A portable OpenCL-based unstructured edge-based Finite Element Navier-Stokes solver on graphics hardware

R. Rossi[a,b], F. Mossaiby[c,*], S. Idelsohn[a,d]

[a]*Centre Internacional de Mètodes Numèrics en Enginyeria (CIMNE)*
[b]*UPC, BarcelonaTech, Campus Norte UPC, 08034 Barcelona, Spain*
[c]*Department of Civil Engineering, Faculty of Engineering, University of Isfahan, 81744-73441 Isfahan, Iran*
[d]*ICREA Research Professor at CIMNE, Barcelona, Spain*

Abstract

The rise of GPUs in modern high-performance systems increases the interest in porting portion of codes to such hardware. The current paper aims to explore the performance of a portable state-of-the-art FE solver on GPU accelerators. Performance evaluation is done by comparing with an existing highly-optimized OpenMP version of the solver. Code portability is ensured by writing the program using the OpenCL 1.1 specifications, while performance portability is sought through an optimization step performed at the beginning of the calculations to find out the optimal parameter set for the solver. The results show that the new implementation can be several times faster than the OpenMP version.

Keywords: Unstructured grids, Navier-Stokes, Edge-based, GPU, OpenCL, OpenMP

1. Introduction

The solution of the incompressible Navier-Stokes problem, which describes the motion of Newtonian fluids, represents an open challenge for the numerical community. Given the importance of the problem, a large effort was spent over the years in the development of dedicated numerical techniques to improve the speed and the accuracy of the solution process. The use of lattice-based approaches such as the Finite

[*]Corresponding author. Tel.: +98 (311) 793 4015, Fax: +98 (311) 793 2089.
Email addresses: `rrossi@cimne.upc.edu` (R. Rossi), `mossaiby@eng.ui.ac.ir` (F. Mossaiby), `sergio@cimne.upc.edu` (S. Idelsohn)

Differences or Lattice-Boltzmann schemes lead to the development of highly-efficient schemes, which can deal effectively with very large computational meshes. The relative simplicity of the computational kernels together with the highly regular structure of the computations were found to fit perfectly to the architectural needs of modern accelerators. Various authors (see e.g. [1]) were able to achieve performance boosts by developing optimized kernels with respect to their single-CPU counterparts. Although local adaptivity was shown to be very effective, any modification with respect to the simple approach of using regular meshes typically leads to a decrease in the computational efficiency and a drop in the performance boost for hardware accelerated algorithms. Furthermore, all of such techniques find major limitations in dealing with complex geometries and curved boundaries.

Unstructured discretization of the space, typically based on tetrahedral meshes, represent a possible solution for such problems. The strength of such approaches is the possibility of using body-fitted meshes and spatially adapted discretizations of the space. The price paid to achieve this advantage is an increasingly irregular computational pattern which reflects in a variable number of edges surrounding the nodes of the computational mesh. Such a situation does not represent a major problem for cache-based processors used in conjunction with OpenMP or MPI programming paradigms, since computations can be organized so to take full advantage of the cache, while the number of parallel OpenMP threads is typically kept low. However, prior experiences of the authors [2], confirmed by the reports of others [3], seem to suggest that only low speedups can be achieved with respect to the CPU-only solutions.

The current paper aims to examine the OpenCL porting of an OpenMP edge-based solver so as to identify and discuss the performance bottlenecks. The paper starts with a brief description of the solution algorithm used, followed by the presentation of the data structure employed and of the structure of the parallel computations. An effort is performed to make the discussion as implementation-independent as possible. The impact of using black-box solvers (such as the ViennaCL library [4]) in comparison to in-house optimized implementations for the solution of the implicit pressure step is evaluated. Benchmarking data over platforms with different software and hardware configurations are presented so as to allow a broader comparison. Finally the impact of a run-time optimization step will be evaluated. The proposed implementation is open-source and freely available as a part of the Kratos framework [5]. More information can be found in the Wiki page for Kratos [6].

2. Finite Element Formulation

Between the many existing possibilities for the solution of Navier-Stokes problem, we use a fractional-step approach. In this scheme we solve explicitly the momentum equation (using a 4-step Runge-Kutta scheme) and implicitly the pressure correction step. A detailed discussion of the algorithm is presented in [7] for the incompressible case and in [8] for the solution of Low-Mach compressible problems.

In order to understand the basic concepts of the method, we may start by considering the strong form of the Navier-Stokes equations, written for the constant-density case

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} - \nu \Delta \mathbf{u} + \nabla p = \mathbf{b} \tag{1}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{2}$$

where $\mathbf{u}$ is the velocity, ν is the kinematic viscosity, p is the pressure and $\mathbf{b}$ is the applied body force . By applying the standard Galerkin approach, using the test functions $\mathbf{v}$ and q, we obtain

$$(\mathbf{v}, \mathbf{b}) - \left(\mathbf{v}, \frac{\partial \mathbf{u}}{\partial t}\right) - (\mathbf{v}, \mathbf{u} \cdot \nabla \mathbf{u}) + (\mathbf{v}, \nu \Delta \mathbf{u}) - (\mathbf{v}, \nabla p) = \mathbf{0} \tag{3}$$

$$(q, \nabla \cdot \mathbf{u}) = 0 \tag{4}$$

Equations(3-4) define the discrete equivalent of the original continuous problem. Since our goal is to use low-order simplicial meshes and equal order velocity-pressure pairs, a stabilized formulation is needed to allow the solution of the resulting mixed $(\mathbf{u}, \mathbf{p})$ problem. Different possibilities exist for this purpose [9]. We favor here the use of the so-called split-OSS approach [10], which is known to work properly in a wide range of applications. FIC stabilization [11, 12] could be used as an alternative since it leads to very similar discrete forms. Since a discussion of the properties of the chosen stabilization method falls outside the scope of this work, we refer the reader to the literature for a detailed description of the properties of such technique. In the following we express the stabilization terms as the non linear operators $\mathbf{S}(\mathbf{u}) := (\mathbf{u} \cdot \nabla \mathbf{v}, \tau \Pi_\perp (\mathbf{u} \cdot \nabla \mathbf{u}))$ and $\mathbf{S}_p(\mathbf{p}) := (q, \tau \Pi_\perp (\nabla p))$ where $\Pi_\perp$ represents the orthogonal projection operator and τ is a suitably defined scalar. Reader should check [10] or [14] for a detailed description of such terms. The resulting final form of the discrete equations is

$$\mathbf{M}\frac{\partial \mathbf{u}}{\partial t} + [\mathbf{C}(\mathbf{u}) - \nu \mathbf{L} + \mathbf{S}(\mathbf{u})]\,\mathbf{u} + \nabla \mathbf{p} = \mathbf{F} \tag{5}$$

$$\mathbf{D}\mathbf{u} + \mathbf{S}_p \mathbf{p} = \mathbf{0} \tag{6}$$

where $\mathbf{M}$ is the lumped mass matrix and

$$\nabla_{IJ} := \int_\Omega N_I \nabla N_J \mathrm{d}\Omega \tag{7}$$

$$\mathbf{G}_{IJ} := \int_\Omega \nabla N_I N_J \mathrm{d}\Omega \tag{8}$$

$$\mathbf{D}_{IJ} := \int_\Omega N_I \nabla N_J^T \mathrm{d}\Omega = \nabla_{IJ}^T \tag{9}$$

$$\mathbf{L}_{IJ} := \int_\Omega \nabla N_I \cdot \nabla N_J \mathrm{d}\Omega \tag{10}$$

are linear operators obtained by integrating over the domain the Finite Element shape functions indicated with N. The convection operator $\mathbf{C}(\mathbf{u})$ is a non-linear term which depends on the velocity. It can be defined as

$$\mathbf{C}_{IJ} := \int_\Omega N_I \mathbf{u} \cdot \nabla N_J^T \mathrm{d}\Omega \tag{11}$$

Equation (5) has 3 components in 3D. This is reflected in $\mathbf{G}$ and $\mathbf{D}$ being respectively 3×1 and 1×3 matrices for each couple of indices IJ of the FE mesh. The description is completed by the exact definition of the stabilization terms. A detailed description of such terms, particularized to the Split-OSS case, and using the same notation used here can be found in [13]. Choosing a fractional-step approach implies approximating the original system of equations as

$$\mathbf{M}\frac{\partial \hat{\mathbf{u}}}{\partial t} + [\mathbf{C}(\hat{\mathbf{u}}) - \nu \mathbf{L} + \mathbf{S}(\hat{\mathbf{u}})]\,\hat{\mathbf{u}} + \nabla \mathbf{p}_n = \mathbf{F} \tag{12}$$

$$\mathbf{M}\frac{\partial (\mathbf{u} - \hat{\mathbf{u}})}{\partial t} + \frac{\Delta t}{2} \nabla (\mathbf{p}_{n+1} - \mathbf{p}_n) = \mathbf{0} \tag{13}$$

$$\mathbf{D}\mathbf{u} + \mathbf{S}_p \mathbf{p}_{n+1} = \mathbf{0} \tag{14}$$

where $\hat{\mathbf{u}}$ is the so-called fractional-step velocity. The fractional-step velocity is modified at each step so that its value in the past coincides with the velocity $\mathbf{u}_n$, that is, $\hat{\mathbf{u}}_n = \mathbf{u}_n$.

Since Equation (12) is still continuous in time, we have different options in choosing a time integration scheme to properly advance in time the momentum equation (the first of the three equations above). As we stated previously, our choice in the current work is the use of a 4th order Runge-Kutta scheme. As observed in [7, 14], by making the fundamental approximation of considering the end-of-step velocity to *depend linearly* on the pressure despite the non-linearity of the convection terms, we may conclude that the end-of-step velocity can be expressed as

$$\mathbf{u} = \hat{\mathbf{u}} + \frac{\Delta t}{2}\mathbf{M}^{-1}\nabla\left(\mathbf{p}_{n+1} - \mathbf{p}_n\right) \tag{15}$$

where $\mathbf{p}_{n+1}$ and $\mathbf{p}_n$ indicate respectively the *new* value of the pressure (at time t_{n+1}), and its latest known value (at time t_n). Substituting symbolically this expression into the mass conservation equation (Equation (14)) and replacing the discrete Laplacian $\mathbf{DM}^{-1}\mathbf{G}$ with the continuous one $\mathbf{L}$ we obtain the equation

$$\frac{\Delta t}{2}\mathbf{L}\left(\mathbf{p}_{n+1} - \mathbf{p}_n\right) + \mathbf{S}_p\mathbf{p}_{n+1} = \mathbf{D}\hat{\mathbf{u}} \tag{16}$$

The above equation needs to be solved for the pressure. This ultimately leads to a solution strategy articulated in the following steps:

1. Solve Equation (12) for the fractional-step velocity, $\hat{\mathbf{u}}$.
2. Solve Equation (16) for the pressure, p_{n+1}.
3. Solve Equation (15) for the end-of-step velocity, $\mathbf{u}$.

From a practical point of view, the equation to be solved in Step 1 is mathematically equivalent to the solution of a convection-diffusion equation for each of the velocity components. The pressure gradient and the external pressure term act here as source terms. The idea we leverage in this work is to take advantage of the efficient edge-based data structure described in [13, 14] to allow the efficient computation of the fractional-step velocity.

The essential idea is that, basing on a slight approximation of the viscous and convective term, it is possible to *approximate* all of the operators described in terms of constant operators. Such operators can be computed at the beginning of the calculations and stored for each edge IJ of the Finite Element mesh, making the approach very appealing for use on GPU since the operators can be computed on the CPU and transferred to the GPU memory for later usage.

Unfortunately while such operation is straightforward for the gradient $\mathbf{G}$ and divergence $\mathbf{D}$ operators, this is not the case neither for the convective term nor for the viscous operator. We will briefly discuss the approximations needed in the next paragraph, devoted to the edge based data structure. The final requirement for the implementation of the fractional step algorithm on GPUs, is the implementation of a solver for the pressure equation. In this work we make use of a classic CSR sparse matrix storage for the computation of the pressure system. This allows us to compare different iterative solvers and preconditioner both in CPU and in GPU.

3. Data Structure

As pointed out in the previous section, the efficiency of the new solver relies on the construction of *constant* operators, to be computed at the beginning of the calculations for each edge IJ of the mesh. Our aim in the current section is to describe and justify how such operators can be defined and used, so as to define an *edge-based* approach. Such approach is based on the partition of unity property of the Finite Element Method, which states that

$$\sum_J N_J(\mathbf{x}) = 1 \quad \forall \mathbf{x} \in \Omega \tag{17}$$

from which follows

$$\sum_J \nabla N_J(\mathbf{x}) = 0 \longrightarrow \nabla N_I = -\sum_{J \neq I} \nabla N_J \tag{18}$$

Here and in the following we will indicate with $\sum_J$ the sum over all the indices J which correspond to the nodes connected by one edge to the node I. Taking into account the above properties, and making some approximations to the convection and viscous terms, it is possible to cast all of the operations needed in terms of a minimal number of operators. Following the presentation in [2] and [13] we store, on each edge IJ with $I \neq J$, the terms ∇_{IJ} and $\mathbf{G}_{IJ}$ together with the integrals

$$\mathbf{M}^c_{IJ} := \int_\Omega N_I N_J \mathrm{d}\Omega \tag{19}$$

$$\mathbf{L}^d_{IJ} := \int_\Omega \nabla N_I \otimes \nabla N_J \mathrm{d}\Omega \tag{20}$$

which are obtained by numerical integration of the shape function and its derivatives on the finite element mesh. Note that Equation(20) defines here a 3×3 matrix. Additionally we will store the lumped mass matrix in vector form so that $M_{II} := \sum_J \int_\Omega N_I N_J d\Omega$.

Despite the terms stored in the edges show obvious symmetries, we deliberately choose to ignore this fact and employ a non-symmetric CSR storage format. This choice is made for reasons of parallel efficiency and it can be understood by analogy with the Sparse Matrix-Vector (SpMV) multiplication algorithm. The SpMV algorithm shows many similarities with the computational structure we employ. In a SpMV product of the type $\mathbf{y} = \mathbf{A}\mathbf{x}$, where $\mathbf{A}$ is given in non symmetric CSR format, every entry $\mathbf{y}_I$ can be computed efficiently as $\mathbf{y}_I = \sum_J \mathbf{A}_{IJ}\mathbf{x}_J$, and the overall algorithm can be written as

```
for I = 0 to rows of A
  y(I) = 0
  for J in non-zeros of the Ith line of A
    y(I) += A(I, J) * x(j)
```

in which the outer loop is embarrassingly parallel and hence a very good candidate for implementation on GPUs.

On the contrary, if we store only the upper-triangular part of $\mathbf{A}$, the algorithm needs to be modified as

```
for I = 0 to rows of A
  y(I) = 0

for I = 0 to rows of A
  for J in non-zeros of the Ith line of A, with J > I
    aIJ = A(I, J)
    y(I) += aIJ * x(J) // Potential write conflict
    y(J) += aIJ * x(I) // Potential write conflict
```

This second algorithm is *not* trivially parallel since a potential write conflict arises in the computations. Furthermore, a separate loop needs to be performed to initialize the **y** vector to zero. This is an undesired feature, since it requires enqueuing two kernels instead of one, thus producing a non negligible overhead when translated to OpenCL.

Since a very high degree of parallelism is needed for efficient use of GPUs we choose the first option and go for the non-symmetric storage trading memory for parallel efficiency.

From a practical point of view, the computational structure we employ for the construction of the residuals could be understood as a sort of modified SpMV algorithm in which each entry $\mathbf{A}_{IJ}$ packs all of the needed entries, and the `*` is *appropriately overloaded*, as we shall describe next. The key advantage of such an approach is that the amount of indirect addressing is largely reduced with respect to the scalar SpMV case, thus improving the efficiency both on cache-based machines and on accelerators. A specific characteristics of our GPU implementation is that this *packing* is done by storing the different terms within a `cl_double16` (a native opencl vector datatype). In our specific implementation the different terms are stored as

$$\mathbf{M}^c_{IJ} \longrightarrow position(0) \tag{21}$$

$$\mathbf{L}^d_{IJ} \longrightarrow position(1-9) \tag{22}$$

$$\nabla_{IJ} \longrightarrow position(10-12) \tag{23}$$

$$\mathbf{G}_{IJ} \longrightarrow position(13-15) \tag{24}$$

thus occupying the whole datatype.

The next subsections address the computation of the terms needed for the residual calculations, showing how the edge data can be used to express the operations needed in the residual calculation (that is, defining how to *overload* the operator `*` of the equivalent SpMV approach).

3.1. Pressure gradient computation

The computation of the pressure gradient is often required in the residual computations. A finite element code requires to either evaluate the *strong* pressure gradient

$$\pi_I := \sum_J \int_\Omega N_I \nabla N_J \mathbf{p}_J \mathrm{d}\Omega \tag{25}$$

or the *weak* one

$$\mathbf{P}_I := \sum_J \int_\Omega \nabla N_I N_J \mathbf{p}_J \mathrm{d}\Omega \tag{26}$$

If we focus on the calculation of π and isolate the index I in the summation, we obtain

$$\pi_I := \sum_J \int_\Omega N_I \nabla N_J \mathbf{p}_J \mathrm{d}\Omega = \sum_{J \neq I} \int_\Omega N_I \nabla N_J \mathbf{p}_J \mathrm{d}\Omega + \int_\Omega N_I \nabla N_I \mathbf{p}_I \mathrm{d}\Omega \tag{27}$$

Using the *partition of unity* property (see Equation (17)) we obtain

$$\pi_I := \sum_{J \neq I} \int_\Omega N_I \nabla N_J \left(\mathbf{p}_J - \mathbf{p}_I\right) \mathrm{d}\Omega \tag{28}$$

showing that the gradient calculation can be expressed in terms of the ∇_{IJ} operator as

$$\pi_I := \sum_{J \neq I} \nabla_{IJ} \left(\mathbf{p}_J - \mathbf{p}_I\right) \tag{29}$$

The implementation of the Navier-Stokes equation also requires the computation of the *weak* pressure gradient. The obvious approach is to use integration by parts starting from the strong one to get

$$\mathbf{P}_I := \sum_J \int_\Omega \nabla N_I N_J \mathbf{p}_J \mathrm{d}\Omega = -\sum_J \int_\Omega N_I \nabla N_J \mathbf{p}_J \mathrm{d}\Omega + \int_\Gamma N_I N_J \mathbf{p}_J \mathbf{n} \mathrm{d}\Gamma \tag{30}$$

where we introduced the new symbol $\mathbf{n}$ to indicate the normal pointing to the outside of the domain. While this approach is clearly possible, it requires storing the domain surface and performing an integration over such boundary. Even if the memory cost of this additional data is typically negligible, since the surface is small compared to the volume, the computation would require on GPUs at least one additional kernel, which is undesirable due to the high latency that characterizes OpenCL kernel enqueuing. For this reason, we prefer addressing directly the computation of the original volume integral, without recurring to additional calculations on the surface. This can be done by considering that

$$\mathbf{P}_I := \sum_J \int_\Omega \nabla N_I N_J \mathbf{p}_J \mathrm{d}\Omega = \sum_{J \neq I} \int_\Omega \nabla N_I N_J \mathbf{p}_J \mathrm{d}\Omega + \int_\Omega \nabla N_I N_I \mathbf{p}_I \mathrm{d}\Omega \tag{31}$$

and using Equation (18)

$$\mathbf{P}_I := \sum_{J \neq I} \int_\Omega \nabla N_I N_J \mathbf{p}_J \mathrm{d}\Omega - \sum_{J \neq I} \int_\Omega \nabla N_J N_I \mathbf{p}_I \mathrm{d}\Omega = \sum_{J \neq I} \left(\mathbf{G}_{IJ} \mathbf{p}_J - \nabla_{IJ} \mathbf{p}_I\right) \tag{32}$$

which allow computing the integral of interest without having to perform surface integrals. Such operation requires accessing both $\mathbf{G}_{IJ}$ and ∇_{IJ} and explains the reason why these gradients are stored. Once again our choice is to trade memory for efficiency.

3.2. Viscous term

Assuming the use of the Laplacian form for the viscous term [15], the computation of the viscous forces can be performed using a similar approach, to give (on each component k)

$$\mathbf{B}_{Ik} := \sum_{J} \int_{\Omega} \nabla N_I \cdot \nu \nabla N_J \mathbf{u}_{Jk} \mathrm{d}\Omega = \sum_{J \neq I} \int_{\Omega} \nabla N_I \cdot \nu \nabla N_J \mathbf{u}_{Jk} \mathrm{d}\Omega - \int_{\Omega} \nabla N_I \cdot \nu \nabla N_I \mathbf{u}_{Jk} \mathrm{d}\Omega \tag{33}$$

if the viscosity is constant it can be taken out of the integrals. Taking into account that the term $\int_{\Omega} \nabla N_I \cdot \nabla N_J d\Omega$ can be computed as the trace of $\mathbf{L}^d_{IJ}$, that is

$$\mathbf{L}_{IJ} := \mathrm{Tr}\left(\mathbf{L}^d_{IJ}\right) := \sum_{k} \left(\mathbf{L}^d_{IJ}\right)_k \tag{34}$$

we can thus obtain

$$\mathbf{B}_I := \nu \sum_{J \neq I} \mathbf{L}_{IJ} \left(\mathbf{u}_J - \mathbf{u}_I\right) \tag{35}$$

Unfortunately, if the viscosity varies in space it can not be taken outside of the integrals, thus invalidating this derivation. For the viscous case we hence need to introduce an approximation. In our work we follow [17] and approximate the viscosity with its nodal value, to obtain

$$\mathbf{B}_I := \nu_{II} \sum_{J \neq I} \mathbf{L}_{IJ} \left(\mathbf{u}_J - \mathbf{u}_I\right) \tag{36}$$

alternative techniques, which approximate $\nu = \nu_{IJ} = \frac{\nu_I + \nu_J}{2}$ to give

$$\mathbf{B}_I := \sum_{J \neq I} \nu_{IJ} \mathbf{L}_{IJ} \left(\mathbf{u}_J - \mathbf{u}_I\right) \tag{37}$$

are also possible with the current approach, since the calculation structure we use automatically ensures that the viscous force is zero in case of constant velocity field.

3.3. Convective term

The convective term is non-linear, and approximations are needed to allow expressing it in terms of constant operators. We follow ([17]) and use the approximation

$$\sum_J \int_\Omega N_I \mathbf{u} \cdot \nabla N_J \mathbf{u}_J \mathrm{d}\Omega \approx \mathbf{u}_I \cdot \sum_J \int_\Omega N_I \nabla N_J \mathbf{u}_J \mathrm{d}\Omega \tag{38}$$

which allows computing the convection term as

$$\mathbf{A}_I := \sum_{J \neq I} (\nabla_{IJ} \cdot \mathbf{u}_I)(\mathbf{u}_J - \mathbf{u}_I) \tag{39}$$

3.4. Split-OSS convection stabilization

Finally we need to describe the computation of the split-OSS stabilization. The contribution to node I of such stabilization is defined as

$$\mathbf{S}_I := \sum_J \int_\Omega \nabla N_I \mathbf{u} \otimes \mathbf{u} \cdot \nabla N_J \mathbf{u}_J - \int \sum_J \int_\Omega \mathbf{u} \cdot \nabla N_I N_J \xi = \mathbf{S}_I^{low} - \mathbf{S}_I^{high} \tag{40}$$

where ξ is the L^2 projection of the convective term onto the finite element mesh defined as

$$\xi_{Ik} := \mathbf{M}_{II}^{-1} \sum_{I \neq J} (\nabla_{IJ})(\mathbf{u}_{Jk} - \mathbf{u}_{Ik}) \tag{41}$$

The edge-based approximation of the stabilization is computed as

$$\mathbf{S}_{Ik}^{low} := \sum_{I \neq J} \left(\sum_{m=1}^{3} \sum_{l=1}^{3} \mathbf{u}_{Im} \mathbf{u}_{Il} \left(\mathbf{L}_{IJ}^{d} \right)_{ml} \right) (\mathbf{v}_{Jk} - \mathbf{u}_{Ik}) \tag{42}$$

and

$$\mathbf{S}_{Ik}^{high} := \sum_{I \neq J} ((\nabla_{IJ}) \cdot \mathbf{u}_I)(\mathbf{u}_J \cdot \xi_J - \mathbf{u}_I \cdot \xi_I) \tag{43}$$

3.5. Complete momentum residual calculation

Taking into account all of the above derivations, the complete residual can be computed by summing the different contributions as

$$\mathbf{R}_I(\mathbf{u}) := \mathbf{A}_I + \mathbf{B}_I + \mathbf{P}_I + \left(\mathbf{S}_I^{low} - \mathbf{S}_I^{high} \right) \tag{44}$$

3.6. Pressure calculation

The fractional-step algorithm requires the construction (and solution) of a Laplacian-like system with varying coefficients (depending on the stabilization that is locally required) as described in Equation (16). If, as in our case, a split-OSS stabilization is assumed, the stabilization term $\mathbf{S}_p$ assumes a form of the type $\int_\Omega (\tau \nabla q \nabla \mathbf{p} - \tau \nabla q \cdot \xi)$ where ξ is the L^2 projection of the discontinuous pressure gradient onto the finite element space. In order to use standard iterative solver technology a sparse matrix has to be constructed. In our implementation we allocate a scalar matrix with the same graph as the original edge matrix but also containing the diagonal terms. Within each solution step of the fluid-solver we will then fill such matrix by scaling the Laplacian coefficients of the original edge matrix by the appropriate stabilization coefficients τ. The approach described in [2, 14, 16] for the calculation of such matrix leads to a non-symmetric system when different values are used for the stabilization coefficient on the different nodes (as the stabilization theory requires). In the current work we tested this approach and the symmetrization approach described in [17]. Both approaches were found to work satisfactorily. We finally decided to use the symmetric form in order to be able to use the CG solver instead of the BiCGstab solver so as to enjoy a lower computational cost. As for the edge-graph, we chose not to use the symmetry of the graph of the matrix in order to ease the parallelization of the SpMV operation within the iterative solvers.

In implementation terms we need to solve a system of the type $\mathbf{H}\mathbf{p} = \mathbf{f}$ for the pressure. The terms which correspond to $I \neq J$ are defined as

$$\mathbf{H}_{IJ} := \left(\tau_{IJ} + \frac{\delta t}{2}\right) \mathrm{Tr}\, \mathbf{L}_{IJ} \tag{45}$$

with $\tau_{IJ} = \frac{\tau_I + \tau_J}{2}$ and $\tau_I = \frac{1}{\left(1/\delta t + \nu/h_I^2 + \|\mathbf{u}\|/h_I\right)}$. The term h_I should be understood here as a measure of the element size which we take as $h_I := \sqrt[3]{\mathbf{M}_{II}}$. Diagonal terms are defined as

$$\mathbf{H}_{II} := -\sum_{J \neq I} \mathbf{H}_{IJ} \tag{46}$$

so to enforce discrete conservation properties ($\sum_J H_{IJ} = 0$). The right hand side $\mathbf{f}$ is then constructed as

$$\mathbf{f}_I := \sum_{J \neq I} \nabla_{IJ} \cdot (\hat{\mathbf{u}}_J - \hat{\mathbf{u}}_I) + \sum_{J \neq I} \tau_{IJ} \mathbf{G}_{IJ} \cdot (\xi_J - \xi_I) + \frac{\delta t}{2} \sum_{J \neq I} \mathbf{L}_{IJ} (\mathbf{p}_{nJ} - \mathbf{p}_{nI}) \tag{47}$$

where the pressure projection ξ_I is to be computed as

$$\xi_I := \mathbf{M}_{II}^{-1} \sum_{J \neq I} \nabla_{IJ} (\mathbf{p}_J - \mathbf{p}_I) \tag{48}$$

4. OpenCL Implementation

The first challenge in porting programs to OpenCL or CUDA [18] platform is the difference in programming model between traditional CPU programming and OpenCL. Modern GPUs are massively parallel devices, and may contain tens to hundreds of cores. Each of these cores is much weaker compared to the cores in multi-core CPUs, however the total computational power of modern GPUs outperforms even the most recent CPUs on the market. Unlocking this great computational power needs special programming which is in many ways different from serial (or even multi-core) programming in CPUs. Designed primarily for graphical operations, GPUs have (from the programmer's point of view) a much more complex architecture than CPUs. This complex architecture is essential for the execution model of the device which guarantees that the computational power scales up with the number of cores. Memory access is one of the most important aspects of programming on all platforms. Despite the familiar hardware design of PCs where there is only a single type of memory known as RAM, GPUs have several types of memory for different purposes. An effective software design for GPUs must consider using the appropriate memory type to harness full potential of the GPU. A complete description of different kinds of memories in GPUs is beyond the scope of this work and the reader can consult References [19, 20] for more details. OpenCL takes into account these architectural differences, and is designed for heterogeneous computing on massively parallel devices such as GPUs and other type of accelerators. OpenCL is the first (and only) open standard which can guarantee code portability on different hardware. Performance portability of the programs however, is an open question. We will deal with this issue later.

Following the experiences described in Reference [2], we ported the Navier-Stokes solver in Kratos [5] to OpenCL. The base solver leverages OpenMP technology to benefit from multi-core CPUs. This solver has been optimized over the years and is mature enough to solve real world engineering problems. It will be used as a reference when benchmarking the ported solver. Comparing the results with an optimized solver provides us with realistic results and deep insight on the real usefulness of porting available codes to GPU platforms. For a complex problem like Navier-Stokes on unstructured grids where the memory access is quite irregular, one may not expect a very high speedup, since the work to be performed is not sufficiently regular. This reasoning is in line with experiences from other researchers [21, 22].

By analysing the computational structure of the problem we can make the following observations. In each time-step the solver computes several quantities, such as momentum residuals. These computations require looping over all nodes I of the domain. For each node I, an inner loop is made over all the nodes J surrounding such node. Each pair of nodes IJ represents an *edge* of the Finite Element mesh and is stored in the CSR-like structure defined in Section 2. For each edge IJ a large number of local floating point operations is executed. As the operations done inside the outermost loops are independent, the loop is hence perfectly parallel and can be written as an OpenCL *kernel*. The kernel is scheduled for execution by the OpenCL runtime in parallel on all available cores in the GPU hardware. The inner loop, which is where all the local computations are done, is moved inside the kernel which will be called once for each node I in the domain. Since the number of neighbours around each node I may vary in an unstructured grid, the work to be done in such inner loop varies depending on the node. This leads to load imbalance. Fortunately, for the Navier-Stokes problem the inner loop is computationally intensive, and has a much better computation to memory access ratio compared to convection-diffusion problem. This aspect makes the Navier Stokes problem more sutable to achieve a good speedup using GPUs than the simpler convection-diffusion problem benchmarked in [2].

4.1. Solution of implicit pressure equation

The most important challenge in the porting process compared to [2] is the need to solve implicitly for the pressure once per time step. This implies solving a linear system of equations with a Laplacian matrix. Despite the possiblity to perform such computation on the CPU, so as to take advantage of advanced solvers and preconditioners in Kratos, this would imply transferring all the computed results back and forth to the CPU.

As the CPU-GPU data transfers are usually very slow compared to the high memory bandwidth of the GPU, this would slow down the computations. The only option left is to solve the system of equations directly on the GPU. This is more logical, as the data needed to solve the system of equation is generated and already resides in the GPU memory, and no transfer cost is to be needed. When it comes to OpenCL, there are few free / open source solvers available for sparse systems of linear equations. One of the best available options is the ViennaCL library [4]. The latest version of this library implements a complete set of BLAS functions in OpenCL, as well as various solvers and preconditioners for sparse linear system of equations in common formats. Recent versions of this library have the ability to incorporate the OpenCL data structures built already in the main program. This greatly eases merging the library with the previous codebase.

Unfortunately, after implementing an interface to ViennaCL and using it in the program, the performance of the program appeared to be quite poor. The execution of the program was carefully reviewed with profilers available in the GPU Software Development Kits (SDKs). Profiling made clear that the solution of the implicit pressure equation was taking almost all of the execution time. Several attempts were made to improve the performance of ViennaCL (with the help of the author of the library), unfortunately none of them were successful enough, for the size of problems of interest. Further investigations revealed that, for *small* matrix sizes (i.e. problems below 200,000 nodes), 80% of solution time was consumed in the SpMV kernel inside ViennaCL. This library uses a naïve implementation of the algorithm. The performance of SpMV kernels on graphics hardware has been the subject of many recent researches [23–33]. It has been shown that the naïve implementation of SpMV kernel is quite ineffective on such platforms [23]. It is worth noting that the performance of SpMV kernels is in general highly dependent on the non-zero structure of the system matrix, which defines the memory access pattern of the kernel.

Due to small number of non-zeros in each row of the system matrix in this problem, the vectorization algorithm devised in [23] was found to be inadequate. The final conclusion was that we needed to implement a flexible optimizable solver based on OpenCL. The essential idea of our proposal is to attempt performing vectorization over more than one line of the CSR matrix, thus allowing to take advantage of the vectorization approach even for matrices that have, as in our case, only relatively few nonzeros per row.

4.2. Implementation of a solver for linear system of equations with run-time optimization

Implementing an iterative (CG or BiCGStab) solver for linear system of equations on GPUs is a rather tedious work and requires, as a very minimum, a careful implementation of the SpMV kernel and vector dot product. Since most of the solution time was employed in the SpMV kernel, we focused our attention on the optimization of such procedure. As mentioned earlier, performance of this kernel is highly dependent on the non-zero structure of the main matrix. With almost no a priori information on the matrix structure, we tried to design an algorithm which tunes, using some parameters, the parallelization scheme of the SpMV kernel. The parameters considered here were the size of OpenCL *workgroup* and the number of rows to be handled by each workgroup. The main idea was to develope an algorithm for SpMV which could use an arbitrary sized workgroup, where each workgroup does multiplication for an specified number of rows. The algorithm devised in [23] can be seen as a particular case of this algorithm. The following excerpt shows the proposed algorithm in pseudocode.

```
// WORKGROUP_SIZE_BITS and ROWS_PER_WORKGROUP_BITS are to be specified
// on the command-line of the OpenCL compiler

#define WORKGROUP_SIZE (1 << WORKGROUP_SIZE_BITS)
#define ROWS_PER_WORKGROUP (1 << ROWS_PER_WORKGROUP_BITS)

#define LOCAL_WORKGROUP_SIZE_BITS \
  (WORKGROUP_SIZE_BITS - ROWS_PER_WORKGROUP_BITS)
```

```
#define LOCAL_WORKGROUP_SIZE (1 << LOCAL_WORKGROUP_SIZE_BITS)

__kernel void
  __attribute__((reqd_work_group_size(WORKGROUP_SIZE, 1, 1)))
  SpMV(
  __global unsigned int const *RowPointers,
  __global unsigned int const *ColumnIndices,
  __global double const *Values,
  __global double const *X,
  __global double *Y,
  unsigned int LinesNo,
  __local double *Buffer)
{
  gid = get_group_id(0);
  tid = get_local_id(0);

  lgid = tid >> LOCAL_WORKGROUP_SIZE_BITS;
  ltid = tid & (LOCAL_WORKGROUP_SIZE - 1);

  Row = (gid << ROWS_PER_WORKGROUP_BITS) + lgid;

  if (Row < LinesNo)
  {
    Buffer[tid] = 0.00;

    Start = RowPointers[Row];
    End = RowPointers[Row + 1];

    // Actual multiplication
    for (i = Start + ltid; i < End; i += LOCAL_WORKGROUP_SIZE)
    {
      Buffer[tid] += Values[i] * X[ColumnIndices[i]];
    }
  }

  // Parallel reduction of the results in Buffer[tid]
  :
  :

  // Store the final result in Y
  if (ltid == 0)
  {
    Y[Row] = Buffer[tid];
  }
}
```

The code makes use of C pre-processor macros, which helps to keep a single and clear code base while eliminating the need for producing the kernel code dynamically at runtime. The parameters, `WORKGROUP_SIZE_BITS` and `ROWS_PER_WORKGROUP_BITS` prescribe the size of workgroup and the number of rows to be handled by each workgroup (assumed to be both powers of two). They are specified as pre-processor macros on the command-line of the OpenCL compiler. As kernels are compiled at runtime, the OpenCL compiler can substitute the macros with the given values and produce an efficient code. With the size of workgroup being known in terms of given parameters, this information is given to the OpenCL compiler via `__attribute__((reqd_work_group_size(WORKGROUP_SIZE, 1, 1))`, which further helps the compiler optimize the code and, for example, remove the unnecessary `barrier()`'s in the actual code. Restricting the aforementioned parameters to powers of 2, helps use arithmetic shift operations, which are much faster than ordinary multiplication and division. `RowPointers`, `ColumnIndices` and `Values` arrays are the usual components defining the CSR matrix structure of `A`, while `X` and `Y` are the appropriate vectors in `Y = AX`. `Buffer` is a `__local` array, shared among the members of a workgroup, containing partial results of the multiplication. The multiplications are ordered so that memory accesses remain as coalesced as possible, contributing to the effectiveness of the algorithm.

This new kernel showed an average speed up of 5 compared to the naïve version on the test matrices of [23]. After implementation of a parallel dot product subroutine with similar tunable parameters, we had all the building blocks for a CG or BiCGStab solver. We chose to implement the three-term recurrence variant of CG (algorithm 6.18 of [34]), which seemed like a good candidate for implementation on GPUs. For maximum performance, kernels were fused where possible, reducing the kernel launch overhead. The implemented solver was, as expected, on average 6 times faster than ViennaCL's solver on the symmetric, positive definite matrices of [23].

As previously stated, the parameters in the SpMV and dot product kernels change the parallelization scheme of their respective kernels. Hence to obtain a good performance we needed to determine the best possible set of parameters for a given problem. A given set of parameters can not be expected to be optimal on different hardware. This issue, known as *performance portability*, is very important and the subject of many recent researches [35–39]. Futhermore, the parameters in the SpMV and vector dot product kernels, depend strongly on the non-zero pattern of the system matrix.

Though it may be possible defining a model for each class of GPU hardware and predict the best set of parameters (like what is done in [30]), we prefer to look for optimal parameters dynamically and decided to employ a simple brute-force approach with precise timing to determine optimal parameters. The rationale is that typical problems need several thousands of time-steps and at each step the implicit pressure equation needs to be solved. Each step will thus require calling the SpMV and dot product kernels a few hundred times. Optimizing the kernel parameters at the beginning of the calculations (and thus for the exact problem size to be handled) by a brute force approach for a limited range of eligible parameters requires only a few hundreds calls to the target functions and will thus provide only a negligible overhead with respect to the total solution cost. This approach is not only simpler but also very accurate and adapts automatically to new hardware and to different problem sizes. Of course, if the set of optimal parameters is known a priori (for example from a previous run), the information can be supplied to the program, eliminating the need for the optimization process.

5. Performance analysis and numerical benchmarks

In this section we present two numerical benchmarks, and compare the performance of developed OpenCL code with an existing, highly-optimized OpenMP solver. For a broader investigation, we report all the results on some platforms with different configurations. We chose platforms with different relative ages and computational power, so that the results are not specific to a particular hardware or software. Where possible, the latest available OpenCL SDKs have been used with appropriate drivers. The specifications of the platforms can be found in Table 1. As mentioned earlier, memory bandwidth is the most important factor in SpMV operation, which consumes a considerable part of the solution. To have a better understanding, Table 2 lists the results of ‘STREAM’ memory benchmark [40] for the hosts, and a similar OpenCL-based, self-developed benchmark for devices. The values reported for hosts are measured for a single thread (i.e. without OpenMP support) with the original untuned code, so they are easily comparable with other platforms. It is noteworthy that the values reported in Table 2 represent the achieved memory bandwidth on the platforms, not the

Table 1: Platforms description

Platform	CPU	GPU	OpenCL SDK	GCC ver.
1	AMD Phenom Quad core 9950	NVIDIA GTX 280	NVIDIA CUDA 4.1	4.4
2	Intel Core i7 2700	AMD Radeon HD 6970	AMD APP SDK 2.7	4.7
3	Intel Core2 Quad 8300	NVIDIA GTX 550 Ti	NVIDIA CUDA 5.0	4.7
4	Intel Core i7 920	AMD Radeon HD 7970	AMD APP SDK 2.6	4.6
5	Intel Core i7 3820	NVIDIA GTX 580	NVIDIA CUDA 5.0	4.6

Table 2: Memory bandwidth for host and device on the benchmark platforms

Platform	Size / Type of RAM		RAM bandwidth			
	Host	Device	Host (Add)	Host (Copy)	Device (Add)	Device (Copy)
1	4 GB DDR2	1 GB GDDR5	5.3 GB/s	4.9 GB/s	36.5 GB/s	121.0 GB/s
2	8 GB DDR3	2 GB GDDR5	14.2 GB/s	13.1 GB/s	65.3 GB/s	134.0 GB/s
3	4 GB DDR2	1.5 GB GDDR3	5.2 GB/s	4.6 GB/s	13.1 GB/s	26.0 GB/s
4	12 GB DDR3	1 GB GDDR5	12.3 GB/s	11.7 GB/s	66.7 GB/s	180.8 GB/s
5	8 GB DDR3	3 GB GDDR5	15.4 GB/s	17.0 GB/s	85.6 GB/s	166.5 GB/s

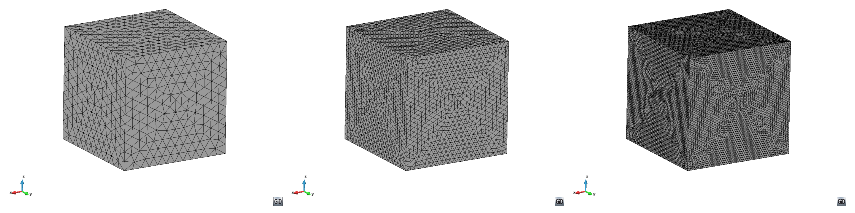
peak theoretical ones, so they constitute a good basis for comparison.

The first benchmark is the standard 'Cavity Flow' problem for which a reference solution can be found in [41]. Here we perform a refinement test and measure the performance on different test machines. The second benchmark is the classical 'Ahmed's body' test, for which we will use the same geometry as in [2]. Once again we measure the performance on different hardware.

'Cavity Flow' problem

This test consists of a unit cube (aligned with the xyz axes) for which the velocity is fixed to zero on all of the edges and on 5 out of the 6 faces. The top face is fixed to a velocity of 1 in the positive x direction. Pressure is fixed to zero on one of the y edges of the bottom surface. The simulation starts with the whole volume still and runs during sufficient time for the whole domain to be put into motion.

The definition of the test is completed by the prescription of the viscosity. If the side of the cube (which has unit length) is taken as a characteristic length, the Re (Reynolds Number) of the problem is $Re = 1/\nu$. In our case we considered the case of $Re = 100.0$ for the time in the range of $0.0s$ to $0.1s$. Three different levels of



(a) Coarse mesh (b) Medium mesh (c) Fine mesh

Figure 1: Meshes used in the benchmarking process for the ‘Cavity Flow’ example

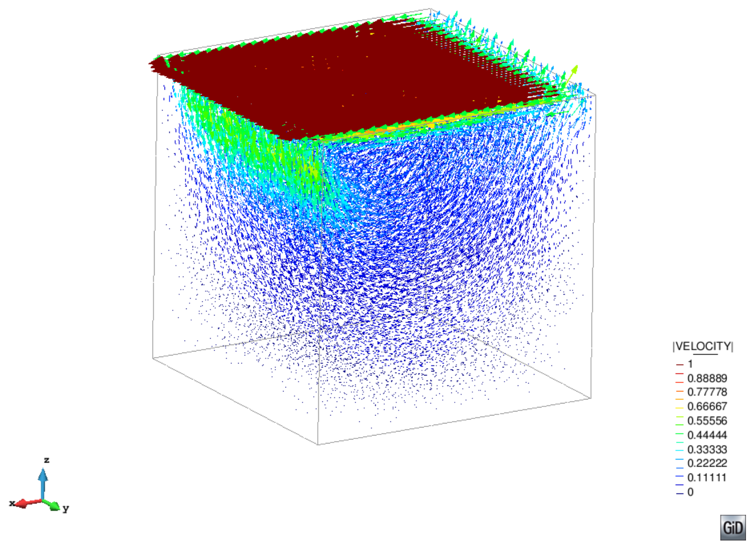


Figure 2: View of the velocity field computed on the medium mesh

Table 3: Mesh data for different levels of refinement in ‘Cavity Flow’ problem

Refinement level	No. of nodes	No. of elements	No. of edges
0	3347	16579	41710
1	24202	132632	321106
2	184755	1061056	2521380

Table 4: Run times for ‘Cavity flow’ problem

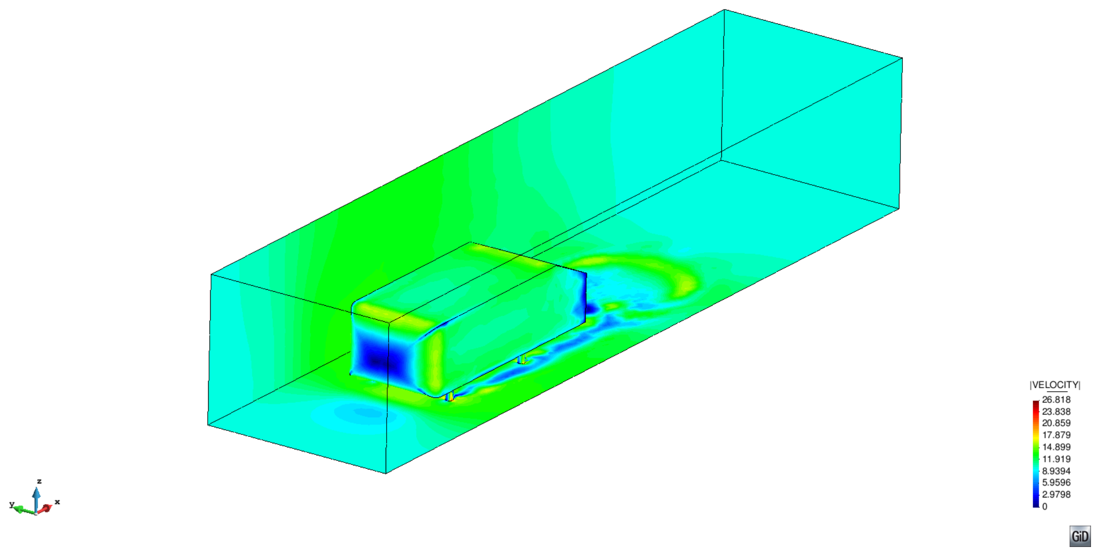
Platform	0-level refinement			1-level refinement			2-level refinement		
	OpenMP	OpenCL	Ratio	OpenMP	OpenCL	Ratio	OpenMP	OpenCL	Ratio
1	8.2s	15.9s	0.52	332.6s	201.6s	1.65	22030.8s	5366.0s	4.11
2	3.1s	15.7s	0.20	108.0s	117.1s	0.92	11536.9s	2941.5s	3.92
3	9.9s	13.3s	0.74	540.7s	249.1s	2.17	32729.1s	8980.7s	3.64
4	5.3s	64.3s	0.08	159.9s	409.6s	0.39	15251.1s	3970.2s	3.84
5	3.0s	9.6s	0.31	104.6s	106.2s	0.99	8711.0s	2770.9s	3.14

refinement are considered and the corresponding mesh data are shown in Table 3. The three meshes are shown in Figure 1. Figure 2 shows a view of the velocity field at the end of the simulation. Table 4 shows the run times obtained for the problem on GPU (OpenCL) along with reference CPU implementation (OpenMP) on CPU. The results shown in the table suggest that the OpenCL implementation takes the lead for sufficiently large problem sizes. For the finest mesh a speedup above 3 is obtained on all of the benchmarked architectures, which proves that the goal of performance portability is reached. For smaller problem sizes the CPU still has the lead, probably thanks to optimal use of the cache. It is also interesting to observe how recent Core i7 3820 almost doubles the performance of the Core i7 920 for this benchmark.

We remark that for this benchmark we verified heuristically that the results appeared to be rather independent on the node renumbering scheme used. This was an unexpected result, probably related to the data encapsulation strategy used.

In order to provide a deeper characterization of the performance characteristics of the solver, we report in Table 5 a more detailed benchmark of the run times measured on the GPU for the finest mesh case. The columns labelled “CPU→GPU” and “GPU→CPU” reflect transfer times between the database on the CPU and the data

Figure 3: View of the velocity field at the surface for the ‘Ahmed’s body’ problem



structure on the GPU. The columns labelled “Step 1” to “Step 3” report the timings of the three steps which compose the fractional step algorithm. The results reported in the table refer to the first time step of the solution.

For the benchmark case, as expected, the number of iterations of the linear solver is very similar between the CPU and the GPU (identical for the first steps). Small differences arise during the calculations due to the differences in precision between the CPU and the GPU. Although the exact number of iterations varies from time step to

Table 5: Benchmarking data for the finest mesh case on GPU - timings of “Solve” function per solution step

Platform	Total Step time	CPU→GPU	Step 1	Step 2	Step 3	GPU→CPU
1	0.970s	0.039s (4.0%)	0.001 (0.1%)	0.888s (92%)	5.3e-5s (0.005%)	0.042s (4.3%)
2	0.704s	0.028s (4.0%)	0.019 (2.7%)	0.632s (90%)	1.7e-5s (0.002%)	0.024s (3.4%)
3	1.673s	0.047s (2.8%)	0.0014 (0.08%)	1.551s (92%)	5.9e-5s (0.003%)	0.074s (4.6%)
4	0.787s	0.026s (3.3%)	7.8e-5 (0.01%)	0.730s (93%)	1.4e-5s (0.002%)	0.029s (3.7%)
5	0.469s	0.015s (3.2%)	6.8e-4 (0.01%)	0.632s (92%)	3.0e-5s (0.006%)	0.023s (4.7%)

Table 6: Benchmarking data for the finest mesh case on the CPU - timings of "Solve" function per solution step

Platform	Total step time	Step 1	Step 2	Step 3
1	3.26s	0.381s (11.7%)	2.81 (86.3%)	0.066s (2.0%)
2	1.71s	0.171s (10.0%)	1.49 (87.2%)	0.047s (2.7%)
3	4.84s	0.495s (10.2%)	4.25 (88.0%)	0.099s (2.0%)
4	2.24s	0.259s (11.5%)	1.93 (86.4%)	0.088s (2.0%)
5	1.13s	0.131s (11.6%)	0.97 (85.7%)	0.029s (2.6%)

time step, it averages to around 80 for the first level of refinement, 170 for the case with one level of refinement and 380 for the finest case. Diagonal scaling is employed both on the CPU and on the GPU. The number of steps needed for the total solution phase is approximately 500 for the first level of refinement, 1500 for the second and 5400 at the maximum refinement level. Although the number of steps is not exactly identical between host and device, the differences are minimal (5437 steps on the GPU vs 5450 steps on the CPU).

If we consider for reference the corresponding results on the CPU, only three steps can be identified and are reported in Table 6. For a fair comparison the reader should consider that all of the data managment is included in "Step 1" and "Step 3" in the CPU version.

A somewhat unexpected result is that the overall speedup of the OpenCL version vs. the OpenMP version is better when the overall run is considered than when a single step is benchmarked. We verified the percentage of time spent in the different algorithmic steps is very consistent when the benchmark is performed on a single step and as an average of all of the calculation steps. This leads to the conclusion that the empirical observation of the OpenCL driver performance fluctuation during the run is true.

'Ahmed's body' problem

The Ahmed's body problem is a benchmark used in the automotive industry to evaluate the performance of different solvers. In this paper we use it purely as a benchmark of numerical efficiency and report the run times obtained for a given computa-

Table 7: Run times for the ‘Ahmed’s body’ problem

Platform	OpenMP	OpenCL	Ratio
1	45854.3s	6221.4s	7.37
2	25343.8s	3812.5s	6.65
3	67755.6s	9636.6s	7.03
4	26378.8s	4766.5s	5.53
5	18079.7s	3707.3s	4.88

tional mesh using either OpenMP or OpenCL. The results for the different platforms considered are shown in Table 7. The mesh used in the benchmark included around 80,000 nodes and 400,000 elements. A view of the solution is shown in Figure 3. The Reynolds number of this test, approximately 4×10^6, is much higher than in the cavity example (the mesh used is not sufficiently fine to deliver accurate results, but enough for benchmarking purposes). A very good speedup, above 5 in almost all cases, was measured on all of the architectures benchmarked. This improvement is most likely related to a different relative importance between the pressure solution step and the momentum solution step with respect to the cavity problem.

6. Conclusions

The present work discusses an unstructured Navier-Stokes solver which runs natively on GPUs. The solver is benchmarked considering two relevant test cases and various meshes, showing a satisfactory speedup with respect to a reference, highly optimized OpenMP solver. Code portability is ensured by the use of the OpenCL programming language, while performance portability is addressed via an automatic optimization step, performed at the beginning of the computations. Given the relatively large sparsity of the pressure system, standard SpMV vectorization techniques were fount to be ineffective. A novel multi-line vectorization approach is developed to provide competitive performance.

Acknowledgments

The second author would like to thank International Center for Numerical Methods in Engineering (CIMNE) for kindly supporting this work. The work is also supported under the auspices of the “eDams” project of the MEC (BIA2010-21350-C03-00) and of the ERC Advanced Grant “RealTime” project AdG_2009325. The authors also wish to thank Dr. Prashanth Nadukandi for his help in improving the level of english in the text. The help and patience of Ing. Pablo Becker in running some of the benchmarks and in proofreading the text is gratefully acknowledged.

References

[1] Storti M, Paz R, Dalcín L, Costarelli S, Idelsohn S. FFT Preconditioning Technique for the Solution of Incompressible Flow on GPU’s. *Computers and Fluids*, under review.

[2] Mossaiby F, Rossi R, Dadvand P, Idelsohn S. OpenCL-based implementation of an unstructured edge-based Finite Element convection-diffusion solver on graphics hardware. *International Journal for Numerical Methods in Engineering* 2012. **89**(13): 1635–1651.

[3] Corrigan A, Camelli F, Löhner R, Wallin J. Running unstructured grid-based CFD solvers on modern graphics hardware. *International Journal for Numerical Methods in Fluids* 2011. **66**(2): 221–229.

[4] ViennaCL project homepage. http://viennacl.sourceforge.net [12 August 2012].

[5] Dadvand P, Rossi R, Oñate E. An object-oriented environment for developing finite element codes for multi-disciplinary applications. *Archives of Computational Methods in Engineering* 2010. **17**(3):253–297.

[6] Kratos wiki page. http://kratos-wiki.cimne.upc.edu [26 October 2012].

[7] Ryzhakov P. Lagrangian FE methods for coupled problems in fluid mechanics. *PhD thesis, Universitat Politécnica de Catalunya* 2010.

[8] Ryzhakov P, Rossi R, Oñate, E. An algorithm for the simulation of thermally coupled low speed flow problems. *International Journal for Numerical Methods in Fluids* 2011. **70**(1):1–19.

[9] Donea J, Huerta A. *Finite element method for flow problems*. Wiley edition, New York, 2003.

[10] Codina R. Stabilized finite element approximation of transient incompressible flows using orthogonal subscales. *Computer Methods in Applied Mechanics and Engineering* 2002. **191**(39–40):4295–4321.

[11] Oñate E, Valls E, Garcia J. Modeling incompressible flows at low and high Reynolds numbers via a finite calculus-finite element approach. *Journal of Computational Physics Archive* 2007. **224**(1):332–351.

[12] Nadukandi P, Oñate E, Garcia E. A high resolution Petrov-Galerkin method for the 1D convection-diffusion-reaction problem. *Computer Methods in Applied Mechanics and Engineering* 2010. **199**(9–12):525–546.

[13] May M, Rossi R, Oñate E. Implementation of a General Algorithm for Incompressible and Compressible Flows within the Multi-Physics code KRATOS and Preparation of Fluid-Structure Coupling. *Technical Report*, CIMNE, Barcelona, 2008.

[14] Rossi R, Larese A, Dadvand P, Oñate, E. An efficient edge-based level set finite element method for free surface flow problems. *International Journal for Numerical Methods in Fluids* 2012. DOI: 10.1002/fld.3680.

[15] Limache A, Idelsohn S, Rossi R, Oñate, E. The violation of objectivity in Laplace formulations of the Navier-Stokes equations. *International Journal for Numerical Methods in Fluids* 2007. **54**(6–8):639–664.

[16] Codina R, Folch A. A stabilized finite element predictor-corrector scheme for the incompressible Navier-Stokes equations using a nodal based implementation. *International Journal for Numerical Methods in Fluids* 2004. **44**(5):483–503.

[17] Soto O, Löhner R, Cebral J, Camelli F. Stabilized edge-based implicit incompressible flow formulation. *Computer Methods in Applied Mechanics and Engineering* 2004. **193**(23–26): 2139–2154.

[18] NVIDIA CUDA home page. http://www.nvidia.com/object/cuda_home_new.html [15 December 2010].

[19] AMD Accelerated Parallel Processing OpenCL Programming Guide. Available from http://developer.amd.com/gpu/ATIStreamSDK/documentation/Pages/default.aspx [17 December 2010].

[20] OpenCL Programming Guide for the CUDA Architecture. Available from http://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Programming_Guide.pc [25 August 2012].

[21] Lee VW, Kim C, Chhugani J, Deisher M, Kim D, Nguyen AD, Satish N, Smelyanskiy M, Chennupaty S, Hammarlund P, Singhal R, Dubey P. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture*. ACM: New York, NY, USA, 2010; 451–460.

[22] Vuduc RW, Chandramowlishwaran A, Choi JW, Guney M, Shringarpure A. On the limits of GPU acceleration. In *HotPar'10: Proceedings of the 2nd USENIX conference on Hot topics in parallelism*. USENIX Association: Berkeley, CA, USA, 2010.

[23] Bell N, Garland M. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM: New York, NY, USA, 2009; 1–11.

[24] Dehnavi MM, Fernandez D, Giannacopoulos D. Finite element sparse matrix vector multiplication on GPUs. *IEEE Transaction on Magnetics* 2010. **46**(8): 2982–2985.

[25] Grewe D, Lokhmotov A. Automatically Generating and Tuning GPU Code for Sparse Matrix-Vector Multiplication from a High-Level Representation. In *GPGPU-4: Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. ACM, NY, USA, 2011.

[26] Su BY, Keutzer K. clSpMV: A Cross-Platform OpenCL SpMV Framework on GPUs. In *ICS'12: Proceedings of the 26th ACM international conference on Supercomputing*. ACM, NY, USA, 2012.

[27] Wu T, Wang B, Shan Y, Yan F, Wang Y, Xu N. Efficient PageRank and SpMV Computation on AMD GPUs. In *ICPP'10: Proceedings of the 39th International Conference on Parallel Processing*. IEEE Computer Society, Washington, DC, USA, 2010.

[28] El Zein AH, Rendell AP. Generating optimal CUDA sparse matrixvector product implementations for evolving GPU hardware. *Concurrency and Computation: Practice and Experience* 2012. **24**(1):3–13.

[29] Vázquez F, Ortega G, Fernández JJ, Garzón EM. Improving the performance of the sparse matrix vector product with GPUs. In *CIT'10: Proceedings of the 10th IEEE International Conference on Computer and Information Technology*. IEEE Computer Society, Washington, DC, USA, 2010.

[30] Choi JW, Singh A, Vuduc RW. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA, 2010; 115–126.

[31] Williams S, Oliker L, Vuduc R, Shalf J, Yelick K, Demmel J. Optimization of sparse matrixvector multiplication on emerging multicore platforms. *Parallel Computing* 2010. **35**(3):178–194.

[32] Baskaran MM, Bordawekar R. Optimizing Sparse Matrix-Vector Multiplication on GPUs. *IBM Reserach Report, RC24704 (W0812-047)* 2009.

[33] Tanabe N, Ogawa Y, Takata M, Joe K. Scaleable Sparse Matrix-Vector Multiplication with Functional Memory and GPUs. In *PDP'11: Proceedings of the 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*. IEEE Computer Society, Washington, DC, USA, 2011.

[34] Saad Y. *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics, 2003.

[35] Rul S, Vandierendonck H, D'Haene J, De Bosschere K. An Experimental Study on Performance Portability of OpenCL Kernels. In *SAAHPC'10: Proceedings of 2010 Symposium on Application Accelerators in High-Performance Computing*. Knoxville, TN, USA.

[36] Pennycook SJ, Hammond SD, Wright SA, Herdman JA, Miller I, Jarvis SA. An investigation of the performance portability of OpenCL. *Journal of Parallel and Distributed Computing* 2012. DOI: 10.1016/j.jpdc.2012.07.005.

[37] Du P, Weber R, Luszczek P, Tomov S, Peterson G, Dongarra JJ. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing* 2012. **38**:391–407.

[38] Bosilca G, Bouteiller A, Herault T, Lemarinier P, Saengpatsa NO, Tomov S, Dongarra JJ. Performance Portability of a GPU Enabled Factorization with the DAGuE Framework. In *CLUSTER'11: Proceedings of the 2011 IEEE International Conference on Cluster Computing*. IEEE Computer Society, Washington, DC, USA, 2011.

[39] Kessler C, Dastgeer U, Thibault S, Namyst R, Richards A, Dolinsky U, Benkner S, Traff JL, Pllana S. Programmability and performance portability aspects of heterogeneous multi-/manycore systems. In *Design, Automation Test in Europe Conference Exhibition (DATE)* 2012. Dresden, Germany.

[40] McCalpin JD, Memory Bandwidth and Machine Balance in Current High Performance Computers, *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* Dec. 1995. 19–25.

[41] Ghia U, Ghia KN, Shin, CT. High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method. *Journal of Computational Physics* 1982. **48**(3):387–411.