GPU implementation of algorithm SIMPLE-TS for calculation of unsteady, viscous, compressible and heat-conductive gas flows

Kiril S. Shterev

Institute of Mechanics, Bulgarian Academy of Sciences,
Acad. G. Bonchev Str., Block 4, Sofia 1113, Bulgaria

Abstract

The recent trend of using Graphics Processing Units (GPU’s) for high performance computations is driven by the high ratio of price performance for these units, complemented by their cost effectiveness. At first glance, computational fluid dynamics (CFD) solvers match perfectly to GPU resources because these solvers make intensive calculations and use relatively little memory. Nevertheless, there are scarce results about the practical use of this serious advantage of GPU over CPU, especially for calculations of viscous, compressible, heat-conductive gas flows with double precision accuracy. In this paper, two GPU algorithms according to time approximation of convective terms were presented: explicit and implicit scheme. To decrease data transfers between device memories and increase the arithmetic intensity of a GPU code we minimize the number of kernels. The GPU algorithm was implemented in one kernel for the implicit scheme and two kernels for the explicit scheme. The numerical equations were put together using macros and optimization, data copy from global to private memory, and data reuse were left to the compiler. Thus keeps the code simpler with excellent maintenance. As a test case, we model the flow past squares in a microchannel at supersonic speed. The tests show that overall speedup of AMD Radeon R9 280X is up to 102x compared to Intel Core i5-4690 core and up to
184x compared to Intel Core i7-920 core, while speedup of NVIDIA Tesla M2090 is up to 11x compared to Intel Core i5-4690 core and up to 20x compared to Intel Core i7-920 core. Memory requirements of GPU code are improved compared to CPU one. It requires 1[GB] global memory for 5.9 million finite volumes that are two times less compared to C++ CPU code. After all the code is simple, portable (written in OpenCL), memory efficient and easily modifiable moreover demonstrates excellent performance.

Keywords: GPU, OpenCL, SIMPLE-TS, double precision, unsteady, viscous, compressible and heat-conductive flow, Navier-Stokes-Fourier equations

PACS: 47.11.Df, 47.45.Gx, 47.40.Ki

2000 MSC: 65Y05, 65Y10, 65Y20, 68M20, 68W10, 76M12, 76N15

1. Introduction

Computational analysis of fluid dynamics problems depends strongly on the computational resources [1]. The computational demands are related mainly to the floating point performance and the memory size.

In the last few years, the performance of Graphics Processing Units (GPU’s) overcame significantly the performance of Central Processor Units (CPUs), see [2], [3]. At first glance, computational fluid dynamics (CFD) solvers match perfectly to GPU resources, because these solvers make intensive calculations and use relatively little memory. Nevertheless, there are scarce results about the practical use of this serious advantage of GPU over CPU, especially for calculations of viscous, compressible, heat-conductive gas flows with double precision accuracy. The reported speedups of GPU code to CPU code strongly depend on the mathematical model and the precision of floating point operations. The calculation of Euler flow with single precision in [4] demonstrates speedup of over 40x when comparing GPU NVIDIA 8800GTX and CPU Intel Core 2 Duo. The calculation of incompressible fluid demonstrates speedup up to 48 times compared to serial CPU code. Zaspel and Griebel [5] report 4.0x speedup when comparing GPU NVIDIA C2050 and two Intel Xeon X5650 (six cores CPU)
that is equivalent to a speedup of 48x when the GPU code is compared to one CPU core (serial code). Cohen and Molemaker \[6\] report 8.5x speedup when comparing GPU code run on NVIDIA Quadro FX5800 and CPU parallel code run on an 8-core dual socket Xeon E5420 at 2.5GHz. The equivalent speedup is 42x when the GPU code is compared to one CPU core (serial code). The calculation of compressible fluid reached slower speedup on GPU than Euler and incompressible flow calculations. Salvadore, Bernardini, and Botti demonstrate speedup of 11x when comparing GPU NVIDIA Tesla S2070 and serial code executed on Intel Xeon X5650, \[7\]. Liang, Liu, and Yuan calculate the seven-equation model for compressible two-phase flow on NVIDIA Tesla C2075 GPU. The GPU code is 31x faster compared with one Intel Xeon Westmere 5675 CPU core, see \[8\]. The reported speedups depend on the calculated problem and used hardware.

In this paper is presented GPU algorithm for calculation of unsteady, viscous, compressible and heat-conductive gas. The algorithm is a mix of a couple of ideas. The first idea is the minimization of data transfers between memories. We copy all simulation data to the GPU once at the beginning of the application. Therefore, almost no GPU ↔ CPU data transfers are necessary during the simulation, similar as \[5\]. Data transfers between global and local device memories are another possible bottleneck. GPU version of algorithm SIMPLE-TS is developed so that minimize number of kernels to one (see Fig. 2) or two (see Fig. 1). The algorithm SIMPLE-TS is developed to be easily parallel organized that makes a possible realization of this minimal kernel concept up to one or two kernels. As a result data transfers between memories of host ↔ device and global ↔ local/private memories of the device are minimized. The other idea is to left optimization to the compiler. We put together numerical equations using macros. As a result floating point operations per equation reached up to 388. The optimization, data copy from global to private memory, and data reuse were left to the compiler. Thus keeps the code simpler with excellent maintenance.

The proposed concept was applied to different approximation schemes of convective terms. According to time, explicit scheme (Forward Euler) and im-
plicit scheme (Backward Euler) convective terms approximation. On the other side, upwind first order scheme and Total Variation Diminishing (TVD) second order scheme, with Van Leer limiter [9] approximate convective terms by space.

The portability of the code is important to run the code on different devices with none or minimal corrections. To this aim GPU code was written in OpenCL (Open Computing Language). OpenCL is a royalty-free standard for cross-platform, parallel programming of modern processors found in personal computers, servers, and handheld/embedded devices (see [10]). OpenCL implementers are Intel, Texas Instruments, Marvell, Apple Inc., NVIDIA Corporation, MediaTek Inc, QUALCOMM, AMD, Altera Corporation, Vivante Corporation, Xilinx Inc., ARM Limited, Imagination Technologies, STMicroelectronics International NV, IBM Corporation, Creative Labs and Samsung Electronics. One can view a complete list of companies and their conformant products in [11]. OpenCL gives the possibility of Portable Heterogeneous programming of diverse compute resources. One code tree can be executed on CPUs, GPU’s, DSPs, FPGA, and hardware. Can be organized dynamically interrogate system load and balance work across available processors. One can find out more information about OpenCL programming in [10], [12], [13]. On the other hand, CUDA (Compute Unified Device Architecture) language is widely used from the scientific community to calculate computationally expensive problems, see [14], [15], [16]. Contrary OpenCL the CUDA is supported only by NVIDIA devices, see [17]. The GPU code in presented paper was written in OpenCL and performance was obtained on AMD GPU (AMD Radeon R9 280X) and NVIDIA GPU (NVIDIA Tesla M2090). The terminology related to GPU used here is according OpenCL.

CPU serial code performance was obtained on CPU Intel Core i7-920 and CPU Intel Core i5-4690 while GPU code performance was obtained on GPU AMD Radeon R9 280X and GPU NVIDIA Tesla M2090. Both codes use double precision floating point operations.
2. Continuum model equations

A two dimensional system of equations describing the unsteady flow of viscous, compressible, heat-conductive fluid can be expressed in a general form as follows:

\[
\frac{\partial \rho}{\partial t} + \frac{\partial (\rho u)}{\partial x} + \frac{\partial (\rho v)}{\partial y} = 0 \tag{1}
\]

\[
\frac{\partial (\rho u)}{\partial t} + \frac{\partial (\rho u^2)}{\partial x} + \frac{\partial (\rho uv)}{\partial y} = \rho g_x - A \frac{\partial p}{\partial x} + B \left\{ \frac{\partial}{\partial x} \left( \Gamma \frac{\partial u}{\partial x} \right) + \frac{\partial}{\partial y} \left( \Gamma \frac{\partial u}{\partial y} \right) - \frac{2}{3} \frac{\partial}{\partial x} \left[ \Gamma \left( \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) \right] \right\} \tag{2}
\]

\[
\frac{\partial (\rho v)}{\partial t} + \frac{\partial (\rho uv)}{\partial x} + \frac{\partial (\rho vv)}{\partial y} = \rho g_y - A \frac{\partial p}{\partial y} + B \left\{ \frac{\partial}{\partial x} \left( \Gamma \frac{\partial v}{\partial x} \right) + \frac{\partial}{\partial y} \left( \Gamma \frac{\partial v}{\partial y} \right) - \frac{2}{3} \frac{\partial}{\partial y} \left[ \Gamma \left( \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) \right] \right\} \tag{3}
\]

\[
\frac{\partial (\rho T)}{\partial t} + \frac{\partial (\rho u T)}{\partial x} + \frac{\partial (\rho v T)}{\partial y} = C^{T1} \left[ \frac{\partial}{\partial x} \left( \Gamma \lambda \frac{\partial T}{\partial x} \right) + \frac{\partial}{\partial y} \left( \Gamma \lambda \frac{\partial T}{\partial y} \right) \right] + C^{T2} \Gamma \Phi + C^{T3} \frac{Dp}{Dt} \tag{4}
\]

\[
p = \rho T \tag{5}
\]

where:

\[
\Phi = 2 \left[ \left( \frac{\partial u}{\partial x} \right)^2 + \left( \frac{\partial v}{\partial y} \right)^2 \right] + \left( \frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \right)^2 - \frac{2}{3} \left( \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right)^2 \tag{6}
\]

\(u\) is the horizontal component of velocity, \(v\) is the vertical component of velocity, \(p\) is pressure, \(T\) is temperature, \(\rho\) is density, \(t\) is time, \(x\) and \(y\) are coordinates of a Cartesian coordinate system. The parameters \(A, B, g_x, g_y, C^{T1}, C^{T2}, C^{T3}\) and diffusion coefficients \(\Gamma\) and \(\Gamma^\lambda\), given in Eqs. \(1\)-\(5\), depend on the gas model and the equation in non-dimensional form. Upwind first order scheme and Total Variation Diminishing (TVD) second order scheme, with Van Leer limiter, approximate convective terms. A second order central difference scheme approximate diffusion terms. According time explicit scheme (Forward Euler) and implicit scheme (Backward Euler) approximate convective terms,
while implicit scheme (Backward Euler) approximate diffusion terms. This approach is a typical approach for approximation of partial differential equations of convective-diffusion type [18].

3. Porting algorithm SIMPLE-TS to GPU

GPU algorithm development requires an understanding of specifics of the algorithm for implementation and target device, GPU in this case. Firstly is presented GPU implementation of algorithm SIMPLE-TS, subsection 3.1. Next subsection 3.2 present implementation of numerical equations. After that is presented GPU specifics that was taken into account, subsections 3.3. Finally, in this section are presented important tips and tricks that can increase performance significantly, subsection 3.4.

3.1. Algorithm SIMPLE-TS

The algorithm SIMPLE-TS [19] is developed with the idea of easy parallel implementation. It is an iterative Jacobi method; however SIMPLE-TS is faster than SIMPLE [20] and PISO [21] and do not need under-relaxation coefficients to ensure convergence.

In the early stage of development of GPU algorithm SIMPLE-TS used only implicit scheme to approximate convective and diffusion terms. The internal loop for calculation of time step was in a single kernel, see loop 2 on Fig. 2. This work is reported in [22]. The presented here version of the corresponding algorithm is with improved performance of a couple of times.

In this paper are presented and tested algorithm with different approximations of convective terms. According time explicit scheme (Forward Euler) and implicit scheme (Backward Euler) approximate convective terms. On the other side according space upwind first order scheme and Total Variation Diminishing (TVD) second order scheme, with Van Leer limiter approximate convective terms. After all four variants of SIMPLE-TS algorithms are tested, which are noted as follow:
- explicit TVD second-order scheme - approximate convective terms with explicit (Forward Euler) and TVD second-order scheme
- explicit upwind first-order scheme - approximate convective terms with explicit (Forward Euler) and upwind first-order scheme
- implicit TVD second-order scheme - approximate convective terms with explicit (Backward Euler) and TVD second-order scheme
- implicit upwind first-order scheme - approximate convective terms with explicit (Backward Euler) and upwind first-order scheme

Explicit and implicit schemes possess well-known advantages and disadvantages. Explicit scheme compared to the implicit scheme are less stable for fast flows, but are simpler to program and requires less computational efforts. Explicit scheme corresponds very well to TVD second order schemes. TVD schemes are applicable for calculation of steady, slow or moderate fluid flows. They reduce the number of nodes in computation domain significantly, because of it is second-order accuracy in space. On the other side TVD scheme increase the number of floating point operations, see Table 1 and Table 2. Explicit TVD second-order scheme reduces approximately two times floating point operations compared to implicit one. An explicit TVD second-order scheme is recommended for calculation of steady, slow or moderate fluid flows. The fast flows, where explicit TVD second-order scheme obtains physical unrealistic osculations, can be calculated with an implicit upwind first-order scheme. The implicit upwind first-order scheme is the most stable of discussed schemes.

| Approximation scheme   | Convective terms | loop 2 |
|------------------------|------------------|--------|
| TVD second-order       | 687              | 618    |
| upwind first-order     | 143              | 486    |

Table 1: Number of floating point operations for GPU algorithm SIMPLE-TS, when convective terms are approximated with explicit scheme and without implementation of boundary conditions in numerical equations.
Table 2: Number of floating point operations for GPU algorithm SIMPLE-TS, when convective terms are approximated with implicit scheme and without implementation of boundary conditions in numerical equations.

| Approximation scheme | loop 2 |
|----------------------|--------|
| TVD second-order     | 1229   |
| upwind first-order   | 687    |

The number of kernels (functions executed on GPU) of explicit and implicit schemes is different. The implicit scheme execute one kernel on GPU that do all calculations of loop 2, see Fig. 2. The explicit scheme calculate convective terms once per time step. Therefore explicit scheme needs two kernels: one to calculate convective terms and other to calculate loop 2, see Fig. 1. The convective terms are calculated before loop 2, the results are stored in GPU global memory and reused in loop 2.
CPU (serial)
Initialize variables.
Start loop 1:
Set the initial condition for the calculated time step.

Calculate convective terms of velocities and temperature equations, which are approximated with explicit scheme.

Start loop 2 (calculate a state for a new time step):
Calculate diffusion fluxes.
Calculate pseudo velocities (velocities, without pressure term), coefficients for pressure equation.

Start loop 3:
Calculate the coupled equations for energy and pressure.
Stop loop 3. In most cases two iterations are sufficient.
Calculate velocities using pseudo velocities and pressure (calculated within loop 3).
Compute density, using pressure and temperature calculated within loop 3.

Convergence of loop 2: Check for convergence of the iteration process for the current time step.
Convergence of loop 1: If the final time is not reached continue.

GPU code
Initialize variables.
Start loop 1:
Set the initial condition for the calculated time step.

Queue kernel for execution on GPU to calculate convective terms:
Calculate convective terms of velocities and temperature equations, approximated with explicit scheme and store data in global device memory.
Finish kernel execution.
Start loop 2 (calculating a state for a new time step):
Queue kernel for execution on GPU to do calculations in loop 2:
Calculate equation for energy.
Calculate pseudo velocities (velocities, without pressure term), coefficients for pressure equation and store data in local memory.
Calculate equation for pressure.
Calculate velocities using pseudo velocities and pressure.
Finish kernel execution.

Convergence of loop 2: Check for convergence of the iteration process for the current time step.
Convergence of loop 1: If the final time is not reached continue.

Figure 1: Algorithm SIMPLE-TS for CPU (serial) and GPU that use explicit approximation scheme for convective terms.
CPU (serial) Initialize variables.  
Start loop 1: 
Set the initial condition for the calculated time step. 
Start loop 2 (calculating a state for a new time step): 
  Calculate convective and diffusion fluxes. 
  Calculate pseudo velocities (velocities, without pressure term), coefficients for pressure equation. 
Start loop 3: 
  Calculate the coupled equations for energy and pressure. 
Stop loop 3. In most cases two iterations are sufficient.  
Calculate velocities using pseudo velocities and pressure (calculated within loop 3).  
Compute density, using pressure and temperature calculated within loop 3.  
Convergence of loop 2: Check for convergence of the iteration process for the current time step.  
Convergence of loop 1: If the final time is not reached continue.  

GPU code Initialize variables.  
Start loop 1: 
Set the initial condition for the calculated time step. 
Start loop 2 (calculating a state for a new time step): 
  Run a kernel on GPU: 
    Calculate equation for energy. 
    Calculate pseudo velocities (velocities, without pressure term), coefficients for pressure equation and store data in local memory. 
    Calculate equation for pressure. 
  Calculate velocities using pseudo velocities and pressure. 
Finish kernel execution.  
Convergence of loop 2: Check for convergence of the iteration process for the current time step.  
Convergence of loop 1: If the final time is not reached continue.  

Figure 2: Algorithm SIMPLE-TS for CPU (serial) and GPU that use implicit approximation scheme for convective terms.
A domain decomposition (data partitioning) approach is used to separate calculations between work groups. A subdomain corresponds to each work group, Fig. 3. Brandvik and Pullan [23] calculate each subdomain by fix index of x-axes and z-axes to a work item (thread) of a group and doing iteration in kernel along y-axes. They keep neighbors data in local memory (shared memory, according CUDA terminology) and write results straight to the global memory. This approach is partially adopted in presented algorithm. The difference is that we keep temporary arrays in local memory, while neighbors data was copied straight from global memory. Iteration in kernel along the y-axis have important advantages:

- keep temporary variables as pseudo velocities and pressure coefficients in local memory, instead of slower global memory. This decrease global memory usage and global ↔ local memory data transfers.
- increase number of floating point operations in a kernel (see Table 1 and Table 2) and reuse copied data from global memory to private memory.

On the other hand, some preliminary calculations of a subdomain have to be done. After all the influence of number of rows per sub-domain over performance is important and is investigated in Section 5 further in paper.

Figure 3: Domain decomposition of computational domain.
3.2. Numerical equations

Mapping SIMPLE-TS to GPU needs to take into account domain decomposition, sequence of the nodes in mesh calculation and device specifics. SIMPLE-TS and numerical equations derivation are presented in details in [19] while here are analysed and mapped to GPU.

The algorithm has to correspond to GPU architecture specific to reach good performance. GPU device architecture use Single Instruction, Multiple Data (SIMD) parallelism, in which multiple processors execute the same instructions on different pieces of data. Therefore, the algorithm SIMPLE-TS have to be re-organised as fully Jacobi iterative solver. Therefore, right-hand side of numerical equations has to contain only constant data according to the current iteration of loop 2. The equations for pressure and velocities are coupled and needs a detailed analysis. To this aim are presented the definition of variables in the control volume (Fig. 4) and numerical equations used in algorithm SIMPLE-TS (equations from (7) to (30)).

Figure 4: Cell volume
In Fig. 4 the following grid variables are denoted:

- $x^f_i$ - the left frontier x-coordinate of the control volume of node $i$
- $y^f_j$ - bottom frontier y-coordinate of the control volume of node $j$
- $\Delta x_i = x^f_{i+1} - x^f_i$ - step on OX
- $\Delta y_j = y^f_{j+1} - y^f_j$ - step on OY
- $x^v_i$ - x-coordinate of the centre of the control volume of node $i$, $x^v_i = x^f_i + 0.5\Delta x_i$
- $y^v_j$ - y-coordinate of the centre of the control volume of node $j$, $y^v_j = y^f_j + 0.5\Delta y_j$
- $\phi_{i,j}$ - field variables defined at point $(x^v_i, y^v_j)$, pressure, temperature, density, diffusion coefficient and etc.
- $u_{i,j}$ - the horizontal component of velocity defined at point $(x^f_i, y^v_j)$
- $v_{i,j}$ - the vertical component of velocity defined at point $(x^v_i, y^f_j)$
- $F_{i,j}^x$ - the convective mass flux through the surface between control volumes $(i-1,j)$ and $(i,j)$ (in horizontal direction)

$$F_{i,j}^x = \rho_{i,j}^u u_{i,j}^{old} \Delta y_j$$  \hspace{1cm} (7)

- $F_{i,j}^y$ - the convective mass flux through the surface between control volumes $(i,j-1)$ and $(i,j)$ (in vertical direction)

$$F_{i,j}^y = \rho_{i,j}^v v_{i,j}^{old} \Delta x_i$$  \hspace{1cm} (8)

where:

$$\rho_{i,j}^u = \begin{cases} 
\rho_{i-1,j}^{old} + \psi_s \left( \rho_{i-2,j}^{old}, \rho_{i-1,j}^{old}, \rho_{i,j}^{old}, \rho_{i,j+1}^{old}, \Delta x_{i-2}, \Delta x_{i-1}, \Delta x_i, \Delta x_{i+1}, u_{i,j}^{old} \right) & \text{if } u_{i,j} > 0, \\
\rho_{i,j}^{old} + \psi_s \left( \rho_{i-2,j}^{old}, \rho_{i-1,j}^{old}, \rho_{i,j}^{old}, \rho_{i,j+1}^{old}, \Delta x_{i-2}, \Delta x_{i-1}, \Delta x_i, \Delta x_{i+1}, u_{i,j}^{old} \right) & \text{otherwise.}
\end{cases}$$  \hspace{1cm} (9)

$$\rho_{i,j}^v = \begin{cases} 
\rho_{i,j-1}^{old} + \psi_s \left( \rho_{i,j-2}^{old}, \rho_{i,j-1}^{old}, \rho_{i,j}^{old}, \rho_{i,j+1}^{old}, \Delta y_{j-2}, \Delta y_{j-1}, \Delta y_j, \Delta y_{j+1}, v_{i,j}^{old} \right) & \text{if } v_{i,j} > 0, \\
\rho_{i,j}^{old} + \psi_s \left( \rho_{i,j-2}^{old}, \rho_{i,j-1}^{old}, \rho_{i,j}^{old}, \rho_{i,j+1}^{old}, \Delta y_{j-2}, \Delta y_{j-1}, \Delta y_j, \Delta y_{j+1}, v_{i,j}^{old} \right) & \text{otherwise.}
\end{cases}$$  \hspace{1cm} (10)

Density ($\rho$) is computed at the middle points, i.e. $\rho_{i,j}^u$ at point $(x^f_i, y^v_j)$ and $\rho_{i,j}^v$ at point $(x^v_i, y^f_j)$, by using first order upwind scheme or second order TVD scheme
according to convective terms approximation. The implementation of TVD scheme corresponds to presented in [1]. The functions $\psi_{c}^{i,j}$ and $\psi_{s}^{i,j}$ approximate TVD correction to the first order upwind scheme at center or surface of the control volumes, respectively. $\psi_{c}^{i,j}$ is defined at point $(x_{v}^{i}, y_{v}^{j})$ while $\psi_{s}^{i,j}$ is defined at point $(x_{f}^{i}, y_{v}^{j})$:

$$
\psi_{c}(\phi_{1}, \phi_{2}, \phi_{3}, \phi_{4}, \Delta_{1}, \Delta_{2}, \Delta_{3}, v) = \begin{cases} 
0.5\psi \left( \frac{\Delta_{2}(\phi_{2} - \phi_{1})}{\Delta_{1}(\phi_{3} - \phi_{2})} \right) & \text{if } v > 0, \\
-0.5\psi \left( \frac{\Delta_{2}(\phi_{2} - \phi_{1})}{\Delta_{1}(\phi_{3} - \phi_{2})} \right) & \text{otherwise}.
\end{cases}
$$

(11)

$$
\psi_{s}(\phi_{1}, \phi_{2}, \phi_{3}, \phi_{4}, \Delta_{1}, \Delta_{2}, \Delta_{3}, \Delta_{4}, v) = \begin{cases} 
\Delta_{2}\Delta_{3} \psi \left( \frac{(\Delta_{2} + \Delta_{3})(\phi_{2} - \phi_{1})}{(\Delta_{1} + \Delta_{2})(\phi_{3} - \phi_{2})} \right) & \text{if } v > 0, \\
-\Delta_{3}\Delta_{2} \psi \left( \frac{(\Delta_{2} + \Delta_{3})(\phi_{4} - \phi_{3})}{(\Delta_{3} + \Delta_{4})(\phi_{3} - \phi_{2})} \right) & \text{otherwise}.
\end{cases}
$$

(12)

where $\psi(r)$ is TVD limiter. Here is used Van Leer [9] TVD limiter: $\psi(r) = (r + |r|)/(1 + r)$. When convective terms and density in middle points are approximated using upwind first order scheme TVD corrections are null ($\psi_{c} = 0$ and $\psi_{s} = 0$).

SIMPLE-TS use pseudo velocities in the same way as SIMPLER. Therefore, the numerical equations for $u$ and $v$ can be written in form:

$$
u_{i,j} = \hat{u}_{i,j} - a_{i,j}^{u} (p_{i,j} - p_{i-1,j})
$$

(13)

$$
v_{i,j} = \hat{v}_{i,j} - a_{i,j}^{v} (p_{i,j} - p_{i,j-1})
$$

(14)

where $\hat{u}_{i,j}$ and $\hat{v}_{i,j}$ are pseudo velocities:

$$
\hat{u}_{i,j} = \frac{a_{1}^{u}u_{i-1,j}^{old} + a_{2}^{u}u_{i+1,j}^{old} + a_{3}^{u}u_{i,j-1}^{old} + a_{4}^{u}u_{i,j+1}^{old} + b^{u} + u^{explicit}_{i,j}}{a_{0}^{u}}
$$

(15)

$$
\hat{v}_{i,j} = \frac{a_{1}^{v}v_{i-1,j}^{old} + a_{2}^{v}v_{i+1,j}^{old} + a_{3}^{v}v_{i,j-1}^{old} + a_{4}^{v}v_{i,j+1}^{old} + b^{v} + v^{explicit}_{i,j}}{a_{0}^{v}}
$$

(16)

Where the terms $u_{i,j}^{explicit}$ and $v_{i,j}^{explicit}$ contain explicit approximation of the convective terms.
For brevity, here the coefficients for $v$ are given only:

$$b^v = \left( \frac{p_{i,j}^{n-1}}{T_{i,j-1}} \Delta y_j + \frac{p_{i,j-1}^{n-1}}{T_{i,j-1}} \Delta y_{j-1} \right) \frac{\Delta x_i}{2\Delta t} v_{i,j}^{n-1}$$

$$+ B \left( \Gamma_{i,j} \left( u_{i+1,j}^{old} - u_{i+1,j-1}^{old} \right) - \Gamma_{i,j} \left( u_{i,j}^{old} - u_{i,j-1}^{old} \right) 
- \frac{2}{3} \Gamma_{i,j} \left( u_{i+1,j}^{old} - u_{i,j}^{old} \right) + \frac{2}{3} \Gamma_{i,j} \left( u_{i+1,j-1}^{old} - u_{i,j-1}^{old} \right) \right), \quad (17)$$

$$d_{i,j}^v = \frac{A}{a_0^3} \Delta x_i,$$

$$\overline{F}_{i,j} = \tau_{i,j}^{old} \Delta x_i,$$

$$\tau_{i,j}^{old} = 0.5 \left( v_{i,j-1}^{old} + v_{i,j}^{old} \right).$$

The following coefficients correspond to implicit approximation scheme of con-
vective terms:

\[ a_0^v = a_1^v + a_2^v + a_3^v + a_4^v + 0.5 \left( F_{i+1,j}^v - F_{i,j}^v + F_{i+1,j-1}^v - F_{i,j-1}^v \right) \]
\[ + F_{i,j}^y - F_{i,j-1}^y + \left( \rho_{i,j}^{old} \Delta y_j + \rho_{i,j-1}^{old} \Delta y_{j-1} \right) \frac{\Delta x_i}{2 \Delta y} \]
\[ a_0^y = a_1^y + D_{i,j}^{vy}, \quad a_2^y = a_2^y + D_{i+1,j}^{vy}, \quad a_3^y = a_3^y + \frac{4}{3} D_{i,j}^{vy}, \quad a_4^y = a_4^y + \frac{4}{3} D_{i,j+1}^{vy} \]
\[ a_0^{vc} = a_1^{vc} + a_2^{vc} + a_3^{vc} + a_4^{vc} + 0.5 \left( F_{i+1,j}^{vc} - F_{i,j}^{vc} + F_{i+1,j-1}^{vc} - F_{i,j-1}^{vc} \right) + F_{i,j}^y - F_{i,j-1}^y \]
\[ a_1^{vc} = 0.5 \left[ \max \left( 0, F_{i+1,j}^y \right) \right] \]
\[ - F_{i,j}^y \psi_a(v_{i-2,j}^{old}, v_{i-1,j}^{old}, v_{i,j}^{old}, v_{i+1,j}^{old}, \Delta x_{i=2}, \Delta x_{i=1}, \Delta x_i, \Delta x_{i=1}, v_{i,j}^{old}) \]
\[ + \max \left( 0, F_{i+1,j}^y \right) \]
\[ - F_{i,j-1}^y \psi_a(v_{i-1,j}^{old}, v_{i,j}^{old}, v_{i+1,j}^{old}, \Delta x_{i=2}, \Delta x_{i=1}, \Delta x_i, \Delta x_{i=1}, v_{i,j}^{old}) \]
\[ a_2^{vc} = 0.5 \left[ \max \left( 0, -F_{i+1,j}^y \right) \right] \]
\[ - F_{i+1,j}^x \psi_a(v_{i-1,j}^{old}, v_{i,j}^{old}, v_{i+1,j}^{old}, \Delta x_{i=2}, \Delta x_{i=1}, \Delta x_i, \Delta x_{i=1}, v_{i+1,j}^{old}) \]
\[ + \max \left( 0, -F_{i+1,j}^y \right) \]
\[ - F_{i,j-1}^x \psi_a(v_{i-1,j}^{old}, v_{i,j}^{old}, v_{i+1,j}^{old}, \Delta x_{i=2}, \Delta x_{i=1}, \Delta x_i, \Delta x_{i=1}, v_{i+1,j}^{old}) \]
\[ a_3^{vc} = \max \left( 0, F_{i,j}^y \right) - F_{i,j}^y \psi_a(v_{i,j}^{old}, v_{i,j+1}^{old}, \Delta y_j, \Delta y_{j+1}, \Delta y_{j+1}, \Delta y_{j+2}, \Delta y_{j+3}) \]
\[ a_4^{vc} = \max \left( 0, -F_{i,j}^y \right) - F_{i,j}^y \psi_a(v_{i,j}^{old}, v_{i,j+1}^{old}, \Delta y_j, \Delta y_{j+1}, \Delta y_{j+2}, \Delta y_{j+3}) \]
\[ e_{i,j}^{explicit} = 0. \]

The terms \( F_{i,j}^y \) and \( v_{i,j}^{old} = 0.5(v_{i,j}^{old} + v_{i,j+1}^{old}) \) are defined at point \((x_i, y_j)\). With \( D \) we denote the diffusion conductance at cell face. To determine the value of \( D \), we assume that the diffusion \( \Gamma \) varies continuously between the adjacent control volumes and use the bilinear interpolation of the diffusion coefficients at the control volume surfaces to solve \( \Gamma_{i,j}^{old} \) in \( D_{i,j}^{vy} \). For \( D_{i,j}^{vy} \) no interpolation is needed, because the diffusion coefficient \( \Gamma_{i,j}^{old} \) is defined in nodes \((x_i, y_j)\).

\[ D_{i,j}^{vy} = B_i \Gamma_{i,j}^{old} \frac{\Delta y_j + \Delta y_{j-1}}{\Delta x_i + \Delta x_{i-1}} \]
\[ D_{i,j}^{vy} = B_i \Gamma_{i,j}^{old} \frac{\Delta x_i}{\Delta y_{j-1}} \]

(18)

(19)
The following coefficients correspond to explicit approximation scheme of convec-tive terms:

\[
\begin{align*}
a_0^n &= a_1^n + a_2^n + a_3^n + a_4^n + \left( \rho_{i,j}^{old} \Delta y_j + \rho_{i,j-1}^{old} \Delta y_{j-1} \right) \frac{\Delta x_i}{2 \Delta t} \\
 a_1^n &= D_{i,j}^x, \quad a_2^n = D_{i+1,j}^x, \quad a_3^n = \frac{4}{3} D_{i,j}^{xy}, \quad a_4^n = \frac{4}{3} D_{i+1,j+1}^{xy}
\end{align*}
\]

\[
\begin{align*}
v_{i,j}^{explicit} &= -0.5 \left[ F_{i+1,j-1}^x \left( \text{upwind}(v_{i,j}^{n-1}, v_{i+1,j}, u_{i+1,j-1}) \\
&+ (v_{i+1,j}^{n-1} - v_{i,j}^{n-1}) \psi_s(v_{i-1,j}^{n-1}, v_{i,j}^{n-1}, v_{i+1,j}^{n-1}, v_{i+2,j}^{n-1}, \Delta x_{i-1}, \Delta x_i, \Delta x_{i+1}, \Delta x_{i+2}, u_{i+1,j-1}^{n-1}) \right) \\
&+ F_{i+1,j}^x \left( \text{upwind}(v_{i,j}^{n-1}, v_{i+1,j}, u_{i+1,j}) \right) \\
&+ (v_{i+1,j}^{n-1} - v_{i,j}^{n-1}) \psi_s(v_{i-1,j}^{n-1}, v_{i,j}^{n-1}, v_{i+1,j}^{n-1}, v_{i+2,j}^{n-1}, \Delta x_{i-1}, \Delta x_i, \Delta x_{i+1}, \Delta x_{i+2}, u_{i+1,j-1}^{n-1}) \right) \\
&+ 0.5 \left[ F_{i,j-1}^x \left( \text{upwind}(v_{i,j}^{n-1}, v_{i,j-1}, u_{i,j-1}) \\
&+ (v_{i,j-1}^{n-1} - v_{i,j}^{n-1}) \psi_s(v_{i-2,j}^{n-1}, v_{i,j}^{n-1}, v_{i,j+1}^{n-1}, \Delta x_{i-2}, \Delta x_{i-1}, \Delta x_i, \Delta x_{i+1}, u_{i,j-1}^{n-1}) \right) \\
&+ F_{i,j}^x \left( \text{upwind}(v_{i,j}^{n-1}, v_{i,j-1}, u_{i,j-1}) \right) \\
&+ (v_{i,j-1}^{n-1} - v_{i,j}^{n-1}) \psi_s(v_{i-2,j}^{n-1}, v_{i,j}^{n-1}, v_{i,j+1}^{n-1}, \Delta x_{i-2}, \Delta x_{i-1}, \Delta x_i, \Delta x_{i+1}, u_{i,j-1}^{n-1}) \right) \\
&- \Delta x_i \rho_{i,j}^{n-1} v_{i,j}^{n-1} \left[ \text{upwind}(v_{i,j}^{n-1}, v_{i,j+1}, u_{i,j+1}) \\
&+ (v_{i,j+1}^{n-1} - v_{i,j}^{n-1}) \psi_c(v_{i,j-1}^{n-1}, v_{i,j}^{n-1}, v_{i,j+1}^{n-1}, \Delta y_{j-1}, \Delta y_j, \Delta y_{j+1}, u_{i,j+1}^{n-1}) \right) \\
&+ \Delta x_i \rho_{i,j-1}^{n-1} v_{i,j}^{n-1} \left[ \text{upwind}(v_{i,j}^{n-1}, v_{i,j}, u_{i,j}) \\
&+ (v_{i,j}^{n-1} - v_{i,j-1}^{n-1}) \psi_c(v_{i,j-2}^{n-1}, v_{i,j-1}^{n-1}, v_{i,j}^{n-1}, \Delta y_{j-2}, \Delta y_{j-1}, \Delta y_j, u_{i,j}^{n-1}) \right], \\
\end{align*}
\]

\[
(20)
\]

where:

\[
\text{upwind}(\phi_1, \phi_2, v) = \begin{cases} 
\phi_1 & \text{if } v > 0, \\
\phi_2 & \text{otherwise.}
\end{cases}
\]

\[
(21)
\]

$F^x$ and $F^y$ calculated in explicit terms use previous time step values, $v_{i,j}^{explicit}$ is calculated in separate kernel together with $u_{i,j}^{explicit}$ and $T_{i,j}^{explicit}$ before loop 2 (see Fig. 1). The results are stored in global device memory and used as constant variables in loop 2.
The numerical equation for pressure is expressed as follows:

\[
a^p_0 p_{i,j} = (a^{px}_{i,j} p_{i-1,j}^{old} + a^{px}_{i+1,j} p_{i+1,j}^{old} + a^{py}_{i,j} p_{i,j-1}^{old} + a^{py}_{i,j+1} p_{i,j+1}^{old}) \Delta t + b^p, \tag{22}
\]

where:

\[
a^p_0 = \frac{1}{T_{i,j}} \Delta x_i \Delta y_j + (a^{px}_{i,j} + a^{px}_{i+1,j} + a^{py}_{i,j} + a^{py}_{i,j+1}) \Delta t
\]

\[
b^p = \frac{p_{i,j}^{n-1}}{T_{i,j}} \Delta x_i \Delta y_j - (b^{px}_{i+1,j} - b^{px}_{i,j} + b^{py}_{i,j+1} - b^{py}_{i,j}) \Delta t
\]

\[
a^{px}_{i,j} = \rho_{i,j}^u d^{\hat{u}}_{i,j} \Delta x_i, \quad a^{py}_{i,j} = \rho_{i,j}^v d^{\hat{v}}_{i,j} \Delta y_j
\]

\[
b^{px}_{i,j} = \rho_{i,j}^u \hat{u}_{i,j} \Delta x_i, \quad b^{py}_{i,j} = \rho_{i,j}^v \hat{v}_{i,j} \Delta y_j
\]

The discrete equation for temperature is:

\[
a^T_0 T_{i,j} = \Delta t \left( a^T_{i-1,j} T_{i-1,j}^{old} + a^T_{i+1,j} T_{i+1,j}^{old} + a^T_{i,j-1} T_{i,j-1}^{old} + a^T_{i,j+1} T_{i,j+1}^{old} + S^T_{i,j} + T_{i,j}^{explicit} \right)
\]

\[
+ \rho_{i,j}^{n-1} \Delta x_i \Delta y_j, \tag{24}
\]

where the coefficients that correspond to implicit approximation scheme of con-
The finite-difference representation of the source term $S^T$ is expressed as:

$$S^T = S_c^T + S_p^T T_{i,j},$$

(28)

The diffusion coefficients are:

$$D_{i,j}^{T_x} = C_f^{T_1} \Gamma_{i,j}^{\lambda_{old}}|_{x_i} \frac{\Delta y_j}{0.5(\Delta x_j + \Delta x_{j-1})}, \quad D_{i,j}^{T_y} = C_f^{T_1} \Gamma_{i,j}^{\lambda_{old}}|_{y_j} \frac{\Delta x_i}{0.5(\Delta y_j + \Delta y_{j-1})},$$

(26)

A harmonic average between two neighboring nodes is used to calculate $\Gamma_{i,j}^{\lambda_{old}}|_{x_i}$ and $\Gamma_{i,j}^{\lambda_{old}}|_{y_j}$:

$$\Gamma_{i,j}^{\lambda_{old}}|_{x_i} = \frac{(\Delta x_{i-1} + \Delta x_i) \Gamma_{i,j}^{\lambda_{old}}|_{x_{i-1}} + \Delta x_i \Gamma_{i,j}^{\lambda_{old}}|_{x_i}}{\Delta x_{i-1} \Delta x_i \Gamma_{i,j}^{\lambda_{old}}|_{x_{i-1}} + \Delta x_i \Gamma_{i,j}^{\lambda_{old}}|_{x_i}}, \quad \Gamma_{i,j}^{\lambda_{old}}|_{y_j} = \frac{(\Delta y_{j-1} + \Delta y_j) \Gamma_{i,j}^{\lambda_{old}}|_{y_{j-1}} + \Delta y_j \Gamma_{i,j}^{\lambda_{old}}|_{y_j}}{\Delta y_{j-1} \Delta y_j \Gamma_{i,j}^{\lambda_{old}}|_{y_{j-1}} + \Delta y_j \Gamma_{i,j}^{\lambda_{old}}|_{y_j}},$$

(27)

different terms are:

$$a_0^T = \Delta t (a_1^T + a_2^T + a_3^T + a_4^T + F_{i+1,j}^x - F_{i,j}^y + F_{i,j+1}^y)$$

$$+ \rho_{i,j} \Delta x_i \Delta y_j$$

$$a_1^T = \max(0, F_{i,j}^x) - F_{i,j} \psi \left( T_1^{old}, T_1^{old}, T_1^{old}, T_1^{old}, T_1^{old}, T_1^{old}, T_1^{old}, T_1^{old}, \Delta x_{i-1}, \Delta x_i, \Delta x_{i+1}, \Delta y_j, \Delta y_{j+1}, \Delta y_{j+2}, \psi_{i,j}, \psi_{i,j+1} \right)$$

$$+ D_{i,j}^{T_x}$$

$$a_2^T = \max(0, -F_{i+1,j}^x) - F_{i+1,j} \psi \left( T_1^{old}, T_1^{old}, T_1^{old}, T_1^{old}, T_1^{old}, T_1^{old}, T_1^{old}, T_1^{old}, \Delta x_{i-1}, \Delta x_i, \Delta x_{i+1}, \Delta y_j, \Delta y_{j+1}, \Delta y_{j+2}, \psi_{i,j}, \psi_{i,j+1} \right)$$

$$+ D_{i,j}^{T_x}$$

$$a_3^T = \max(0, F_{i,j+1}^y) - F_{i,j+1} \psi \left( T_1^{old}, T_1^{old}, T_1^{old}, T_1^{old}, T_1^{old}, T_1^{old}, T_1^{old}, T_1^{old}, \Delta x_{i-1}, \Delta x_i, \Delta x_{i+1}, \Delta y_j, \Delta y_{j+1}, \Delta y_{j+2}, \psi_{i,j}, \psi_{i,j+1} \right)$$

$$+ D_{i,j}^{T_y}$$

$$a_4^T = \max(0, -F_{i,j+1}^y) - F_{i,j+1} \psi \left( T_1^{old}, T_1^{old}, T_1^{old}, T_1^{old}, T_1^{old}, T_1^{old}, T_1^{old}, T_1^{old}, \Delta x_{i-1}, \Delta x_i, \Delta x_{i+1}, \Delta y_j, \Delta y_{j+1}, \Delta y_{j+2}, \psi_{i,j}, \psi_{i,j+1} \right)$$

$$+ D_{i,j}^{T_y}$$

$$T_{i,j}^{\text{explicit}} = 0.$$  

(25)
where:

\[ S_p^T = 0 \]

\[ S^T_{ci,j} = C_f T^2 I_{i,j} \left\{ 2 \left[ \left( \frac{u^\text{old}_{i+1,j} - u^\text{old}_{i,j}}{\Delta x_i} \right)^2 + \left( \frac{v^\text{old}_{i,j+1} - v^\text{old}_{i,j}}{\Delta y_j} \right)^2 \right] \right. \]

\[ + \left( \frac{v^\text{old}(x_i^f, y^f_j) - v^\text{old}(x_i^f, y^f_j)}{\Delta y_j} \right) \]

\[ - 2 \left( \frac{u^\text{old}_{i+1,j} - u^\text{old}_{i,j}}{\Delta x_i} + \frac{v^\text{old}_{i,j+1} - v^\text{old}_{i,j}}{\Delta y_j} \right) \] \Delta x_i \Delta y_j

\[ + C_f T^3 I_{i,j} \left( \frac{u^\text{old}_{i+1,j} - u^\text{old}_{i,j}}{\Delta x_i} + \frac{v^\text{old}_{i,j+1} - v^\text{old}_{i,j}}{\Delta y_j} \right) \Delta x_i \Delta y_j \]

(29)

To interpolate velocities \( v^\text{old}(x_i^f, y^f_j) \), \( u^\text{old}(x_i^f, y^f_j) \), \( v^\text{old}(x_i^f, y^f_j) \) and \( u^\text{old}(x_i^f, y^f_j) \) a bilinear interpolation between four neighboring nodes is used for each one.

The coefficients that correspond to explicit approximation scheme of convective terms for temperature are:

\[ a_0^T = \Delta t \left( a_1^T + a_2^T + a_3^T + a_4^T \right) + \rho^\text{old}_{i,j} \Delta x_i \Delta y_j \]

\[ a_1^T = D_{i,j}^T, \ a_2^T = D_{i+1,j}^T, \ a_3^T = D_{i,j+1}^T, \ a_4^T = D_{i+1,j+1}^T \]

\[ T^\text{explicit}_{i,j} = - F_{i+1,j}^x \left[ \text{upwind}(T_{i-1,j-1}^{n-1}, T_{i-1,j}^{n-1}, u_{i-1,j}^{n-1}) \right] \]

\[ + (T_{i+1,j}^{n-1} - T_{i-1,j}^{n-1}) \psi_s \left( T_{i-1,j}^{n-1}, T_{i,j}^{n-1}, T_{i,j+1}^{n-1}, T_{i,j+2}^{n-1}, \Delta x_{i-1}, \Delta x_i, \Delta x_{i+1}, \Delta x_{i+2}, u_{i+1,j}^{n-1} \right) \]

\[ + F_{i,j}^x \left[ \text{upwind}(T_{i-1,j}^{n-1}, T_{i,j}^{n-1}, u_{i,j}^{n-1}) \right] \]

\[ + (T_{i,j}^{n-1} - T_{i-1,j}^{n-1}) \psi_s \left( T_{i-1,j}^{n-1}, T_{i,j}^{n-1}, T_{i,j+1}^{n-1}, T_{i,j+2}^{n-1}, \Delta x_{i-2}, \Delta x_{i-1}, \Delta x_i, \Delta x_{i+1}, u_{i+1,j}^{n-1} \right) \]

\[ - F_{i,j+1}^y \left[ \text{upwind}(T_{i,j}^{n-1}, T_{i,j+1}^{n-1}, u_{i,j+1}^{n-1}) \right] \]

\[ + (T_{i,j+1}^{n-1} - T_{i,j}^{n-1}) \psi_s \left( T_{i,j}^{n-1}, T_{i,j+1}^{n-1}, T_{i,j+2}^{n-1}, \Delta y_{j-1}, \Delta y_j, \Delta y_{j+1}, \Delta y_{j+2}, u_{i,j+1}^{n-1} \right) \]

\[ + F_{i,j}^y \left[ \text{upwind}(T_{i,j}^{n-1}, T_{i,j+1}^{n-1}, u_{i,j+1}^{n-1}) \right] \]

\[ + (T_{i,j}^{n-1} - T_{i,j-1}^{n-1}) \psi_s \left( T_{i,j-2}^{n-1}, T_{i,j-1}^{n-1}, T_{i,j}^{n-1}, T_{i,j+1}^{n-1}, \Delta y_{j-2}, \Delta y_{j-1}, \Delta y_j, \Delta y_{j+1}, u_{i,j-1}^{n-1} \right) \]

(30)
The sequence of calculation of numerical equations in GPU can be determined after detailed analysis of calculations in a loop along the y-axis. The numerical equations for \( u, v, p \) and \( T \) are (13), (14), (22) and (24), respectively. The equations for temperature (24) and pseudo velocities \( \hat{u} (15) \) and \( \hat{v} (16) \) depend on the values from the previous iteration and the previous time step that are constant variables for the iteration (loop 2). On the other hand the equations for \( u (13), v (14) \) and \( p (22) \) contain information from current loop 2, (see Fig. 2). The basic numerical equations can be represented as function of constant and calculated variables in loop along the y-axis as follow:

\[
T_{i,j} = f(\text{constant variables in loop 2}) \quad (31)
\]

\[
\hat{u}_{i,j} = f(\text{constant variables in loop 2}) \quad (32)
\]

\[
d_{i,j} = f(\text{constant variables in loop 2}) \quad (33)
\]

\[
\hat{v}_{i,j} = f(\text{constant variables in loop 2}) \quad (34)
\]

\[
d_{i,j} = f(\text{constant variables in loop 2}) \quad (35)
\]

\[
p_{i,j} = f(\hat{u}_{i,j}, d_{i,j}, \hat{u}_{i+1,j}, d_{i+1,j}, \hat{v}_{i,j}, d_{i,j}, \hat{v}_{i,j+1}, d_{i,j+1}, T_{i,j}, \text{constant variables in loop 2}) \quad (36)
\]

\[
u_{i,j} = f(\hat{u}_{i,j}, d_{i,j}, p_{i-1,j}, p_{i,j}, \text{constant variables in loop 2}) \quad (37)
\]

The variables \( T, \hat{u}, d^u, \hat{v}, d^v \) depend on the constant variables in loop 2 only, see equations (31) – (35), therefore, they are sequence independent in the loop along the y-axis. On the other hand, \( p, u \) and \( v \) are coupled and depends on calculated variables in loop 2. In CPU algorithm, calculated variables \( \hat{u}, d^u, \hat{v} \) and \( d^v \) are stored in arrays in RAM memory and used in pressure equation. After pressure calculation \( \hat{u}, d^u, \hat{v}, d^v \) and obtained pressure
are used to calculate $u$ and $v$, see Fig. 2. In GPU algorithm we can reduce write/read to/from global GPU memory using local memory. The variables $p_{i-1,j}, p_{i,j}, \hat{u}_{i,j}, d_{i,j}^u, \hat{v}_{i+1,j}, d_{i,j+1}^v$ and $v_{i,j}$ are temporarily stored in local memory while $T_{i,j}$ is temporarily stored in private memory. Fig. 5 shows detailed information about subdomain calculations. In upper part Fig. 5 shows subdomains with halo regions calculated from CU and direction of internal loop along the $y$-axes. In blue color are calculated variables that are copied in global memory and do not have relation to current calculations. In black color are variables that will be calculated in the next iterations along the $y$-axis. In red color are variables calculated in previous iterations ($j - 1$) along the $y$-axis. The red colored variables used in current iteration ($j$) are: $p_{i,j-1}, \hat{v}_{i,j}$ and $d_{i,j}^v$. In green color are variables calculated in current iteration along the $y$-axes ($j$): $p_{i,j}, u_{i,j}, \hat{u}_{i,j}, d_{i,j}^u, \hat{v}_{i,j}, \hat{v}_{i,j+1}$ and $d_{i,j+1}^v$. In the middle part are placed relation between x-coordinate and indexing of the arrays defined in global and local device memory. In lower part Fig. 6 shows control volumes for pressure and velocities. The detailed notations of variables together (lower left part) and separately (lower right part) that contains only calculated non-constant variables in numerical equation. One can find a pseudo code of the kernel that calculate loop 2 in Appendix A. The reuse of temporary variables stored in private and local memory decrease write/read to/from global memory, increase performance and decrease global memory requirements.
The organization of arrays in local memory is relatively simple. We will examine the variables $p_{i,j}$ and $p_{i,j-1}$. They were stored in array $\texttt{p.local}$ in the local memory. The size of the array is $\texttt{Nx.local} \times \texttt{Ny.p.local}$, where $\texttt{Nx.p.local}$ varies according to local memory size (here is fixed to 252), the number of rows $\texttt{Ny.p.local}$ is fixed to 2 and do not depend on local memory.
size. The organization of indexing of arrays is done using macros, see Fig. 6. \(\hat{u}, \hat{v}\) and \(d^v\) are other variables stored in arrays in local memory. The arrays sizes are \(N_x\) local \(\times N_y\) u\_pseudo\_local, \(N_x\) local \(\times N_y\) du\_local, \(N_x\) \(\times N_y\) v\_pseudo\_local and \(N_x\) local \(\times N_y\) dv\_local, respectively, where \(N_y\) u\_pseudo\_local = 1, \(N_y\) du\_local = 1, \(N_y\) v\_pseudo\_local = 2 and \(N_y\) dv\_local = 2.

\[
\text{#define p\_local(i,j) p\_local\_store[(i) - i0\_global + halo\_I_1 + ((j) % Ny\_p\_local) * Nx\_local]}
\]

Figure 6: OpenCL pseudo code of expression implemented as macros for indexing array \(p\_local\_store\) defined in local memory that store data for \(p_{i,j}\) and \(p_{i,j-1}\), where \(i0\_global\) (\(i0\_global\)) is global index of first element in the subdomain, \(halo\_I_1\) is the number of halo elements in direction \(i\)-1, \(Nx\_local\) is the number of elements along the x-axis, \(Ny\_p\_local\) is the number of elements along the y-axis.

3.3. GPU specifics

As far the GPU algorithm and organization is presented mainly from the algorithmic and numerical equations point of view. The GPU’s specifics are very important part of kernel development. In this and next subsection are presented GPU’s specifics that influence over kernel development and performance.

In the last few years, the performance of Graphics Processing Units (GPU’s) overcame significantly the performance of Central Processor Units (CPUs). The reason is that desktop CPUs have 4 cores, while the GPU’s have much more Compute Units (CU). As an example, AMD Radeon R9 280X have 32 CU. The CU are very similar to CPU core; therefore AMD Radeon R9 280X possess 8 times more “cores” than ordinary desktop CPU. GPU architecture is very similar to CPU one: GPU CU corresponds to CPU core; GPU private memory corresponds to CPU registers; GPU local memory corresponds to CPU L1 cache, and GPU global memory corresponds to computer RAM memory. On the other hand CPU core handle one thread, while CU can handle a lots of threads (work-items) simultaneously. One CU of AMD Radeon R9 280X can handle maximum 2560 work items. Unfortunately, the block of work-items executed together are 64 (a wavefront) \[24\]. Up to four work-items from the same

24
wavefront on the same stream core are pipelined to hide latency due to memory accesses and processing element operations. Therefore one CU maximum pipeline is 4x64=256 wave-items, see [24].

On the other hand, GPU’s are specific devices for parallel calculations. GPU’s possess smaller private (registers), local and global memory compared to CPUs. Therefore, the suitable algorithm should possess excellent parallel scalability and to be highly arithmetic intensive i.e. to do very intensive calculations over relatively small number of variables.

GPU devices require highly arithmetic intensive algorithms to reach good performance. Volkov present importance of registers usage and instruction-level parallelism (ILP) to reach better performance at lower occupancy [25]. This idea was adopted and implemented appropriately in presented GPU algorithm. On the other hand, a small expressions can be optimized very precisely, while numerical equations of CFD are very complex and large. Independent variables for 2D case are 4 \((p, T, u \text{ and } v)\) and requires up to 388 floating point operations per numerical equation. For each node, we have to copy from memory values from 15 to 20 control volumes. The number of all copied variables per node is approximately from 60 to 80. The number floating point operations per node is from 687 to 1229, see Table 1 and Table 2. Therefore, the arithmetic intensity is high: from \(687/80 \approx 9\) to \(1229/60 \approx 20\). Manual optimization of these big expressions is hard work, where many errors can occur. Furthermore, the changes in partial differential equations or numerical scheme will require corresponding changes in a code. Almost all optimization was left to a compiler to overcome difficulties related to manual optimization. The compiler organizes the copy of data from global to private memory and reuse of calculated numerical expressions in a code. The basic elements in numerical expressions were substituted using macros. The number of floating point operations was count after macros expansion in expressions, Table 1 and Table 2. This number of floating point operations was used to calculate GPU performance. The compiler can reduce number of floating point operations using common subexpression elimination (CSE), but analysis of assembler is out of the scope of this paper, and CSE
reduction was not taken into account. Presented idea in this paper is to use big expressions that gives possibility for ILP, use registers to store temporary variables and leave copy of data between memories and latency hiding organization to a compiler.

3.4. Tips and tricks

Execution of some operations on GPU’s cause great performance lost. To overcome this have to use alternative operation or code reorganization. The execution time of division and square root are slow operations. To speedup code density \( \rho = p/T \) and diffusion coefficient \( \Gamma = \sqrt{T} \) was stored in global memory. The logical operations could use with care in GPU code. Even a couple logical operations per kernel could decrease performance a couple of times. One can replace logical operators with equivalent multiplication and addition operations, taking into account specifics of data converting between boolean and integer variables. The boolean type, known in C++ as \texttt{bool}, can only represent one of two states, true or false. Boolean type true and false converted to an integer type, known in C++ as \texttt{int}, are 1 and 0, respectively. The conversion of integer type to boolean type is 0 to false and every value different from 0 to true; therefore -1, 1, -10, 10 are true. The conversion between boolean type and floating-point types is the same. Take into account conversion between data types logical AND can be replaced with multiplication, logical OR can be replaced with addition, see Table \( \ref{table:logical_operations} \). The power function also could decrease code performance. In presented algorithm only values to the power of 2 were used and were substituted with multiplication, see Table \( \ref{table:logical_operations} \). After all the slow operations as logical AND, logical OR and power function can be replaced with faster equivalent operations that contain the fastest operations as addition, subtraction, multiplication and type conversion.

Simplified test code could make the analysis of performance much easier and show some important tendencies. Testing of corresponding operations directly in CFD code could require additional changes and the results could be unclear. The simplified test code used by the author was the equation of temperature.
All variables were defined in private memory. Simplified temperature equation was calculated within a loop, where variables depend only on a counter and no variables were copied to/from global memory. The simplified test code shows performance lost unquestionably when use division, logical AND, logical OR and power function. Simplified test code shows excellent performance of equivalent operations in Table 3. The developer could check every operator performance when substitute it with multiplication or addition. This substitution will give the information about the performance of operator according the fastest operations as addition and multiplication. If the code accelerates more than two times, it is worth to try to find faster equivalent.

The dependence of the code performance from specific operators is expected, because the GPU’s development was boosted initially by the game industry, where for graphical processing are calculate relatively simple expressions. Furthermore, the variables are integer or floating point with single precision accuracy. On the other hand, scientific expressions could be big and complicated expressions that include different operators and double precision floating point operations. The GPU’s are using actively for a scientific computing only for a couple of years and hardware, drivers and development tools are in a process of rapid development. Nevertheless, short period the fastest supercomputers at the moment use GPU’s as coprocessors, [20].

Other specific in GPU development is related to if-then-else conditions. If-

| Operation   | C++ operation | Faster equivalent for GPU |
|-------------|---------------|--------------------------|
| Logical AND | \( x \&\& y \) | \( x \times y \)         |
| Logical OR  | \( x \mid\mid y \) | \( \text{bool}(x + y) \) |
| Logical NOT | \( !x \)      | \( 1 - \text{bool}(x) \) |
| Logical XOR | \( - \)       | \( \text{bool}(x - y) \) |

\( x \) to the power of 2 \( \text{pow}(x, 2) \) or \( \text{pown}(x, 2) \) \( x \times x \)

Table 3: Equivalent operations using multiplication and addition for faster calculations on GPU.

then-else conditions are widely used in CPU codes to reduce calculations that
speedup the code after all. A conditional of the form if-then-else generates branching (code serialization) in GPU codes and could slow down the calculations significantly. According AMD Accelerated Parallel Processing OpenCL Programming Guide, page 2-4, branching occurs when: "If work-items within a wavefront diverge, all paths are executed serially." This serialization could be a reason for the significant performance loss of GPU code, especially when conditions are nested, page 7-54: "When if blocks are nested, the results are twice as bad; in general, if blocks are nested $k$ levels deep, $2^k$ nested conditional structures are generated." For more information about branching see sections 2.1.3 Flow Control, 6.8.3 General Tips, 6.8.7 Optimizing Kernels for Southern Island GPU’s and 7.10.7 Optimizing Kernels for Evergreen and 69XX-Series GPU’s. The solution is to change the algorithm in a way to avoid if-then-else conditions. One of the possible approaches are Kronecker delta function or ternary operator (?:). Kronecker delta function reduces branching but increase number of operations. Density calculations in middle points using upwind scheme is a typical example of application of Kronecker delta function and ternary operator, Fig. 7. Substituting everything with Kronecker delta function significantly increase the number of floating points operations, when calculating control volumes on the walls. The control volumes on the walls require approximately two times more floating point operations than the control volumes in the fluid. The control volumes on the walls are a small part of control volumes in the computational domain. Therefore, an if-then-else condition was applied to separate control volumes that are on the wall and inside the fluid, see Fig. 8. Kronecker delta function or ternary operator was applied to the control volumes on the walls to avoid nested if-then-else conditions. In GPU code, the performance difference between Kronecker delta function and ternary operator is around 10% and varies according approximation scheme. After all the application of if-then-else conditions requires to take into account algorithm’s and code’s specifics and to do appropriate tests to understand influence of implementation of different operators over performance.
CPU code
if(0<u(i,j))
  rho_u(i,j)=rho(i-1,j);
else
  rho_u(i,j)=rho(i,j);

GPU code
Kronecker delta expansion
rho_u(i,j)=(0<u(i,j))*rho(i-1,j)
+(!(0<u(i,j)))*rho(i,j);

OR
ternary operator
rho_u(i,j)="(0<u(i,j))?rho(i-1,j):rho(i,j);"

Figure 7: Calculate density in middle points using upwind scheme: CPU code (left part) and GPU codes (right part).

if(control_volume_in_fluid(i,j))
{
  //Calculate numerical equations
  //without implementation of boundary conditions.
}
else if(control_volume_on_the_wall(i,j))
{
  //Calculate numerical equations
  //with implementation of boundary conditions.

  //The boundary conditions implementation
  //increase number of floating point operations
  //approximately two times.
}

Figure 8: GPU pseudo code: check place of calculated control volume.

4. Test case formulation

As a test case we use flow past a square particle(s) in a microchannel. The fluid model is described by the Navier-Stokes-Fourier equations (1) - (5). For gaseous microflow description, we use the model of a compressible, viscous hard sphere gas with diffusion coefficients determined by the first approximation of the Chapman-Enskog theory for low Knudsen numbers [27]. The Knudsen number ($Kn$), a nondimensional parameter, determines the degree of appropriateness of the continuum model. It is defined as the ratio of mean free path $\ell_0$ to the macroscopic length scale of the physical system $L$ ($Kn = \ell_0/L$). For the
calculated case, the Knudsen number is equal to $Kn = 0.001$ and the speed is equal to Mach number $M = 2.43$ at the channel inlet. For a hard-sphere gas, the viscosity coefficient $\mu$ and the heat conduction coefficient $\lambda$ (first approximations are sufficient for our considerations) read as follows:

\[
\mu = \mu_h \sqrt{T}, \quad \mu_h = \left(\frac{5}{16}\right)\rho_0 \ell_0 V_{th} \sqrt{\pi} \tag{39}
\]

\[
\lambda = \lambda_h \sqrt{T}, \quad \lambda_h = \left(\frac{15}{32}\right)c_p \rho_0 \ell_0 V_{th} \sqrt{\pi} \tag{40}
\]

The Prandtl number is given by $Pr = 2/3$, $\gamma = c_p/c_v = 5/3$. The dimensionless system of equations (1) - (5) is scaled by the following reference quantities, as given in [27]: molecular thermal velocity $V_0 = V_{th} = \sqrt{2RT_0}$ for velocity, for length - square size $a$ (Fig. 10), for time - $t_0 = a/V_0$, the reference pressure ($p_0$) is pressure at the inflow of the channel, the reference temperature ($T_0$) is equal to the channel walls, reference density ($\rho_0$) is calculated using equation of state (5). The corresponding non-dimensional parameters in the equation system (1) - (5) are computed by using the following formulas:

\[
A = 0.5, \quad B = \frac{5\sqrt{\pi}}{16} Kn, \quad \Gamma = \Gamma^\lambda = \sqrt{T}
\]

\[
C^{T_1} = Kn \sqrt{\frac{\pi}{1024}}, \quad C^{T_2} = \frac{\sqrt{\pi}}{4} Kn, \quad C^{T_3} = \frac{2}{5} \tag{41}
\]
Fig. 9 shows the test case geometry. The channel length is $L_{ch} = 201.6$, the channel inlet is $L_a = 5.5$. The channel height ($H_{ch}$) varies from 10 to 200 because the influence of iterations in kernel along the y-axis was investigated over the performance. The uniform Cartesian grid with special steps $\Delta x = \Delta y = \Delta = 0.05$ was used. The problem is considered in a local Cartesian coordinate system, which is moving with the particle. Thus for an observer moving along with the particle the problem is transformed to a consideration of a gas flow past a stationary square confined in a microchannel with moving walls. Velocity-slip and temperature-jump boundary conditions [28] are imposed on the walls of the channel and the square. The velocity-slip BC is given as:

$$v_s - v_w = \zeta \frac{\partial v}{\partial n} \bigg|_s,$$

where $v_s$ is velocity of the gas at the solid wall surface, $v_w$ is velocity of the wall, $\zeta = 1.1466.K_n_{local} = 1.1466.Kn/\rho_{local}$, $Kn_{local}$ is the local Knudsen number, $\rho_{local}$ is the local density, $\frac{\partial v}{\partial n} \bigg|_s$ is the derivative of velocity normal to the wall surface. The temperature-jump boundary condition is:

$$T_s - T_w = \tau \frac{\partial T}{\partial n} \bigg|_s,$$

where $T_s$ is temperature of the gas at the wall surface, $T_w$ is temperature of the wall, $\tau = 2.1904.Kn_{local} = 2.1904.Kn/\rho_{local}$, $\frac{\partial T}{\partial n} \bigg|_s$ is the derivative of temperature normal to the wall surface.

5. Speedup analysis

The GPU code speedup was obtained with comparison with serial CPU code. Both codes use double precision floating point operations. The GPU and CPU codes demonstrate agreement within double precision floating point accuracy that is 15 significant digits or $10^{-15}$. The GPU kernels are written in OpenCL.
that make it portable without modifications to AMD and NVIDIA GPU’s. The CPU code is written in C++. GPU code performance was obtained on AMD Radeon R9 280X and NVIDIA Tesla M2090 while CPU code performance was tested on Intel Core i5-4690 and Intel Core i7-920. AMD Radeon R9 280X is a gaming GPU with following characteristics: release date 10.2013, peak double precision floating point performance 1024[GF/s], memory size 3072[MB], number of Compute Units 32, see [29]. NVIDIA Tesla M2090 is a server GPU for scientific calculations with following characteristics: release date 03.2011, peak double precision floating point performance 665[GF/s], memory size 6[GB], number of Compute Units 16, see [30]. Intel Core i7-920 characteristics are as follow: launch date Q4’08, Instruction Set Extensions is SSE4.2, Processor Base Frequency is 2.66[GHz], Max Turbo Frequency 2.93[GHz], Number of Cores 4, Number of Threads 8, see [31]. CPU Intel Core i5-4690 is the next generation core architecture according Intel Core i7-920. Intel Core i5-4690 characteristics are as follow: launch date Q2’14, Instruction Set Extensions is SSE4.1/4.2, AVX 2.0, Processor Base Frequency is 3.5[GHz], Max Turbo Frequency 3.9[GHz], Number of Cores 4, Number of Threads 4, see [32].

Speedup tests were obtained on tree configurations:

- CPU Core i5-4690, motherboard Gigabyte Z97-HD3, dual channel memory 32[GB] at 1333[MHz], video card is Sapphire Tri-X R9 280X (AMD Radeon R9 280X), the operating system is Debian GNU/Linux 7.7 (wheezy) x64bit, C++ compiler g++ version is 4.6.3-14, video card driver version is AMD Catalyst 14.501.1003 and AMD OpenCL SDK version is 2.9-1. This configuration is part of cluster of Institute of Mechanics at the Bulgarian Academy of Sciences (IMech-BAS).

- CPU Core i7-920, motherboard ASUS P6TD DELUXE, triple channel memory 18[GB] at 1333[MHz], video card is GeForce GTX 260, the operating system is Debian GNU/Linux 7.7 (wheezy) x64bit and C++ compiler g++ version is 4.6.3-14.

- Two CPU Intel Xeon E5649, memory 96[GB], eight GPU NVIDIA Tesla
M2090, the operating system is Scientific Linux release 6.6 (Carbon) x64bit, C++ compiler g++ version is 4.4.7-11, video card driver version is 319.37 and NVIDIA CUDA SDK version is 5-5. This configuration is part of HPCG cluster [33]. The HPCG cluster is located at Institute of Information and Communication Technologies at the Bulgarian Academy of Sciences (IICT-BAS).

A multiple GPU system needs appropriate cooling. One of the easiest ways to build multiple GPU systems with good cooling is to use PCI-e 1x to 16x riser cable. PCI-e 1x to 16x riser cables are about 20cm long and connect GPU with motherboard PCI-e slot. Mounting GPU on a distance of a motherboard improve cooling of GPU and make possible to increase the number of GPU’s per configuration significantly. A maximum number of GPU’s depends on a number of PCI Express 16x and 1x slots per motherboard. The not specialized motherboard has from 3 to 8 PCI Express slots. The main disadvantage of PCI-e 1x to 16x riser cables is slow down the bandwidth between GPU and motherboard. The performance of presented GPU algorithm does not depend on the bandwidth between GPU and motherboard. The program copy data to the device (GPU) global memory at the beginning of calculations. During the calculations program copy from device memory to host memory only maximum residuals of loop 2. That is a small quantity of information and do not need high bandwidth. A test shows that presented GPU code obtained the same performance on a system when GPU AMD Radeon R9 280X was connected to the motherboard with PCI-e 1x to 16x riser cable as the case when GPU AMD Radeon R9 280X was connected to the motherboard directly. After all PCI-e 1x to 16x riser cable improve cooling of configuration and increase the number of GPU’s per motherboard without performance loss of presented algorithm.

The memory requirements are important for GPU codes because GPU’s memory is fixed and less than CPU memory. The presented GPU algorithm does an internal loop along the y-axis and uses pseudo velocities, pressure coefficients and other temporary variables only in local memory that decrease arrays
in global memory. Finally, the GPU code requires two times less memory than CPU code. GPU code defines 5.9 million control volumes per 1 GB of global GPU memory for explicit and implicit schemes.

The computational domain was decomposition to 32 subdomains, 16 subdomains along the x-axis and 2 in along the y-axis. The number of CU (Compute Units) of GPU AMD Radeon R6 280X is 32 that mean subdomain per CU. The number of CU of NVIDIA Tesla M2090 is 16 that mean two subdomains per CU. A GPU reach maximum performance when the number of subdomains is multiple to the number of CU.

Here was investigated influence of the number of rows per subdomain over performance because this is an important idea of proposed approach. To this aim, the speedup was obtained on four cases with different channel heights ($H_{ch} = 10, 20, 100$ and $200$) that corresponds to the number of square particles along the x-axis: 1, 2, 10 and 20, respectively. Fig. 10 shows fields of pressure, temperature, horizontal and vertical velocities for test case with $H_{ch} = 20$.
and two square particles. The test cases meshes were 4032x200, 4032x400, 4032x2000 and 4032x4000 points that corresponds to \( H_{ch} = 10, 20, 100 \) and 200, respectively. Each work group has 256 work-items (threads) that calculate 252 nodes along the x-axis and four work-items that calculate halo region of the subdomain. Therefore, meshes per subdomain that correspond to test cases are 256x100, 256x200, 256x1000 and 256x2000. \( N^{subdomain}_y \) is number of control volumes along the y-axis per subdomain that for calculated test cases \( N^{subdomain}_y \) is equal to 100, 200, 1000 and 2000. The speedup results were normalized according execution time of serial code on CPU for the corresponding case.

Fig. 11, Fig. 12, Fig. 13 and Fig. 14 shows the influence of the number of rows of subdomain over the performance. AMD Radeon R9 280X increase speedup with increase of \( N^{subdomain}_y \). The obtained performances for \( N^{subdomain}_y = 1000 \) and 2000 are close; therefore we consider that \( N^{subdomain}_y = 1000 \) is sufficient to reach maximum performance on AMD Radeon R9 280X. The performance of NVIDIA Tesla M2090 does not depend on \( N^{subdomain}_y \) and we consider that \( N^{subdomain}_y = 100 \) is sufficient to reach maximum performance of this device.

The performance tests show that AMD Radeon R9 280X is significantly faster than NVIDIA Tesla M2090, CPU Intel Core i7-920, and CPU Intel Core i5-4690. The GPU code executed on AMD Radeon R9 280X is faster compared to CPU serial code executed on Intel Core i7-920 from 150x to 184x times. Also, it is faster compared to CPU Intel Core i5-4690 from 81x to 102x times.
Figure 12: The speedup was obtained by comparison two GPU’s: GPU AMD Radeon R9 280X and GPU NVIDIA Tesla M2090 with serial code executed on two CPUs: CPU Intel Core i7-920 (left part) and CPU Intel Core i5-4690 (right part). In this test case, explicit upwind 1-st order scheme with Van-Leer limiter approximates convective terms.

Figure 13: The speedup was obtained by comparison two GPU’s: GPU AMD Radeon R9 280X and GPU NVIDIA Tesla M2090 with serial code executed on two CPUs: CPU Intel Core i7-920 (left part) and CPU Intel Core i5-4690 (right part). In this test case, implicit TVD second-order scheme with Van-Leer limiter approximates convective terms.

Figure 14: The speedup was obtained by comparison two GPU’s: GPU AMD Radeon R9 280X and GPU NVIDIA Tesla M2090 with serial code executed on two CPUs: CPU Intel Core i7-920 (left part) and CPU Intel Core i5-4690 (right part). In this test case, implicit upwind 1-st order scheme with Van-Leer limiter approximates convective terms.

see Fig. 11, Fig. 12, Fig. 13 and Fig. 14. NVIDIA Tesla M2090 speedup the GPU code compared to serial CPU code executed on Intel Core i7-920 from 9x to 20x times. Also, it is faster compared to CPU Intel Core i5-4690 from
5x to 11x times, see Fig. 11, Fig. 12, Fig. 13 and Fig. 14. The performance of CPU core of Intel Core i5-4690 overcome approximately two times previous generation CPU core of Intel Core i7-920. AMD Radeon R9 280X is the fastest device: it is approximately one order faster than NVIDIA Tesla M2090 and approximately two orders faster than Intel’s CPUs Core i5-4690 and Core i7-920. NVIDIA Tesla M2090 is approximately an order faster than Intel’s CPUs Core i5-4690 and Core i7-920.

6. Conclusions

GPU algorithm SIMPLE-TS calculates Navier-Stokes-Fourier system of partial differential equations describing unsteady, viscous, compressible, heat-conductive gas flows with double precision accuracy. A test case was unsteady flow past a square particles in a microchannel at speed $M = 2.43$ and rarefaction $Kn = 0.001$.

The appropriate use of device memories is important when porting CPU code to GPU. As the private memory is the fastest GPU memory, we use it to keep calculated variables. In local memory were stored temporary calculated arrays that reduce the use of global memory and increase code performance. The equations were put together using macros. As a result, preprocessor composes big expressions that increase Instruction Level Parallelism (ILP). Almost all optimization of the code was left to the compiler. The compile eliminates common subexpression (CSE), organize data copy from global to private device memory and data reuse. The automatic optimization by a compiler improve code maintenance: simplifies code writing and further modifications. The proposed approach demonstrates excellent performance on AMD gaming GPU AMD Radeon R9 280X that overcome one order server NVIDIA Tesla M2090 and two orders serial C++ code run on CPU Intel Core i5-4690 and Intel Core i7-920. After all GPU code obtains excellent speedup on AMD GPU and looks more suitable to AMD GPU architecture than NVIDIA GPU architecture.
An important performance tests would be on AMD FirePro W9100 and V100 GPU Accelerator (Mezzanine). AMD FirePro W9100 double precision compute performance is 2.62 TFLOPS that is 2.62 times more than used here AMD Radeon R9 280X. V100 GPU Accelerator (Mezzanine) double precision compute performance is 7.45 TFLOPS that is 11.2 times more than used here NVIDIA Tesla M2090 and could contain important hardware changes.

An important demonstration would be the calculation of 3D fluid flow in complex geometry that would establish the performance in a realistic engineering settings.

Acknowledgments

We would like to acknowledge the financial support provided by the Bulgarian NSF under Grant DN-02/7-2016. This research has received funding from the European Union Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie MIGRATE grant agreement No. 643095. The calculations on GPU AMD Radeon R9 280X and CPU Intel Core i5-4690 in this work used the cluster of Institute of Mechanics - BAS. The calculations on GPU NVIDIA Tesla M2090 in this work used the EGI Infrastructure and were co-funded by the EGI-Engage project (Horizon 2020) under Grant number 654142.
Appendix A.

In this appendix is presented pseudo code of loop along the y-axis. That is the main part of loop 2 of GPU algorithm SIMPLE-TS (see Fig. 1 and Fig. 2). For brevity, the calculations are presented as function of constant and calculated variables in loop along the y-axis that correspond to the equations (31) - (38) and Fig. 5 notations.

//Loop along the y-axis
for(j_global = j_begin; j_global < j_end; j_global++)
{
    //Calculate density in middle points rho_u, numerical equation (12)
    //Set row index for the next expression for calculation
    j = j_global + 2;
    if(control_volume_in_fluid(i,j))
    {
        //The calculated control volume is not next to body surface, where
        //boundary conditions have to be applied. Therefore, calculate numerical
        //equation without implemented boundary conditions.
        //Write the result in local memory.
        rho_u(i,j) = f(constant variables in loop 2);
    }
    else if(control_volume_on_the_wall(i,j))
    {
        //The calculated control volume is next to the solid surface, therefore,
        //in numerical equations are implemented boundary conditions.
        //All checks using in implementation of boundary conditions in common
        //case increase number of floating point operations approximately twice.
        //Write the result in local memory.
        rho_u(i,j) = f(constant variables in loop 2);
    }
}
//Calculate density in middle points rho_v, numerical equation (13)
//Set row index for the next expression for calculation
j = j_global + 3;
if(control_volume_in_fluid(i,j))
{
  //The calculated control volume is not next to body surface, where
  //boundary conditions have to be applied. Therefore, calculate numerical
  //equation without implemented boundary conditions.
  //Write the result in local memory.
  rho_v(i,j) = f(constant variables in loop 2);
}
else if(control_volume_on_the_wall(i,j))
{
  //The calculated control volume is next to the solid surface, therefore,
  //in numerical equations are implemented boundary conditions.
  //All checks using in implementation of boundary conditions in common
  //case increase number of floating point operations approximately twice.
  //approximately twice.
  //Write the result in local memory.
  rho_v(i,j) = f(constant variables in loop 2);
}
//Wait to finish calculations before continue.
barrier(CLK_LOCAL_MEM_FENCE);

//Calculate temperature, numerical equation (27)
//Set row index for the next expression for calculation
j = j_global;
if(control_volume_in_fluid(i,j))
{
  //The calculated control volume is not next to body surface, where
/in numerical equations are implemented boundary conditions.
//All checks using in implementation of boundary conditions in common
//case increase number of floating point operations approximately twice.
//approximately twice.
//Write the result in local memory.
u_pseudo(i,j) = f(constant variables in loop 2);
du(i,j) = f(constant variables in loop 2);
}

//Calculate vertical pseudo velocity, numerical equation (19) and dv
//Set row index for the next expression for calculation
j = j_global + 1;
if(control_volume_in_fluid(i,j))
{
    //The calculated control volume is not next to body surface, where
    //boundary conditions have to be applied. Therefore, calculate numerical
    //equation without implemented boundary conditions.
    //Write the result in local memory.
v_pseudo(i,j) = f(constant variables in loop 2);
dv(i,j) = f(constant variables in loop 2);
}
else if(control_volume_on_the_wall(i,j))
{
    //The calculated control volume is next to the solid surface, therefore,
    //in numerical equations are implemented boundary conditions.
    //All checks using in implementation of boundary conditions in common
    //case increase number of floating point operations approximately twice.
    //approximately twice.
    //Write the result in local memory.
v_pseudo(i,j) = f(constant variables in loop 2);
dv(i,j) = f(constant variables in loop 2);
} //Wait to finish calculations before continue.
barrier(CLK_LOCAL_MEM_FENCE);

//Calculate pressure, numerical equation (25)
//Set row index for the next expression for calculation
j = j_global;
if(control_volume_to_calculate == true)
{
    //Write result in local and in global memory.
    p(i,j) = f(u_pseudo(i,j), du(i,j), u_pseudo(i+1,j), du(i+1,j),
        v_pseudo(i,j), dv(i,j), v_pseudo(i,j+1), dv(i,j+1),
        T(i,j), constant variables in loop 2);
}
//Wait to finish calculations before continue.
barrier(CLK_LOCAL_MEM_FENCE);

//Calculate horizontal velocity, numerical equation (16)
//Set row index for the next expression for calculation
j = j_global;
if(control_volume_to_calculate == true)
{
    //Write result in global memory.
    u(i,j) = f(u_pseudo(i,j), du(i,j), p(i-1,j), p(i,j),
        constant variables in loop 2);
}

//Calculate vertical velocity, numerical equation (17)
//Set row index for the next expression for calculation
j = j_global;
if(control_volume_to_calculate == true)
{
    //Write result in global memory.
    v(i,j) = f(v_pseudo(i,j), dv(i,j), p(i,j-1), p(i,j),
            constant variables in loop 2);
}
//Wait to finish calculations befor continue.
barrier(CLK_LOCAL_MEM_FENCE);
}
References

[1] H. K. Versteeg, W. Malalasekra, An Introduction to Computational Fluid Dynamics: The Finite Volume Method, 2nd Edition, Prentice Hall, Pearson, 2007.

[2] J. Nickolls, W. Dally, The GPU Computing Era, Micro, IEEE 30 (2) (2010) 56–69. doi:10.1109/MM.2010.41

[3] A. R. Brodtkorb, T. R. Hagen, M. L. Sætra, Graphics processing unit (GPU) programming strategies and trends in {GPU} computing, Journal of Parallel and Distributed Computing 73 (1) (2013) 4–13, metaheuristics on {GPUs}. doi:10.1016/j.jpdc.2012.04.003

[4] E. Elsen, P. LeGresley, E. Darve, Large calculation of the flow over a hypersonic vehicle using a GPU, Journal of Computational Physics 227 (24) (2008) 10148–10161. doi:10.1016/j.jcp.2008.08.023

[5] P. Zaspel, M. Griebel, Solving incompressible two-phase flows on multi-GPU clusters, Computers & Fluids (9) (2012) 1–9. doi:10.1016/j.compfluid.2012.01.021

[6] J. Cohen, M. J. Molemaker, A fast double precision CFD code using CUDA, Parallel Computational Fluid Dynamics: Recent Advances and Future Directions (2009) 414–429.
[7] F. Salvadore, M. Bernardini, M. Botti, GPU accelerated flow solver for direct numerical simulation of turbulent flows, Journal of Computational Physics 235 (0) (2013) 129–142. doi:10.1016/j.jcp.2012.10.012
URL http://www.sciencedirect.com/science/article/pii/S00219991112006018

[8] S. Liang, W. Liu, L. Yuan, Solving seven-equation model for compressible two-phase flow using multiple GPUs, Computers & Fluids 99 (2014) 156–171. doi:10.1016/j.compfluid.2014.04.021
URL http://www.sciencedirect.com/science/article/pii/S0045793014001595

[9] B. van Leer, Towards the ultimate conservative difference scheme. II. Monotonicity and conservation combined in a second-order scheme, Journal of Computational Physics 14 (4) (1974) 361–370. doi:10.1016/0021-9991(74)90019-9
URL http://www.sciencedirect.com/science/article/pii/0021999174900199

[10] The open standard for parallel programming of heterogeneous systems - OpenCL (Open Computing Language), last accessed 29.12.2015.
URL www.khronos.org/opencl/

[11] Complete list of companies and their conformant products that support OpenCL, last accessed 29.12.2015.
URL www.khronos.org/conformance/adopters/conformant-products#opencl

[12] A. Munshi, B. Gaster, T. G. Mattson, D. Ginsburg, OpenCL Programming Guide, Pearson Education, 2011.

[13] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, D. Schaa, Heterogeneous Computing with OpenCL, Second Edition: Revised OpenCL 1.2 Edition, Morgan Kaufmann, 2012.
[14] CUDA zone last accessed 29.12.2015.
URL developer.nvidia.com/cuda-zone

[15] D. B. Kirk, W. H. Wen-mei, Programming Massively Parallel Processors: A Hands-on Approach, Morgan Kaufmann, 2010.

[16] J. Sanders, E. Kandrot, CUDA by Example: An Introduction to General-Purpose GPU Programming, Addison-Wesley Professional, 2010.

[17] CUDA GPUs last accessed 29.12.2015.
URL developer.nvidia.com/cuda-gpus

[18] U. M. Ascher, S. J. Ruuth, Brian, B. T. R. Wetton, Implicit-Explicit Methods For Time-Dependent PDEs, SIAM J. Numer. Anal 32 (1997) 797–823.

[19] K. S. Shterev, S. K. Stefanov, Pressure based finite volume method for calculation of compressible viscous gas flows, Journal of Computational Physics 229 (2) (2010) 461–480. doi:10.1016/j.jcp.2009.09.042
URL http://dx.doi.org/10.1016/j.jcp.2009.09.042

[20] S. V. Patankar, D. B. Spalding, A Calculation Procedure for Heat, Mass and Momentum Transfer in Three-dimensional Parabolic Flows, Int. J. Heat Mass Transfer 15 (1972) 1787.

[21] R. I. Issa, Solution of the implicitly discretised fluid flow equations by operator-splitting, Journal of Computational Physics 62 (1986) 40–65.

[22] E. A. K. Shterev, S. Stefanov, GPU calculations of unsteady viscous compressible and heat conductive gas flow at supersonic speed, in: Large-Scale Scientific Computing, Large-ScaleScientific Computations, Springer-Verlag Berlin Heidelberg, 2013.

[23] T. Brandvik, G. Pullan, Acceleration of a 3D Euler Solver Using Commodity Graphics Hardware, Aerospace Sciences Meetings, American Institute of Aeronautics and Astronautics, 2008, doi:10.2514/6.2008-607. doi:10.2514/6.2008-607
[24] Advanced Micro Devices, Inc., *AMD Accelerated Parallel Processing OpenCL Programming Guide (rev2.7)* Tech. rep. (2013). URL developer.amd.com/appsdk

[25] V. Volkov, *Better Performance at Lower Occupancy*, GTC 2010, 2010. URL http://www.cs.berkeley.edu/textasciitildevolkov/

[26] Top 500 supercomputer sites last accessed 29.12.2015. URL www.top500.org

[27] S. Stefanov, V. Roussinov, C. Cercignani, Rayleigh-Bénard flow of a rarefied gas and its attractors. I. Convection regime, *Physics of Fluids* 14 (7) (2002) 2255–2269. doi:10.1063/1.1483837 URL http://link.aip.org/link/?PHF/14/2255/1

[28] C. Cercignani, Theory and Application of the Boltzmann Equation, Scottish Academic Press, 1975.

[29] List of AMD graphics processing units last accessed 29.12.2015. URL https://en.wikipedia.org/wiki/List_of_AMD_graphics_processing_units

[30] TESLA M2090 Product brief last accessed 29.12.2015. URL http://www.nvidia.com/docs/IO/105880/DS_Tesla-M2090_LR.pdf

[31] Intel Core i7-920 Processor brief last accessed 29.12.2015. URL http://ark.intel.com/products/37147/Intel-Core-i7-920-Processor-8M-Cache-2_66-GHz-4_80-GTs-Intel-QPI

[32] Intel Core i5-4690 Processor brief last accessed 29.12.2015. URL http://ark.intel.com/products/80810/Intel-Core-i5-4690-Processor-6M-Cache-up-to-3_90-GHz
[33] E. Atanassov, T. Gurov, A. Karaivanova, S. Ivanovska, M. Durchova, D. Georgiev, D. Dimitrov, Tuning for scalability on hybrid HPC cluster, Mathematics in Industry, Cambridge Scholars Publishing (2014) 64–77.