Computer Physics Communications 200 (2016) 50–56

Contents lists available at ScienceDirect

Computer Physics Communications

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

Implementation of the Vanka-type multigrid solver for the finite element approximation of the Navier–Stokes equations on GPU

Petr Bauer [a], Vladimír Klement [b], Tomáš Oberhuber [b,*], Vítězslav Žabka [b]

[a] Institute of Thermomechanics, Academy of Sciences of the Czech Republic, Dolejškova 5, Praha 8, 182 00, Czech Republic

[b] Department of Mathematics, Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University in Prague, Trojanova 13, Praha 2, 120 00, Czech Republic

ARTICLE INFO

Article history:
Received 21 February 2014
Received in revised form
19 August 2015
Accepted 29 October 2015
Available online 11 November 2015

Keywords:
Navier–Stokes equations
Mixed finite elements
Multigrid
Vanka-type smoothers
Gauss–Seidel
Red–black coloring
Parallelization
GPU

ABSTRACT

We present a complete GPU implementation of a geometric multigrid solver for the numerical solution of the Navier–Stokes equations for incompressible flow. The approximate solution is constructed on a two-dimensional unstructured triangular mesh. The problem is discretized by means of the mixed finite element method with semi-implicit timestepping. The linear saddle-point problem arising from the scheme is solved by the geometric multigrid method with a Vanka-type smoother. The parallel solver is based on the red–black coloring of the mesh triangles. We achieved a speed-up of 11 compared to a parallel (4 threads) code based on OpenMP and 19 compared to a sequential code.

1. Introduction

Graphical Processing Units (GPUs) are accelerators originally developed for visualization of 3D scenes. They are optimized for parallel processing of millions of polygons and large data sets representing textures. When reading large blocks of data, a GPU can be several times faster (the memory bandwidth can be up to 288 GB/s on Nvidia Tesla K40) compared to a CPU (having a bandwidth of 68 GB/s for systems equipped with PC3-17000 DDR3 modules and quad-channel architecture).[1] Concerning the computational power, the peak performance of a single GPU can be as high as 1.5 TFLOPs which is comparable with a small cluster of personal computers (the peak performance of an Intel Xeon CPU with ten cores can be estimated to 140 GFLOPs). Both, the large memory bandwidth and the great computational power, make the GPUs valuable also for the scientific computing [1] including computational fluid dynamics. Porting numerical algorithms to GPUs is by no means trivial and some numerical solvers must be rewritten from scratch. In the following paragraphs, we summarize some results obtained during the last years by the community. All speed-ups were measured compared to sequential CPU computations.

The first attempts to solve the Navier–Stokes equations on the GPUs date back to 2005. In [2], the authors implemented the SMAC method on rectangular grids. They approximate the velocity explicitly while the pressure implicitly. The linear system for pressure is solved by the Jacobi method. They achieved an overall speed-up more than 20 on GeForce FX 5900 and GeForce 6800 Ultra compared to Pentium IV running at 2 GHz. In [3], the authors present a GPU implementation of a solver for the compressible Euler equations for hypersonic flow on complex domains. They applied a semi-implicit scheme with a finite difference discretization in space. The resulting linear system is solved by the multigrid method. It allowed a speed-up 15–40 using Core 2 Duo E6600 CPU at 2.4 GHz and GeForce 8800 GTX GPU. Both models were restricted to two dimensions.

The results of 3D simulations on a structured grid of quadrilateral elements were presented in [4] with a speed-up 29 and 16 in 2D and 3D respectively (on Core 2 Duo at 2.33 GHz and GeForce 8800 GTX). 3D computations of inviscid compressible flow using

* Corresponding author.
E-mail address: tomas.oberhuber@fjfi.cvut.cz (T. Oberhuber).
URL: http://geraldine.fjfi.cvut.cz/~oberhuber (T. Oberhuber).

[1] Both values are, however, theoretical peak performance throughputs and they are difficult to achieve in real applications.

http://dx.doi.org/10.1016/j.cpc.2015.10.021
0010-4655/

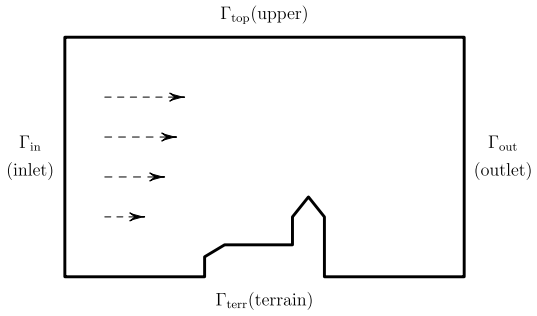


Fig. 1. Example of computational domain and setup of the boundary conditions.

unstructured grids were studied in [5]. The cell-centered finite volume method with the explicit Runge–Kutta timestepping was implemented and a speed-up 33 was achieved (on Intel Core 2 Q9450 and Nvidia Tesla card). The multigrid method for the Full Approximation Scheme in 2D was implemented on GPU in [6] with a speed-up 10. A computation on a multi-GPU system was presented in [7]. On four GPUs, a speed-up was 100 (on AMD Opteron 2.4 GHz and four Nvidia Tesla S870 cards). A two-phase flow solver for the Navier–Stokes equations was partially ported to GPU in [8]. On a GPU cluster with eight Nvidia Tesla S1070 GPUs made of two workstations equipped with Intel Core i7-920 at 2.66 GHz, the authors achieved a speed-up up to 115.

In this paper, we solve a simplified model of air flow in a 2D urban canopy based on the incompressible Navier–Stokes equations. The space discretization is done by the mixed finite element method. We use quasi-regular meshes, however, the code can handle unstructured meshes as well. The scheme is semi-implicit in time and the linear saddle-point problem is solved by the multigrid method with a Vanka-type smoother.

2. Mathematical model

In environmental applications featuring low velocities, the air flow is governed by the system of viscous incompressible Navier–Stokes equations of the form

$$\frac{\partial \vec{u}(t,\vec{x})}{\partial t} + \vec{u}(t,\vec{x})\cdot\nabla\vec{u}(t,\vec{x}) - \nu\Delta\vec{u}(t,\vec{x}) + \nabla p(t,\vec{x}) = \vec{0}, \tag{1a}$$

$$\nabla\cdot\vec{u}(t,\vec{x}) = 0, \tag{1b}$$

where t stands for time, $\vec{x} = (x, y)$, $\vec{u}$ stands for the velocity, p stands for the pressure, and ν denotes the kinematic viscosity of air. We solve this system on the domain Ω similar to the one depicted on Fig. 1 with the boundary parts denoted by Γ_{terr}, Γ_{in}, Γ_{out}, Γ_{top}. We use the following set of initial and boundary conditions

$$\vec{u}(0,\vec{x}) = \vec{u}_0(\vec{x}) \quad \text{in } \Omega, \tag{2a}$$

$$\vec{u} = \vec{0} \quad \text{on } \Gamma_{terr}, \tag{2b}$$

$$u_x = u_{in},\ u_y = 0 \quad \text{on } \Gamma_{in}, \tag{2c}$$

$$-p\vec{n} + \nu(\nabla\vec{u})\cdot\vec{n} = \vec{0} \quad \text{on } \Gamma_{out}, \tag{2d}$$

$$(\nu(\nabla\vec{u})\cdot\vec{n})_x = 0,\ u_y = 0 \quad \text{on } \Gamma_{top}. \tag{2e}$$

The vertical inlet velocity profile is given by the power law

$$u_{in}(y) = \bar{u}_{ref}(y/y_{ref})^{\alpha} \tag{3}$$

for a given average wind speed $\bar{u}_{ref}$ at some reference height y_{ref} and for the profile exponent α. The semi-implicit Oseen scheme is used for the time discretization of (1a)

$$\frac{\vec{u}^n - \vec{u}^{n-1}}{\tau} + \vec{u}^{n-1}\cdot\nabla\vec{u}^n - \nu\Delta\vec{u}^n + \nabla p^n = \vec{0}, \tag{4}$$

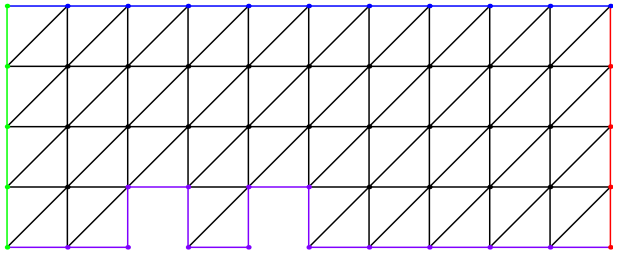


Fig. 2. Initial coarse mesh $\mathcal{T}_0$.

where $\tau > 0$ is the timestep, $\vec{u}^n(\vec{x}) \equiv \vec{u}(n\tau,\vec{x})$ and $p^n(\vec{x}) \equiv p(n\tau,\vec{x})$. The system (4) is discretized in space by the nonconforming P_1/P_0 Crouzeix–Raviart finite elements, and the convective term is stabilized through the upwinding [9]. At each time level n, we solve a linear system of the form

$$\begin{pmatrix} \tilde{\mathbf{A}}_x(\tilde{\mathbf{u}}^{n-1}) & \mathbf{0} & -\tilde{\mathbf{B}}_x^T \\ \mathbf{0} & \tilde{\mathbf{A}}_y(\tilde{\mathbf{u}}^{n-1}) & -\tilde{\mathbf{B}}_y^T \\ \tilde{\mathbf{B}}_x & \tilde{\mathbf{B}}_y & \mathbf{0} \end{pmatrix} \begin{pmatrix} \tilde{\mathbf{u}}_x^n \\ \tilde{\mathbf{u}}_y^n \\ \tilde{\mathbf{p}}^n \end{pmatrix} = \begin{pmatrix} \tilde{\mathbf{f}}_x(\tilde{\mathbf{u}}^{n-1}) \\ \tilde{\mathbf{f}}_y(\tilde{\mathbf{u}}^{n-1}) \\ \tilde{\mathbf{g}} \end{pmatrix}, \tag{5}$$

where $\tilde{\mathbf{A}}_x$, $\tilde{\mathbf{A}}_y$ contain the convective and the viscous terms, and $\tilde{\mathbf{B}}_x$, $\tilde{\mathbf{B}}_y$ contain the divergence term.[2]

3. Geometric multigrid method

Multigrid methods proved to provide some of the most robust and efficient solvers for large problems in computational fluid dynamics [10,11].

The geometric multigrid method for the problem (4) works with a sequence of meshes with different mesh sizes, each producing the respective discretization. Our implementation uses a hierarchy of uniformly refined meshes $\mathcal{T}_l$, $l = 0, \ldots, L$. The initial coarse mesh $\mathcal{T}_0$ is shown in Fig. 2.

The linear systems (5) for coarser mesh levels are used to compute corrections for the solution of the system on the finest mesh, thus eliminating the low-frequency components of the error and improving the convergence rate. The key ingredients of the multigrid methods are:

1. **Smoothers, i.e. solvers on particular meshes**—few iterations are performed on each mesh in order to damp the oscillatory components of the error. The local Vanka-type smoothers are used on all mesh levels. This approach was proposed in [12].
2. **Residual calculation**—to project the problem on coarser mesh, its residual must be calculated. The projected residue figures as a right hand side there.
3. **Operators of restriction and prolongation between two consecutive meshes**—they are necessary for the projection of solutions and residual vectors from one mesh to another. For both types of operators, we use the L^2-projections.

Together they form the so called multigrid cycle. We used standard V-cycle (where each mesh level is visited only twice in each multigrid iteration) in our solver.

[2] The global matrices and vectors are of no further interest to us. We denote them by ˜ to clearly distinguish them from the components of local systems used throughout the text.

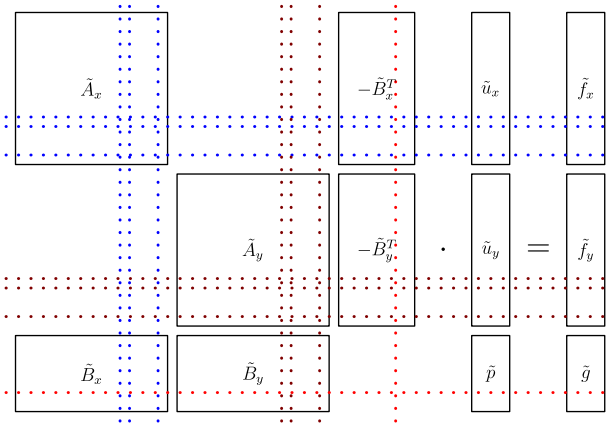


Fig. 3. Extraction of the local system from the global one.

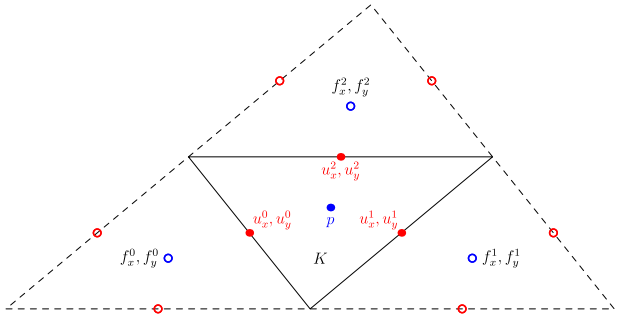


Fig. 4. Nodes of local system for triangle K—7 unknowns (full dots) and 15 off-element values (circles) used to assemble f_x, f_y.

4. Vanka-type smoothers

The Vanka-type smoothers are technically block Gauss–Seidel methods. In case of the P_0/P_1 Crouzeix–Raviart elements, the blocks consist of 7 local degrees of freedom connected with each triangle: three pairs of velocity components in the edge midpoints, plus one pressure unknown in the barycenter.

To construct the local system for given triangle K, we take the respective seven rows from (5), multiply the off-element entries by the respective components of the solution and add the result to the right-hand side; see Figs. 3 and 4.

Local systems have the same structure as the global system matrix (5), consisting of two 3×3 diagonal blocks for x and y components respectively, and two 3×1 blocks representing the divergence terms

$$\begin{pmatrix} \mathbf{A}_x^K & \mathbf{0} & -\mathbf{b}_x^K \\ \mathbf{0} & \mathbf{A}_y^K & -\mathbf{b}_y^K \\ (\mathbf{b}_x^K)^T & (\mathbf{b}_y^K)^T & \mathbf{0} \end{pmatrix} \begin{pmatrix} \mathbf{u}_x^K \\ \mathbf{u}_y^K \\ p^K \end{pmatrix} = \begin{pmatrix} \mathbf{f}_x^K \\ \mathbf{f}_y^K \\ g^K \end{pmatrix}. \tag{6}$$

This system can be solved efficiently via the Schur complement approach; we omit the superscript K for better readability

$$\begin{pmatrix} \mathbf{I} & \mathbf{0} & -\mathbf{A}_x^{-1}\mathbf{b}_x \\ \mathbf{0} & \mathbf{I} & -\mathbf{A}_y^{-1}\mathbf{b}_y \\ \mathbf{0} & \mathbf{0} & \mathbf{b}_x^T\mathbf{A}_x^{-1}\mathbf{b}_x + \mathbf{b}_y^T\mathbf{A}_y^{-1}\mathbf{b}_y \end{pmatrix} \begin{pmatrix} \mathbf{u}_x \\ \mathbf{u}_y \\ p \end{pmatrix}$$

$$= \begin{pmatrix} \mathbf{A}_x^{-1}\mathbf{f}_x \\ \mathbf{A}_y^{-1}\mathbf{f}_y \\ g - \mathbf{b}_x^T\mathbf{A}_x^{-1}\mathbf{f}_x - \mathbf{b}_y^T\mathbf{A}_y^{-1}\mathbf{f}_y \end{pmatrix}. \tag{7}$$

We invert $\mathbf{A}_x$ and $\mathbf{A}_y$ by constructing their adjugates, i.e. $\mathbf{A}_x^{-1} = \mathbf{A}_x^{\text{adj}}/\det \mathbf{A}_x$. The determinant of $\mathbf{A}_x$ is computed using the cofactor expansion along the first column, i.e. $\det(\mathbf{A}_x) = \sum_{j=1}^{3} a_{j1}\alpha_{1j}$, where α_{ij} denotes the ijth element of $\mathbf{A}_x^{\text{adj}}$.

The blocks $\mathbf{A}_x^{-1}$ and $\mathbf{A}_y^{-1}$ for a given timestep are stored between the multigrid cycles, which reduces the computation time of subsequent iterations. We keep the blocks for both components on all triangles despite the higher memory cost.[3]

Algorithm 1 Local system preparation

for all $K \in \mathcal{T}_l$ **do**
  build $\mathbf{A}_x$, $\mathbf{A}_y$ and $\mathbf{b}_x$, $\mathbf{b}_y$ for element K
  compute inverses $\mathbf{A}_x^{-1}$ and $\mathbf{A}_y^{-1}$
  compute auxiliary vectors $\mathbf{c}_x = \mathbf{A}_x^{-1}\mathbf{b}_x$ and $\mathbf{c}_y = \mathbf{A}_y^{-1}\mathbf{b}_y$
  compute Schur complement $s = \mathbf{b}_x^T\mathbf{c}_x + \mathbf{b}_y^T\mathbf{c}_y$
end for

In each timestep n, we first update the global matrices $\mathbf{A}_x(\mathbf{u}_x^{n-1})$ and $\mathbf{A}_y(\mathbf{u}_y^{n-1})$ and the right-hand sides $\mathbf{f}_x(\mathbf{u}_x^{n-1})$ and $\mathbf{f}_y(\mathbf{u}_y^{n-1})$ for all mesh levels using the restriction operator for velocity.[4] Then, we prepare the local systems for all elements on each mesh $\mathcal{T}_l$ using Algorithm 1.

The multigrid solver then uses a standard V-cycle, where in one iteration of the smoother local systems for all elements are solved via the Algorithm 2. The newly computed local values of velocity and pressure are then used to update the global solution vectors $\mathbf{u}_x$, $\mathbf{u}_y$ and $\mathbf{p}$.

Algorithm 2 Solving local system for $K \in \mathcal{T}_l$

build the right-hand side of the system (6) for element K
compute $\mathbf{q}_x = \mathbf{A}_x^{-1}\mathbf{f}_x$ and $\mathbf{q}_y = \mathbf{A}_y^{-1}\mathbf{f}_y$
compute $\tilde{g} = g - \mathbf{b}_x^T\mathbf{q}_x - \mathbf{b}_y^T\mathbf{q}_y$
compute local pressure $p = \tilde{g}/s$ (s is the Schur complement from Algorithm 1)
compute $\mathbf{u}_x = \mathbf{q}_x + p\,\mathbf{c}_x$
compute $\mathbf{u}_y = \mathbf{q}_y + p\,\mathbf{c}_y$

5. Parallelization and GPU implementation

The parallelization for both CPU and GPU will be described in this section, as the general concept is common for both versions (as shown in Fig. 6). The GPU specifics (mostly reordering of data for faster GPU accesses) will be mentioned when appropriate.

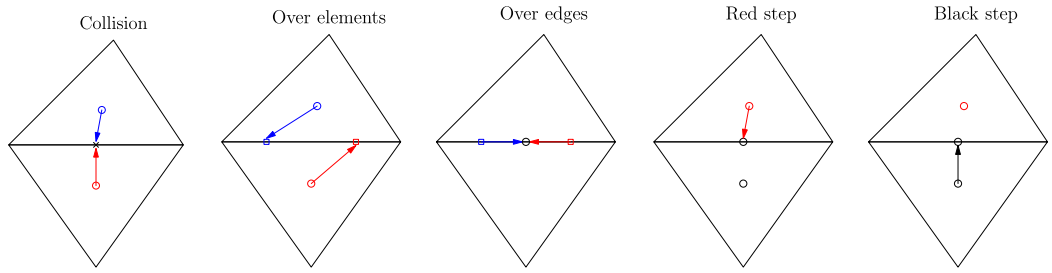
The key multigrid components are implemented by looping over all elements of the mesh and performing the corresponding operations locally on one element. For Vanka-type smoothers, this is natural approach as described above. The prolongation and residual restriction operators are implemented element-wise as well. Assembling the global system (5) and its updating at the beginning of each timestep is based on local contributions and it is done element-wise as well. When looping over elements, write conflicts occur while updating global values shared by multiple elements—in our case the edge-centered values which are accessed from two threads resolving the operation on the adjacent triangles.

We employ two solutions to prevent these conflicts (Fig. 5(a)):

1. In the first approach (Fig. 5(b)), we use auxiliary arrays. For each edge in the mesh, we denote the adjacent triangle with

[3] The blocks $\mathbf{A}_x$ and $\mathbf{A}_y$ only differ on the elements which have non-zero intersection with the free-slip boundary Γ_{top}.

[4] The matrices $\mathbf{B}_x$ and $\mathbf{B}_y$ depend only on the spatial discretization. They are constructed once at the beginning of the computation and do not need to be updated.



(a) Collision. (b) Saving into auxiliary arrays. (c) Using Red–Black method.

Fig. 5. Writing collisions for edge values and solutions to prevent them.

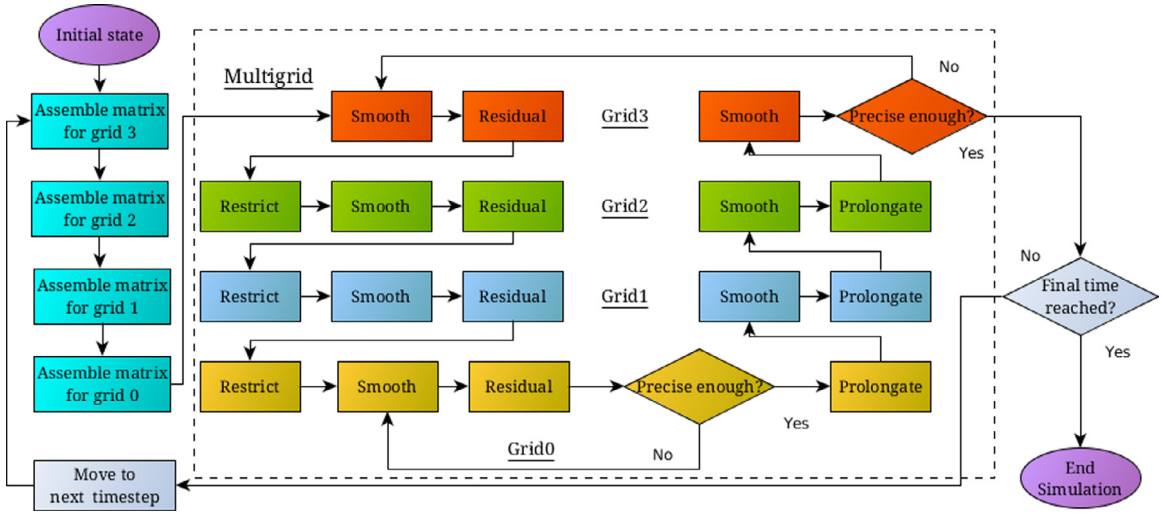


Fig. 6. Flowchart of both CPU and GPU version. Only the initial condition and the numerical mesh is set-up on CPU. The main computation runs entirely either on CPU or GPU. The figure shows general structure of the solver and individual areas of time measurement.

the lower global index as primary with respect to that edge and the other one as secondary. Each triangle holds three boolean values representing whether it is primary for the respective edge. Based on this division, the updates are stored into two separate arrays and combined together afterwards. The restriction and prolongation operators use this approach.

2. By Red–Black method (Fig. 5(c)), assuming all vertices have even degree, we can employ a red–black coloring of triangles. First, we loop over the red triangles storing the contributions into an auxiliary array. After all red triangles have been processed, we perform an update. Then, we process the black triangles in the same fashion. The smoother and the global system update use this approach.

5.1. GPU programming specifics

For detailed description and programming techniques on the GPU, we refer to [13–15]. Let us remind that for the design of an efficient algorithm for the GPU, we must fulfill the following requirements:

1. minimize the communication between the GPU and the CPU,
2. minimize the number of divergent threads,
3. minimize the uncoalesced memory accesses to the GPU global memory.

The particular requirements are fulfilled as follows:

1. In our GPU version, the whole simulation apart from the initial construction of the mesh hierarchy is performed on the GPU, thus no data transfer between CPU and GPU memory occurs throughout the computation.
2. The only branching in our kernels that can cause divergent threads comes from treatment of boundary elements. Since the number of boundary elements is typically small compared to the number of all elements for large problems, the negative effect of divergent threads is negligible.
3. Our main concern was about uncoalesced memory accesses. The majority of CUDA kernels map one thread to one mesh element. Most data therefore needs to be stored by elements. Storing multiple values for each element is done in SoA (structure of arrays) fashion, as opposed to the CPU version where AoS (array of structures) is used. This helps us to achieve coalesced accesses for most read/write operations except when accessing edge-related unknowns (due to the parallelization over elements) and fetching unknowns from neighboring elements on an unstructured mesh.

As a result, the GPU and CPU versions are almost identical, the only significant difference being the data ordering.

5.2. Memory management

First, we show the amount of memory needed to store the numerical mesh and the unknowns on CPU and on GPU. The mesh is represented by its elements (triangles) and edges. For each element, we need the following amount of data:

- 92 bytes for *local mesh data*, i.e. references between triangles and their edges, coordinates of vertices, type of boundary condition for given edge, references to parent/child elements in multigrid hierarchy. These are static values which do not change throughout the computation.

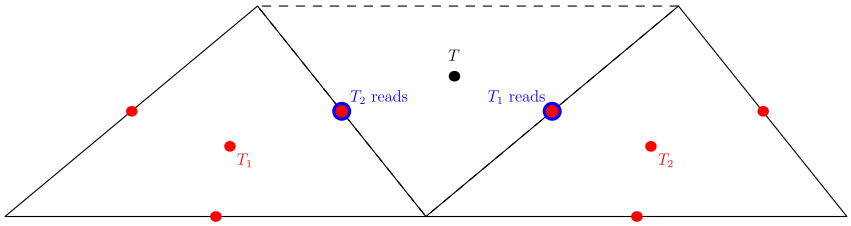


Fig. 7. When constructing the system for T_1, the local unknowns of T_2 are needed (and vice versa), despite both triangles having the same color.

- 8 bytes of additional *coloring and reordering* data for the red–black coloring. These are also static.
- 464 bytes of data for the *main simulation*, i.e. representation of the global matrix, boundary conditions, vectors for velocity and pressure, auxiliary arrays.
- 312 bytes for *local inversions*, used to speed up the computation.
- 44 bytes of *GPU specific redundant data*. This data help us achieve coalesced accesses in GPU kernels, therefore they only apply for GPU version. They include: indices of edges on the finer mesh related to the coarse mesh element, used for prolongation and restriction.

5.3. Assembling global linear system

The computation starts by assembling the linear system (5)—see Fig. 6. To avoid conflicts between two adjacent elements processed in parallel by two threads, first the red elements are processed, and then the black ones follow. Instead of storing the individual blocks $\tilde{\mathbf{A}}_x$ and $\tilde{\mathbf{A}}_y$ of the global matrix which only differ in positions corresponding to boundary data, we store a single block $\tilde{\mathbf{A}}$ where all nodes are treated as unknowns and we only distinguish them in local systems. The global system (5) is constructed for each mesh level.

For the finest mesh $\mathcal{T}^L$, we also construct the right-hand side of (5). The computation of $\tilde{\mathbf{g}}$ is done by elements, while the computation of $\tilde{\mathbf{f}}_x$ and $\tilde{\mathbf{f}}_y$ is done by edges.

After having prepared the global system, we run Algorithm 1 and store the local data for each element K which will be reused throughout the whole timestep: the local inversions $\mathbf{A}_x^{-1}$, $\mathbf{A}_y^{-1}$, the vectors $\mathbf{c}_x$ and $\mathbf{c}_y$ and the Schur complement s. To do that, we must first extract the appropriate $\mathbf{A}_x$, $\mathbf{A}_y$, $\mathbf{b}_x$ and $\mathbf{b}_y$ from the global system.

5.4. Smoother

The red–black modification of the Vanka-type smoother is not entirely straightforward, as coloring alone does not solve all parallel issues. To construct the right-hand side of a local system of either color, the off-element velocity values are needed—which are updated through the local systems of both colors; see Fig. 7.

In order to make this work in parallel, we use the old velocity values while processing the red triangles. When finished, we update the velocity field and proceed with the black triangles. Pressure values can be updated in place, as each belongs only to one triangle.

On the GPU, the values for the red and the black elements are stored separately, and thus coalesced memory accesses are achieved. Based on the version of the algorithm, the inversions of the local blocks are either constructed in advance and coalescedly read, or computed on the fly.

Detailed description of smoothing kernel on GPU is given by Algorithm 3. For each read/write operation it states whether it is coalesced or not.

Algorithm 3 Red-black Vanka kernel for solving local systems on the GPU

```
compute id for element K from threadId and currColor
fetch local inversions from Algorithm 1 (coalesced)
fetch right hand side for the pressure g (coalesced)
for each face do
  fetch faceId and faceType
  fetch velocity right hand side (i.e. f ), via faceId (noncoalesced)
  //Add nonlocal values (i.e. product of global matrix and unknowns of
  //neighbouring triangle) to the right hand side
  if not faceType is boundary then
    fetch ids of neighbouring velocities and pressure (coalesced)

    fetch neighbouring velocities and pressure (noncoalesced)
    fetch appropriate values from global matrix (coalesced)
    do the multiplication and add result to right hand side
  end if
end for
solve local system using the Algorithm 2 (without first step)
save new pressure (coalesced)
for each face do
  save new velocity (noncoalesced)
end for
```

5.5. Prolongation and restriction

When descending down the multigrid cycle, the residual vector on the current mesh is projected to create the right hand side for the coarser mesh.

The GPU kernel for this task is parallelized over the elements of the coarser mesh and uses coalesced read of all indices[5] and coalesced write for saving triangle-centered values.

The prolongation procedure applies the corrections for velocity and pressure from the coarser mesh to improve the solution on the finer mesh. It is parallelized in the same fashion as the restriction procedure.

6. Numerical experiments

The numerical experiments were performed on a system equipped with Intel Core i7-3770K (Ivy Bridge) CPU running at 3.5 GHz and containing 4 × 256 KB L2 cache and 8 MB L3 cache and dual-channel DDR3 memory architecture with the throughput of 34 GB/s. The GPU code was tested on Nvidia Tesla K40 with 12 GB RAM and 288 GB/s bandwidth. To demonstrate the advantages of this architecture (Kepler) over the previous one (Fermi), several tests were performed on Nvidia GeForce GTX 480 with 1.5 GB RAM and 134 GB/s bandwidth. The ECC memory corrections on Tesla K40 decrease the performance by almost 10%, and so they were

[5] In the GPU implementation, each triangle contains nine additional indices to its descendants' edges in order to prevent indirect memory accesses.

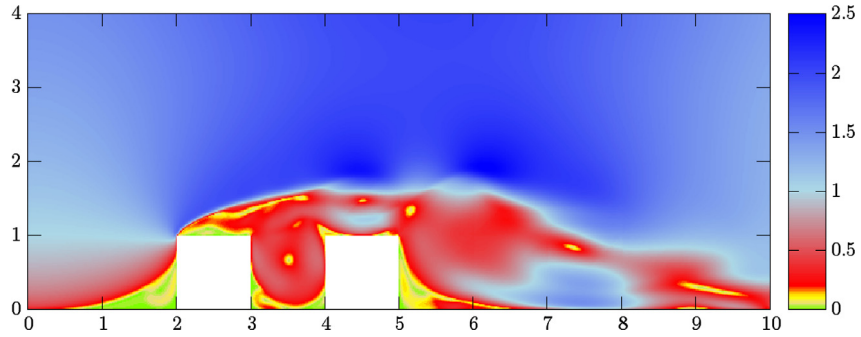


Fig. 8. Example of 2D flow (bottom). Velocity magnitude is displayed.

Table 1
The table shows how much time is spent in each part of the solver.

Algorithm	Tesla K40 GPU	i7-3770 CPU (sequential)
Smoother	88%	85%
Residual	4%	5%
Prolongation	1%	2%
Restriction	1%	2%
Global matrix	2%	5%

turned off. The code was completely written in C/C++ and compiled with gcc 4.8, Intel icc 15.0 and CUDA 6.5.

All simulations were computed in double precision. The initial mesh $\mathcal{T}^0$ depicted in Fig. 2 containing 76 triangles was uniformly refined up to six times. The resulting degrees of freedom (DOFs) for levels 4, 5 and 6 were 310,848, 1,244,288 and 4,978,944 respectively. The result of flow over a pair of buildings with a street canyon is shown in Fig. 8. The Reynolds number was 10^5.

We start with Table 1 showing how much time is spent in each part of the multigrid solver. We see that the most time-consuming part is the smoothing. Even though the other parts take less than 8% on the GPU, they must be performed on the GPU as well to avoid data transfer between the CPU and the GPU in each timestep.

The comparison between the CPU and the GPU version is shown in Table 2. The CPU code was optimized to allow the use of AVX vectorization instructions by the Intel compiler, especially when solving the local systems which is the most CPU time consuming operation. The main performance bottleneck is, however, the memory subsystem. The largest portion of data that must be transferred between memory modules and CPU consists of the local blocks. There is no data reuse and so optimization of the cache utilization would not help. In our code, all local blocks are stored in a single continuous block of data. It avoids memory fragmentation and it helps to access the data in memory efficiently. Apart from the parallel OpenMP version, the sequential one (with standard sequential Gauss–Seidel like smoother) was tested as well. The results show that the parallel CPU version is efficient only with two threads. When employing all four cores, the memory subsystem is not fast enough to deliver data to the CPU. At this point, the GPU can benefit from its much faster memory modules. As we can see, the speed-up of the GPU grows with larger meshes.

Comparison of different GPU architectures – Kepler (Tesla K40) and Fermi (GeForce GTX 480) – together with different handling of the local inversions is shown in Table 3. Two versions of handling local inversions $\mathbf{A}_x^{-1}, \mathbf{A}_y^{-1}$ were used:

Table 3
Performance comparison of different graphics cards for storing (SI) and non-storing (NSI) handling of local inversions.

DOFs	GeForce GTX 480			Tesla K40		
	Time [s]		Speed-up	Time [s]		Speed-up
	NSI	SI		NSI	SI	
78 K	3.1	2.1	1.48	3.2	2.9	1.10
311 K	2.9	2.1	1.38	2.4	2.2	1.09
1247 K	12.1	9.2	1.32	7.8	7.5	1.04

Stored inversions (SI) local inversions are computed once at the beginning of each timestep and stored for subsequent use. It is faster, but needs about 30% more memory.

Non-stored inversions (NSI) inversions are computed every time they are needed. It makes the main smoothing kernel much more computationally demanding.

NSI approach was not used on the CPU, where it is more than ten times slower. On the GPU, this has much smaller impact. NSI for large problems is only about 30% slower on GeForce GTX 480 and less than 5% slower on modern Tesla K40. We see that, NSI scales better on new GPU architectures. On the Tesla it is over 40% faster than on the GTX, compared to only 20% for SI approach. The reason is that the real computational power of new GPUs grows faster than the real memory throughput. This experiment shows that matrix-free methods might be good choice for the GPUs in the future.

At the end, we present Tables 4 and 5 showing the time spent on particular meshes of the multigrid solver (Fig. 6 shows computational parts belonging to particular grids with the same color). The significant amount of time spent on $\mathcal{T}_0$ is due to the fact that we actually solve this system precisely (up to the tolerance required for the global residual). This effect is much worse on GPU due to the small size of $\mathcal{T}_0$, which allows using only a fraction of GPU's computational units. The Kepler architecture of Tesla K40 with lower per-unit performance is even slower for this task. While the GTX 480 card has 480 computational units running on 1400 MHz, Tesla K40 has 2880 units running at 745 MHz. One unit on Tesla K40 has lower performance and therefore more threads must be employed.

Table 2
Performance comparison between the CPU (sequential and parallel) and the GPU version.

DOFs	i7-3770K CPU Time [s]				Tesla K40 GPU				
	Seq.	1 core	2 cores	4 cores	Time [s]	Speed-up			
						Seq.	1 core	2 cores	4 cores
78 K	6.8	8.2	5.0	3.7	2.9	2.35	2.89	1.72	1.28
311 K	14.4	18.6	11.8	10.2	2.2	6.55	8.45	5.36	4.63
1247 K	115.3	120.4	76.7	67.2	7.5	15.38	16.05	10.23	8.96
4989 K	611.2	664.4	424.1	365.0	32.65	18.72	20.34	12.99	11.18

Table 4
Computation time spent on each mesh level for Intel i7-3770 CPU and Nvidia Tesla K40 GPU.

Grid level	DOFs	i7-3770 CPU		Tesla K40 GPU		Speed-up
		Time [s]	Percentage	Time [s]	Percentage	
0	336	0.31	0.5%	1.82	24%	0.17
1	1280	0.05	0.1%	0.10	1%	0.5
2	4992	0.09	0.2%	0.09	1%	1.0
3	20 K	0.34	0.5%	0.12	2%	2.83
4	78 K	2.63	4%	0.24	3%	11
5	311 K	12.05	18%	0.79	10%	15
6	1247 K	48.33	75%	3.01	40%	16

Table 5
Computation time spent on each mesh level for Tesla K40 GPU and GeForce GTX 480 GPU.

Grid level	DOFs	GeForce GTX 480 GPU		Tesla K40 GPU		Speed-up
		Time [s]	Percentage	Time [s]	Percentage	
0	336	1.17	13%	1.82	24%	0.64
1	1280	0.08	1%	0.10	1%	0.8
2	4992	0.08	1%	0.09	1%	0.88
3	20 K	0.12	1%	0.12	2%	1.0
4	78 K	0.34	4%	0.24	3%	1.41
5	311 K	1.21	13%	0.79	10%	1.41
6	1247 K	4.66	50%	3.01	40%	1.54

Conclusion

We presented a GPU implementation of the multigrid method for the mixed finite element approximation of viscous incompressible Navier–Stokes equations in 2D. The solver based on Vanka-type smoothers for the Crouzeix–Raviart finite elements was completely parallelized using the red–black coloring. Compared to the sequential CPU code, we achieved a speed-up 19. Compared to the multicore simulation with 4 cores, the speed-up was 11. In our next work, we would like to extend the solver for multi-GPU systems. Such systems exhibit properties of distributed memory parallel architectures. Therefore appropriate domain decomposition method must be applied.

Acknowledgments

This work was partially supported by the project of the Center of Excellence No. 14-36566G of the Grant Agency of the Czech Republic, project No. 15-27178A of the Czech Ministry of Health and project of the Student Grant Agency of the Czech Technical University in Prague No. SGS14/206/OHK4/3T/14.

References

[1] V.W. Lee, Ch. Kim, J. Chhugani, M. Deisher, D. Kim, A.D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, P. Dubey, Debunking the 100x GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU, in: ISCA'10, 2010, pp. 451–460.

[2] C.E. Scheidegger, J.L.D. Combaand, R.D. da Cunha, Practical CFD simulations on programmable graphics hardware using SMAC, in: Computer Graphics forum, 24, 2005, pp. 715–728.

[3] E. Elsen, P. LeGresley, E. Darve, Large calculation of the flow over a hypersonic vehicle using a GPU, J. Comput. Phys. 227 (2008) 10148–10161.

[4] T. Brandvik, G. Pullan, Acceleration of a 3D Euler solver using commodity graphics hardware, in: 46th AIAAA Aerospace Sciences Meeting, 2008.

[5] A. Corrigan, F.F. Camelli, R. Löhner, J. Wallin, Running unstructured grid-based CFD solvers on modern graphics hardware, Internat. J. Numer. Methods Fluids 66 (2) (2011) 221–229.

[6] A.F. Shinn, S.P. Vanka, Implementation of a semi-implicit pressure-based multigrid fluid flow algorithm on a graphics processing unit, in: Proceedings of the ASME, 2009, pp. 125–133.

[7] J.C. Thibault, I. Senocak, CUDA implementation of a Navier–Stokes solver on multi-GPU desktop platforms for incompressible flows, in: 47th AIAA Aerospace Science Meeting, paper no. AIAA-2009-758, 2009.

[8] M. Griebel, P. Zaspel, A multi-GPU accelerated solver for the three-dimensional two-phase incompressible Navier–Stokes equations, Comput. Sci. Res. Dev. 25 (1-2) (2010) 65–73.

[9] F. Schieweck, L. Tobiska, An optimal order error estimate for upwind discretization of the Navier–Stokes equation, Numer. Methods Partial Differential Equations 12 (4) (1996) 407–421.

[10] M. Benzi, G.H. Golub, J. Liesen, Numerical solution of saddle point problems, Acta Numer. 14 (2005) 1–137.

[11] S. Turek, Efficient Solvers for Incompressible Flow Problems: An Algorithmic and Computational Approach, Vol. 6, Springer Science & Business Media, 1999.

[12] S.P. Vanka, Block-implicit multigrid solution of Navier–Stokes equations in primitive variables, Comput. Phys. (65) (1986) 138–158.

[13] S. Cook, CUDA Programming: A Developer's Guide to Parallel Computing with GPUs, Morgan Kaufmann, 2012.

[14] T. Oberhuber, A. Suzuki, J. Vacata, New row-grouped CSR format for storing the sparse matrices on GPU with implementation in CUDA, Acta Tech. 56 (2011) 447–466.

[15] T. Oberhuber, A. Suzuki, V. Žabka, The CUDA implementation of the method of lines for the curvature dependent flows, Kybernetika 47 (2011) 251–272.