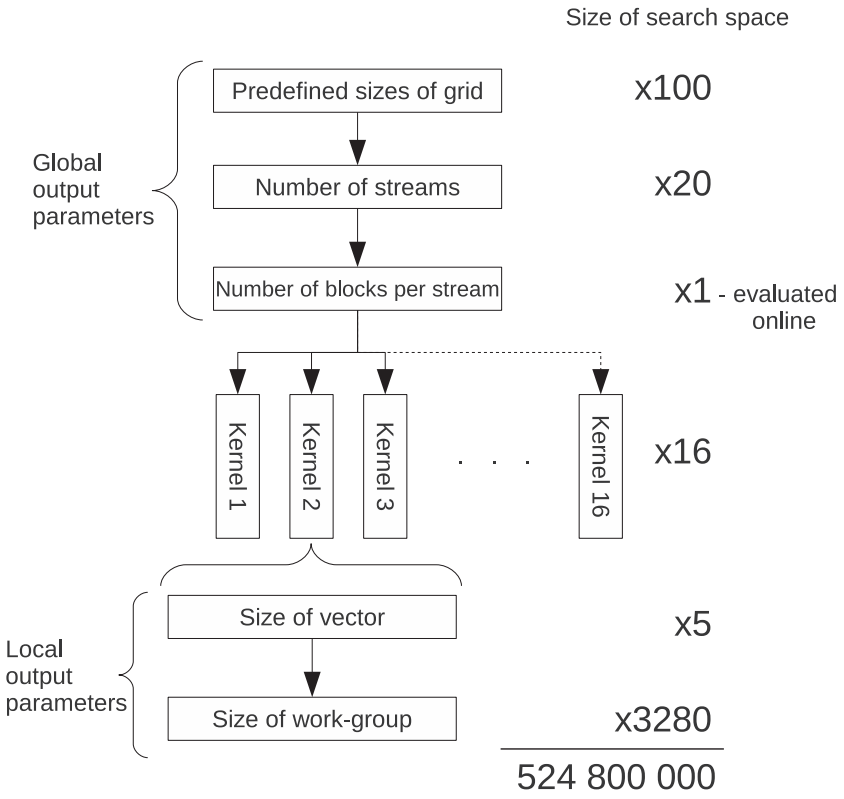

Fig. 12. Global search space for offline autotuning of MPDATA on GPUs.

where the MPDATA grid is of size 1024×1024, and operations are performed in double precision. Taking into account all the considered grid sizes, the average size of S_{local} for the 3rd kernel is 98.7 for AMD, and 202.4 for NVIDIA.

The autotuning mechanism executes the MPDATA algorithm (e.q., 100 time steps of MPDATA) for all possible configurations taken from the search space, to select "the best" configuration corresponding to the shortest execution time. As mentioned before, the search space is defined by the following parameters: sizes of grid, number of streams, size of vector, and sizes of work group. Possible values of these parameters are pruned using constraints previously introduced for size of work groups, and size of vectors. Also, values of the number S_c of streams are pruned using the machine learning module, which uses the results of tests from the past to determine the break point, which gives the value of S_c. In our tests, this value does not exceed 5. In practice, the set of all possible configurations is implemented with 5 nested loops, which correspond to: size of grid, number of streams, different kernels, size of vector, and size of work group. Additionally, the number of blocks per stream is evaluated at runtime, using the mathematical module.

Taking advantage of our autotuning mechanism, the size of the global search space is pruned to $S_{global} = 379,648$ for the ATI Radeon HD 5870 GPU, and to $S_{global} = 965,504$ for the NVIDIA M2070Q GPU, which is about 0.07% and 0.18% of the initial size for the AMD and NVIDIA GPUs, respectively. In consequence, the proposed methods allows for executing the autotuning mechanism in less than 18 h. One must remember that this is performed only once per each GPU device when installing the application.

8.3. Experimental verification of autotuning

The proposed autotuning methodology has been verified experimentally. In particular, the proposed methods of reduction the search space have been tested thoroughly. Example results of such tests are shown in Fig. 13, which confirms the correctness of $g_2 \geqslant g_1$ constraint. These results correspond to stage 4 of MPDATA, performed on the NVIDIA Tesla M2070Q GPU for 100 time steps with grid of size 1024×1024. The presented tests, where the work-group size is fixed at $g = 64$, show that the execution time with work-groups of sizes $g_2 \geqslant g_1$ is shorter than with work-groups of sizes $g_2 < g_1$.

Fig. 14 shows speedups achieved for the autotuned version of MPDATA over a standard version, (i.e. without autotuning) for different sizes of grid and different GPU platforms. The standard version is based on the configuration where the number of streams, the number of blocks per stream, and size of vectors are 1, while work-groups for each kernel are of size 16×16.

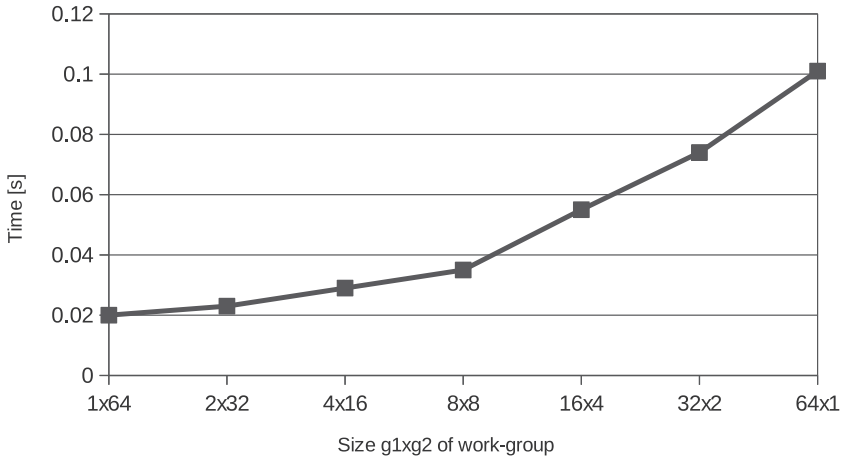

Fig. 13. Relation between execution time and sizes g_1, g_2 for stage 4 of MPDATA.

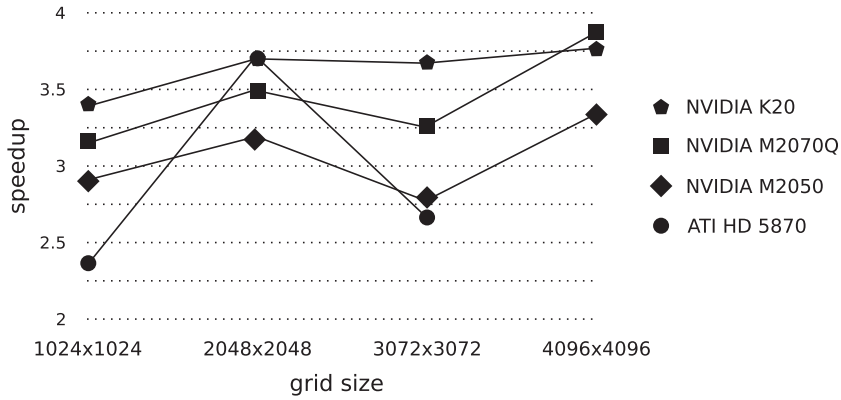

Fig. 14. Speedup of autotuned version of MPDATA over standard version.

The proposed autotuning mechanism enables for achieving the speedup range between 2 and 4, for all the GPU platforms used in experiments. It makes our code more portable across different GPU devices, and allows us to avoid hand-tuning optimizations for each GPU. Our approach to autotuning works well both on AMD and NVIDIA GPUs. The difference between NVIDIA M2050 and M2070Q is relatively small, which results from the fact that these two GPUs are based on the same Fermi architecture. The largest amplitude of changing speedup, and at the same time the smallest increase of performance are observed for ATI HD 5870. It suggests that providing an efficient adaptation of MPDATA to this GPU is more complex than achieving a high performance on other GPUs considered in the paper.

9. Performance results on hybrid CPU–GPU platforms

Table 7 presents the execution time of the 2D MPDATA algorithm for 500 time steps and different sizes of grid, using the hybrid CPU–GPU platforms introduced in Section 3. For CPUs we use the gcc compiler (version 4.4 or 4.6 depending on the platform), while for GPUs it is empowered by CUDA 5.0 Toolkit or AMD APP SDK v2.8 to implement the OpenCL standard (version 1.2). The achieved performance results correspond to the following configurations for each platform:

1. basic serial version implemented on a single CPU core without using the block decomposition and vectorization (compiler optimization option -O3),
2. parallel CPU version using one CPU or two CPUs,
3. parallel GPU version with computations executed on one GPU or two GPUs,
4. parallel hybrid version with computations distributed across CPU and GPU components of each hybrid platform.

For all the platforms, the speedups of parallel versions over the basic serial version are shown in Fig. 15.

Table 7
Execution time (in seconds) of 2D MPDATA on different platforms, for 500 time steps.

Size	Serial	1CPU	2CPUs	1GPU	2CPUs + 1GPU
(a) Intel Xeon E5649 (Westmere) with NVIDIA M2070Q (Fermi)					
1024 × 1024	55.08	7.33	3.84	2.61	1.74
2048 × 2048	220.71	28.66	14.77	9.99	6.71
3072 × 3072	476.49	63.65	32.52	22.33	14.76
4096 × 4096	840.53	120.91	59.67	39.46	26.50

Size	Serial	1CPU	1GPU	1CPU + 1GPU
(b) Intel Xeon E5-2670 (Sandy Bridge-EP) with NVIDIA K20 (Kepler)				
1024 × 1024	36.48	3.15	2.05	1.42
2048 × 2048	158.04	12.02	7.59	5.53
3072 × 3072	320.32	26.15	16.99	13.38
4096 × 4096	580.50	45.99	30.10	23.29
(c) AMD Phenom 1090T (Thuban) with ATI Radeon HD 5870 (Evergreen Cypress XT)				
1024 × 1024	50.32	9.12	5.65	4.73
2048 × 2048	196.67	27.75	23.2	14.47
3072 × 3072	423.21	56.39	48.98	28.96

Size	Serial	1CPU	2CPUs	1GPU	2GPUs	1CPU + 1GPU	2CPUs + 2GPUs
(d) AMD Opteron 6234 (Interlagos) with NVIDIA M2050 (Fermi)							
1024 × 1024	99.24	5.21	2.55	2.81	1.47	2.12	1.38
2048 × 2048	384.68	19.14	9.66	10.83	5.48	8.15	4.74
3072 × 3072	869.91	40.91	20.78	26.12	13.07	18.90	9.45
4096 × 4096	1568.22	74.50	37.81	53.99	22.43	33.71	16.78

An important aspect of these tests is to provide load balancing among CPU and GPU components, taking into account difference in their performance. For a given platform and a fixed grid size, this goal is achieved in an experimental way by selecting accordingly the numbers of rows in the CPU and GPU domains of an MPDATA grid (see Fig. 3) which are distributed across components of a hybrid platform. Starting from the even distribution and using a simple algorithm based on the recursive bisection, it is enough to perform a small number of MPDATA iterations (at most 10) to obtain quite satisfactory results. As a result, the number of rows assigned to CPUs is in range from 34% to 44%, where the lower and upper bounds are obtained for hybrid configurations with respectively one AMD Interlagos CPU and one Intel Sandy Bridge-EP CPU (corresponding to last but one column of Table 7d and last column of Table 7b).

Summarizing the results of these tests, it is worth mentioning that for all the grid sizes and for all target platforms, the hybrid version allows us to achieve the highest performance. The highest performance gains due to distributing computations across both CPU and GPU components are achieved for two platforms: (i) two Intel Xeon E5649 CPUs with one NVIDIA M2070Q GPU (Table 7a and Fig. 15ii) AMD Phenom 1090T with ATI Radeon HD 5870 (Table 7c and Fig. 15c). In the first case, for all the grid sizes, the speedups of the hybrid version over the versions with one GPU and two CPUs are about 1.5 and 2.2, respectively. In the second case, for the largest grid of size 3072 × 3072, the hybrid version is 1.69 and 1.95 times faster than the GPU and CPU versions, respectively.

Table 7d and Fig. 15d show performance results achieved on the platform containing two CPUs (AMD Interlagos) and two GPUs (NVIDIA Fermi). For the grid of size 4096 × 4096, our approach gives the speedup of 93.46 over the basic serial version. For the largest grid size, the hybrid version is about 2.25 times faster then the version with two CPUs, and about 1.33 times faster than the version with two GPUs. The latter is 1.68 times faster than the version with two CPUs.

Comparing all the target platform, the one with two AMD Interlagos CPUs and two NVIDIA Fermi GPUs gives the shortest execution time of the MPDATA algorithm. However, in the case when only one CPU and one GPU are employed, this platform is 1.41–1.49 times slower than the machine using the latest processor architectures - Intel Sandy Bridge-EP and NVIDIA Kepler (see Table 7b and Fig. 15b).

10. Conclusions and future work

EULAG is an established computational model developed for simulating thermo-fluid flows across a wide range of scales and physical scenarios. The dynamic core of EULAG includes the MPDATA advection algorithm and elliptic solver. Rewriting the EULAG dynamic core and replacing conventional HPC systems with heterogeneous clusters using accelerators such as GPUs is a prospective way to improve the efficiency of using the EULAG model in practical simulations. In this work, we investigate aspects of an optimal parallel version of the 2D MPDATA algorithm on modern hybrid architectures with GPU accelerators, where computations are spread across both GPU and CPU components. In order to better utilize features of such computing platforms, exhaustive adaptations of MPDATA computations to hybrid architectures are proposed, taking into account the memory-bounded character of the algorithm.

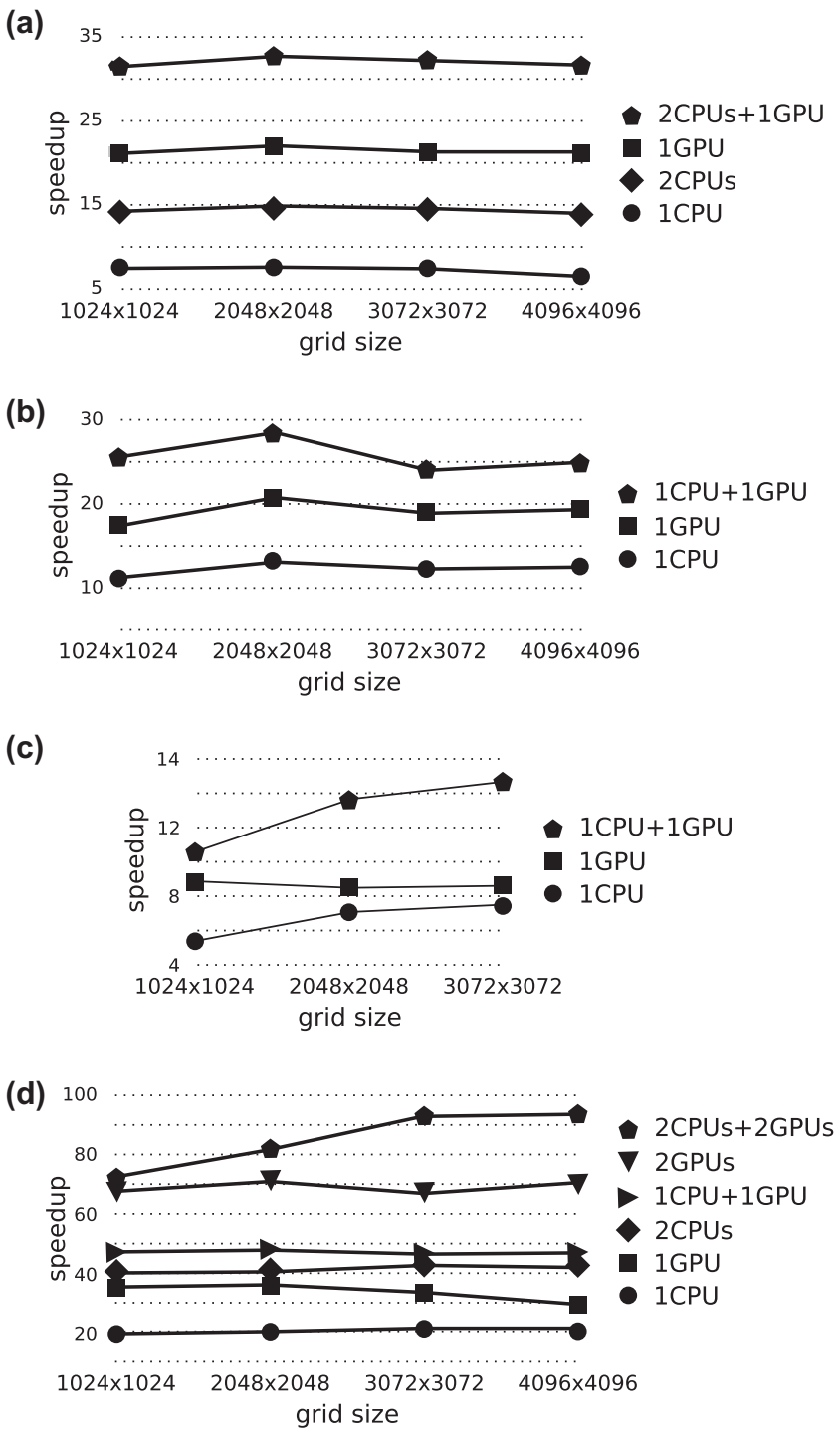

Fig. 15. Speedup of parallel versions over basic serial version, achieved on: (a) Intel Xeon E5649 and NVIDIA M2070Q, (b) Intel Xeon E5-2670 and NVIDIA K20, (c) AMD Phenom 1090T and ATI Radeon HD 5870, (d) AMD Opteron 6234 and NVIDIA M2050.

The proposed adaptations are based on the developed strategies for memory and computing resource management, which allow us to ease memory and communication bounds, and better exploit the theoretical floating point efficiency of hybrid CPU–GPU platforms. Moreover, they provide a portable implementation across different architectures, including CPU and GPU processors from Intel, AMD and NVIDIA vendors.

When adapting MPDATA to hybrid platforms, the main challenge is to reduce traffic over the PCIe bus connecting GPU to CPU. That is why, the proposed decomposition of MPDATA allows for avoiding communications between CPU and GPU within each time step of the MPDATA algorithm at the cost of extra computations corresponding to halo areas. The CPU–GPU cooperation, including communication and synchronization, is now required only after each time step.

The main challenge of CPU parallelization is to take advantage of multicore, vectorization, and cache reuse. For this aim, we propose the block version of the 2D MPDATA algorithm, which combines the space and temporal blocking techniques. This gives possibility to ease memory bounds by increasing the efficient cache reuse, and reducing the main memory traffic. Another technique, which gives a significant performance gain for CPU platforms, is reduction of inter-cache communications across available CPUs, and/or dies within CPUs. In addition, it enables for applying the NUMA "first-touch" policy more efficiently.

When adapting MPDATA to GPUs, three levels of their parallel hierarchy are considered: overlapping data transfer with computations, multithreaded GPU computations, and vectorization of GPU threads. The resulting adaptation is based on a hierarchical decomposition strategy across data and computation domains, with support provided by the developed GPU scheduler which enables for the dynamic management of every level. Finally, using the proposed autotuning methodology, both the code and performance portability among different GPUs are provided, including different GPU vendors.

Hybrid platforms tested in this study contain different numbers of CPUs and GPUs – from two solutions consisting of a single CPU and single GPU, through a platform with two CPUs and one GPU, to the most elaborate configuration containing two CPUs and two GPUs. Processors of different vendors are employed in these systems – both Intel and AMD CPUs, as well as GPUs from NVIDIA and AMD. For all the grid sizes and for all the tested platforms, the hybrid version with computations spread across CPU and GPU components allows us to achieve the highest performance. For the largest MPDATA grids used in our experiments, the speedups of the hybrid versions over GPU and CPU versions vary from 1.30 to 1.69, and from 1.95 to 2.25, respectively.

The achieved performance results have shown good perspectives of using hybrid architectures to the MPDATA algorithm in the 3D case, as well. The future work will focus on adaptation the 3D MPDATA algorithm to clusters with hybrid CPU–GPU nodes. Another option to be investigated are clusters using the Intel Xeon Phi – a recently released [14] high-performance coprocessor which features 61 cores, each supporting 4 hardware threads with 512-bit wide SIMD registers, achieving a peak performance of 1Tflop/s in double precision.

An important aspect of future research is using the fusion of MPDATA stages to reduce overheads of computation. Such a loop fusion optimization [3,17] could improve the locality of MPDATA computation by grouping some stages, which correspond to different loops, into a single loop. Selecting the appropriate number of stages in the MPDATA code is not a trivial issue. Being a part of the EULAG model, the MPDATA algorithm can be configured in many different ways, taking also into account setting various boundary conditions for different stages. In this research, our goal was to provide a fully functional adaptation of MPDATA to hybrid CPU–GPU architectures. To provide the full compatibility of our implementation with the original code, some parts of our code could be included or excluded optionally. For this reason, we decided not to group (fuse) some stages into a single loop, not to mention about fusing all the stages.

At the same time, potential advantages of using this optimization are definitely worth a closer consideration, especially in the case of GPUs. The most important, potential benefit here is possibility to reduce data transfers between the GPU global memory and local memory, as a result of using fewer kernels. However, due to the data dependencies between stages, the more stages are fused into a single kernel, the larger halo area is required. Another consequence is an increasing consumption of local memory by each thread. Taking into account severe restrictions on the size of GPU local memory, and quite a large number of matrices required by the MPDATA algorithm, this limits the GPU occupancy [25] and decreases the resulting performance. The selection of an optimal number of MPDATA kernels (stages) will be the subject of our further research.

Acknowledgments

This work was supported by the Polish National Science Center under Grant No. UMO-2011/03/B/ST6/03500, and by the Polish Ministry of Science and Education under Grant No. BS-1-112-304/99/S.

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] ATI Stream SDK OpenCL Programming Guide. <http://developer.amd.com/gpuassets/ATIStreamSDKOpenCLProgrammingGuide.pdf>
[2] C. Augonnet, J. Clet-Ortega, S. Thibault, R. Namyst, Data-aware task scheduling on multi-accelerator based platforms, in: 2010 IEEE 16th Conf. Parallel and Distributed Systems, 2010, pp. 291–298.
[3] D.F. Bacon, S.L. Graham, O.J. Sharp, Compiler transformations for high-performance computing, ACM Comput. Surv. 26 (4) (1994) 345–420.
[4] M. Blazewicz, S.R. Brandt, M. Kierzynka, K. Kurowski, B. Ludwiczak, J. Tao, J. Weglarz, CaKernel – a parallel application programming framework for heterogenous computing architectures, Sci. Program. 19 (4) (2011) 185–197.
[5] M.M. Christen, Generating and Auto-Tuning Parallel Stencil Code, PhD Thesis, Basel University, 2011.
[6] R. de la Cruz, M. Araya-Polo, J.M. Cela, Introducing the semi-stencil algorithm, Lect. Notes Comput. Sci. 6067 (2010) 496–506.
[7] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, K. Yelick, Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures, in: SC '08 Int. Conf. on High Performance Computing, Networking, Storage and Analysis, 2008, pp. 1–12.

[8] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, K. Yelick, Optimization and performance modeling of stencil computations on modern microprocessors, SIAM Rev. 51 (1) (2009) 129–159.
[9] K. Datta et al, Auto-tuning stencil computations on multicore and accelerators, in: J. Kurzak et al. (Eds.), Scientific Computing with Multicore and Accelerators, CRC Press, 2011, pp. 219–254.
[10] J. Diaz, C. Munoz-Caro, A. Nino, A survey of parallel programming models and tools in the multi and many-core era, IEEE Trans. Parallel Distrib. Syst. 23 (8) (2012) 1369–1386.
[11] C. Ding, Y. He, A ghost cell expansion method for reducing communications in solving PDE problems, in: SC2001 Proc. 2001 ACM/IEEE Conf., 2001.
[12] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, J. Dongarra, From CUDA to OpenCL: towards a performance-portable solution for multi-platform GPU programming, Parallel Comput. 38 (8) (2012) 391–407.
[13] A. Hager, G. Wellein, Introduction to High Performance Computing for Science and Engineers, CRC Press, 2011.
[14] Intel Xeon Phi Coprocessor. <http://software.intel.com/en-us/mic-developer>.
[15] S. Kamil, P. Husbands, L. Oliker, J. Shalf, K. Yelick, Impact of modern memory subsystems on cache optimizations for stencil computations, in: MSP'05 Proc. 2005 Workshop on Memory System Performance, 2005, pp. 36–43.
[16] S. Kamil, C. Chan, L. Oliker, J. Shalf, S. Williams, An auto-tuning framework for parallel multicore stencil computations, in: IPDPS 2010 Proc. 2010 IEEE Int. Symp. on Parallel & Distributed Processing Conf., 2010, pp. 1–12.
[17] K. Kennedy, Fast greedy weighted fusion, Int. J. Parallel Program. 29 (5) (2001) 463–491.
[18] Khronos Home Page. <http://www.khronos.org/opencl>.
[19] K. Kurowski, M. Kulczewski, M. Dobski, Parallel and GPU based strategies for selected CFD and climate modeling models, Environ. Sci. Eng. 3 (2011) 735–747.
[20] J. Kurzak, D. Bader, J. Dongarra (Eds.), Scientific Computing with Multicore and Accelerators, CRC Press, 2011.
[21] Y. Li, J. Dongarra, S. Tomov, A note on autotuning GEMM for GPUs, Lect. Notes Comput. Sci. 5544 (2009) 884–892.
[22] J. Meng, K. Skadron, Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs, in: ICS'09 Proc. 23rd Int. Conf. on Supercomputing, 2009, pp. 256–265.
[23] A. Munshi, B.R. Gaster, T.G. Mattson, J. Fung, D. Ginsburg, OpenCL – Programming Guide, Addison-Wesley, 2011.
[24] A. Nguyen, N. Satish, J. Chhugani, K. Changkyu, P. Dubey, 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs, in: Proc. 2010 ACM/IEEE Int. Conf. for High Performance Computing, Networking, Storage and Analysis, 2010, pp. 1–13.
[25] NVIDIA OpenCL Best Practices Guide. <http://developer.nvidia.com/nvidia-gpu-computing-documentation>.
[26] Z. Piotrowski, A. Wyszogrodzki, P. Smolarkiewicz, Towards petascale simulation of atmospheric circulations with soundproof equations, Acta Geophys. 59 (2011) 1294–1311.
[27] F. Ries, T. De Marco, R. Guerrieri, Triangular matrix inversion on heterogeneous multicore systems, IEEE Trans. Parallel Distrib. Syst. 23 (1) (2012) 177–184.
[28] G. Rivera, Ch.-W. Tseng, Tiling optimizations for 3D scientific computations, in: SC'00 Proc. 2000 ACM/IEEE Conf. on Supercomputing, 2000.
[29] K. Rojek, L. Szustak, Parallelization of EULAG model on multicore architectures with GPU accelerators, Lect. Notes Comput. Sci. 7204 (2012) 391–400.
[30] K. Sato, H. Takizawa, K. Komatsu, H. Kobayashi, Automatic tuning of CUDA execution parameters for stencil processing, in: K. Naono, K. Teranishi, J. Cabazos, S. Reiji (Eds.), Software Automatic Tuning: From Concepts to State-of-the-Art Results, Springer, 2010, pp. 209–228.
[31] P. Smolarkiewicz, W. Grabowski, The multidimensional positive definite advection transport algorithm: nonoscillatory option, J. Comput. Phys. 86 (1990) 355–375.
[32] P. Smolarkiewicz, L Margolin, MPDATA: a finite-difference solver for geophysical flows, J. Comput. Phys. 140 (1998) 459–480.
[33] P. Smolarkiewicz, J. Prusa, Forward-in-time differencing for fluids: simulation of geophysical turbulence, in: D. Drikakis, B.J. Guertz (Eds.), Turbulent Flow Computation, Kluwer Academic Publishers, Dordrecht, 2002, pp. 207–240.
[34] P. Smolarkiewicz, Multidimensional positive definite advection transport algorithm: an overview, Int. J. Numer. Methods Fluids 50 (2006) 1123–1144.
[35] A.R. Surve, A.R. Khomane, S.D. Cheke, Energy awareness in HPC: a survey, Int. J. Comput. Sci. Mobile Comput. 2 (3) (2013) 46–51.
[36] J. Treibig, G. Wellein, G. Hager, Efficient multicore-aware parallelization strategies for iterative stencil computations, J. Comput. Sci. 2 (2011) 130–137.
[37] D. Unat, X. Cai, S.B. Baden, Mint: realizing CUDA performance in 3D stencil methods with annotated C, in: ICS '11 Proc. Int. Conf. on Supercomputing, 2011, pp. 214–224.
[38] S. Venkatasubramanian, R. Vuduc, Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems, in: ICS'09 Proc. 23rd Int. Conf. on Supercomputing, 2009, pp. 244–255.
[39] M. Vinas, J. Lobeiras, B.B. Fraguela, M. Arenaz, M. Amor, J.A. Garca-Rodrguez, M.J. Castro, R. Doallo, A multi-GPU shallow-water simulation with transport of contaminants, Concurrency Comput: Pract. Experience 25 (8) (2013) 1153–1169.
[40] M. Wittmann, G. Hager, J. Treibig, G. Wellein, Leveraging shared caches for parallel temporal blocking of stencil codes on multicore processors and clusters, Parallel Process. Lett. 20 (4) (2010) 359–376.
[41] D. Wojcik, M. Kurowski, B. Rosa, M. Ziemianski, A Study on paralllel performance of the EULAG F90/F95 code, Lect. Notes Comput. Sci. 7204 (2012) 419. 227.
[42] M. Wozniak, T. Olas, R. Wyrzykowski, Parallel implementation of conjugate gradient method on graphics processors, Lect. Notes Comput. Sci. 6067 (2010) 125–135.
[43] R. Wyrzykowski, K. Rojek, L. Szustak, Model-driven adaptation of double-precision matrix multiplication to the cell processor architecture, Parallel Comput. 38 (2012) 260–276.
[44] R. Wyrzykowski, K. Rojek, L. Szustak, Using blue gene/P and GPUs to accelerate computations in the EULAG model, Lect. Notes Comput. Sci. 7116 (2012) 662–670.
[45] A. Wyszogrodzki, Z. Piotrowski, W. Grabowski, Parallel implementation and scalability of cloud resolving EULAG model, Lect. Notes Comput. Sci. 7204 (2012) 252–261.