An optimized GPU implementation of a 2D free surface simulation model on unstructured meshes

A. Lacasta*, M. Morales-Hernandez*, J. Murillo*, P. Garcia-Navarro*

Abstract
This work is related with the implementation of a finite volume method to solve the 2D shallow water equations on Graphic Processing Units (GPU). The strategy is fully oriented to work efficiently with unstructured meshes which are widely used in many fields of Engineering. Due to the design of the GPU cards, structured meshes are better suited to work with than unstructured meshes. In order to overcome this situation, some strategies are proposed and analyzed in terms of computational gain, by means of introducing certain ordering on the unstructured meshes. The necessity of performing the simulations using unstructured instead of structured meshes is also justified by means of some test cases with analytic solution.

Keywords: GPU, Finite volume methods, Unsteady flow, Unstructured meshes, Dry/wet boundaries, CUDA, High Performance Computing

1. Introduction
Physically based simulations of complex systems usually require large computational facilities to be completed in a reasonable time. Moreover when the simulated phenomenon is unsteady and based on a dynamical estimation of the updating time step, the computational performance is an important topic to be taken into account. One of the most widespread strategies to reduce the computational cost is the use of parallel techniques, involving a suitable number of processors. Since CPU frequencies seem to be reaching their maximum capacity [1], nowadays Many-Core parallel techniques appear to be an interesting option.

*LIFTEC-EINA, CSIC-Universidad Zaragoza
Email addresses: alacasta@unizar.es (A. Lacasta), mmorales@unizar.es (M. Morales-Hernandez), javier.murillo@unizar.es (J. Murillo), pigar@unizar.es (P. Garcia-Navarro)
In recent years, Graphic Processing Unit (GPU) has been used to accelerate the calculations because of its inherent vector-oriented designing. This paradigm is known as General-Purpose Computing on Graphics Processing Unit (GPGPU) and it is widely used for a very large range of applications in CFD such as [2],[3],[4] and [5] as well as other environmental applications such [6]. In the present work, special attention is paid to the application of this GPUs to unsteady flows of interest in hydraulics. Shallow Water models in particular are widely used to simulate surface geophysical flows. These situations usually involve large size domains and long time scales. Practical applications require a compromise between spatial accuracy and computational efficiency. In order to achieve the necessary spatial resolution, rather fine grids become necessary in many cases requiring more data storage, increasing proportionally the number of operations and reducing the allowable time step size for explicit calculations. When, at the same time, a reasonable computational time is desired, the use of GPU codes is one of the options for computing large space and temporal domain problems.

The idea of accelerating the calculations in unsteady hydraulic simulation using multiple CPU was recently reported in [7],[8] or [9] as well as using GPU in [10][11][12][5]. Although a very good compromise between number of CPU used and performance is offered by the former option, the cost of using multiple CPU is significant due to the hardware investment and associated use. Alternatively, the GPU technology offers the performance of smaller clusters with less disbursement [13]. The main difficulty, and apparent drawback, when porting codes from CPU to GPU, is the cell order required by the GPU to process data efficiently. This drawback is not present when dealing with structured meshes due to the inherent order and a simple and efficient implementation is relatively easy to be obtained.

Despite the wide use of structured meshes, complex geometries for internal or external boundaries are problematic to be represented if not using unstructured meshes. Moreover, when dealing with topographic representation some recent works [14] have shown the benefit of using unstructured meshes in unsteady hydraulic simulations over irregular topography. The quality of the numerical results is sensitive to the grid resolution. Hence grid refinement is clearly an option to modify the whole resolution. In that sense, adaptive grid refinement, readily available when using triangular unstructured meshes [15], designed to follow local bed variations or irregular boundaries can be very useful. The present work is motivated by the implementation in GPU of a code able to perform unsteady hydraulic simulations on variable density triangular unstructured meshes.

The performance of GPU based calculations with Double Precision (double)
is lower than those that use Single Precision (float) [16], [5]. In the particular case of the 2D Shallow Water Equations with source terms [17], [18], the use of float is not always desirable. In fact, when simulating complex topography cases, wave propagation over dry beds represents a numerical challenge. The past experience with the dynamical stability control of such transient flows involving wet/dry fronts indicates that double precision is always required. All the performance analysis presented will deal with that kind of data.

In the first part of the text, the governing equations are outlined. They are followed by a description of the finite volume updating scheme used. Then, the most relevant general aspects of the implementation in GPU are identified. The particular difficulties encountered when dealing with triangular unstructured meshes and some improvements to overcome them are detailed in the following section. Finally, they are applied to two test cases in order to prove their behaviour when using unstructured meshes.

2. Mathematical model/Governing equations

The two-dimensional SW equations, which represent depth averaged mass and momentum conservation, can be written as follows:

$$\frac{\partial U}{\partial t} + \nabla E = H$$

(1)

where

$$U = (h, q_x, q_y)^T$$

(2)

are the conserved variables with $h$ representing the water depth, $q_x = hu$, $q_y = hv$ and $u = (u, v)$ the depth averaged velocity vector along the $(x, y)$ coordinates respectively. The fluxes of these variables are $E=(F, G)$ given by:

$$F = \left(q_x, \frac{q_x^2}{h} + \frac{1}{2}gh^2, \frac{q_x q_y}{h}\right)^T, \quad G = \left(q_y, \frac{q_x q_y}{h}, \frac{q_y^2}{h} + \frac{1}{2}gh^2\right)^T$$

(3)

where $g$ is the acceleration due to the gravity.

The source terms of the system are the bed slope and the friction terms:

$$H = \left(0, -gh \frac{\partial z}{\partial x} - \frac{\tau_{b,x}}{\rho_w}, -gh \frac{\partial z}{\partial y} - \frac{\tau_{b,y}}{\rho_w}\right)^T$$

(4)
Where $\tau_{b,x}$ and $\tau_{b,y}$ are the components of the bed friction stress and $\rho_w$ is the water density. These friction losses in both $(x, y)$ axis are written in terms of the Manning’s roughness coefficient $n$:

$$\frac{\tau_{b,x}}{\rho_w} = gh \frac{n^2 u \sqrt{u^2 + v^2}}{h^{4/3}}, \quad \frac{\tau_{b,y}}{\rho_w} = gh \frac{n^2 v \sqrt{u^2 + v^2}}{h^{4/3}}$$

(5)

3. Numerical scheme

The numerical resolution of system (5) can be obtained by means of the first order upwind finite volume scheme. Integrating in a volume or grid cell $\Omega$ the numerical scheme can be expressed compactly:

$$\frac{\partial}{\partial t} \int_\Omega U \, d\Omega + \sum_{k=1}^{N_E} (\delta E - T)_k \cdot n_k l_k = 0$$

(6)

It is possible to define a Jacobian matrix $\tilde{J}_k$ of the normal flux at each edge as a result of the local linearization

$$\delta (E \cdot n)_k = \tilde{J}_k \delta U_k$$

(7)

and to diagonalize it in terms of matrices $\tilde{P}$ and $\tilde{\Lambda}$, formed by its eigenvalues $\tilde{\lambda}_m$ and eigenvectors $\tilde{e}_m$ respectively:

$$\tilde{P} = \begin{pmatrix} 1 & 0 & 1 \\ \tilde{u} - \tilde{c} n_x & -\tilde{c} n_y & \tilde{u} + \tilde{c} n_x \\ \tilde{v} - \tilde{c} n_y & \tilde{c} n_x & \tilde{v} + \tilde{c} n_y \end{pmatrix}, \quad \tilde{\Lambda} = \begin{pmatrix} \tilde{\lambda}_1 & 0 & 0 \\ 0 & \tilde{\lambda}_2 & 0 \\ 0 & 0 & \tilde{\lambda}_3 \end{pmatrix},$$

$$\tilde{e}_1 = \begin{pmatrix} 1 \\ \tilde{u} - \tilde{c} n_x \\ \tilde{v} - \tilde{c} n_y \end{pmatrix}, \quad \tilde{e}_2 = \begin{pmatrix} 0 \\ -\tilde{c} n_y \\ \tilde{c} n_x \end{pmatrix}, \quad \tilde{e}_3 = \begin{pmatrix} 1 \\ \tilde{u} + \tilde{c} n_x \\ \tilde{v} + \tilde{c} n_y \end{pmatrix},$$

(8)

where $\tilde{u} \cdot n = \tilde{u} n_x + \tilde{v} n_y$. The definition of the averaged variables is as follows [19]:

$$\tilde{u}_k = \frac{u_i \sqrt{h_i} + u_j \sqrt{h_j}}{\sqrt{h_i} + \sqrt{h_j}}, \quad \tilde{v}_k = \frac{v_i \sqrt{h_i} + v_j \sqrt{h_j}}{\sqrt{h_i} + \sqrt{h_j}}, \quad \tilde{c}_k = \sqrt{\frac{g (h_i + h_j)}{2}}$$

(9)
The difference across the edge $k$ can be projected onto the eigenvectors basis [18]:

$$
\delta U_k = U_j - U_i = \tilde{P}_k \tilde{A}_k
$$

(10)

where $\tilde{A}_k = (\tilde{\alpha}_1, \tilde{\alpha}_2, \tilde{\alpha}_3)^T_k$ contains the set of wave strengths. Following the same procedure with the source terms [18]

$$
(\tilde{Tn})_k = \tilde{P}_k \tilde{B}_k
$$

(11)

where $\tilde{B}_k = (\tilde{\beta}_1, \tilde{\beta}_2, \tilde{\beta}_3)^T_k$ contains the source strengths.

More information about the values of the wave and the source strengths as well as the entropy fix can be found in [18]. The contributions due to the fluxes and the source terms are combined in a compact expression

$$
(\tilde{\gamma}_m)_k = (\tilde{\lambda}_m \tilde{\alpha}_m - \tilde{\beta}_m)_k
$$

(12)

The 2D numerical upwind explicit scheme is formulated using only the contributions that arrive to the cell:

$$
\tilde{\gamma}_k = \frac{1}{2} \left[ 1 - \text{sign} (\tilde{\lambda}_k) \right] \tilde{\gamma}_k
$$

(13)

so that the finite volume approach for the updating of a single cell of area $\Omega_i$ is [20]:

$$
U^n_{i+1} = U^n_i - \frac{\Delta t}{\Omega_i} \sum_{k=1}^{N_E} \sum_{m=1}^{3} (\tilde{\gamma}_m - \tilde{\delta}_m l)_k^n
$$

(14)

Considering that in the explicit scheme (14) each $k$ cell edge is used to deliver information between a pair of neighbouring cells of different size, the time step size compatible with numerical stability is limited by

$$
\Delta t \leq \Delta t^\lambda \Delta t^\lambda = \frac{\min(\chi_i, \chi_j)}{\max_{m=1,2,3} |\lambda^m|}
$$

(15)

so that the following dimensionless quantity is defined

$$
\text{CFL} = \frac{\Delta t}{\Delta t^\lambda} \leq 1
$$

(16)

to control the numerical stability of the method.
Considering unstructured meshes, the distance $\chi_i$ in each cell $i$ must consider the volume of the cell and the length of the shared $k$ edges.

$$\chi_i = \frac{\Omega_i}{\max_{k=1,NE} l_k}$$  \hspace{1cm} (17)

For more details, see [21, 18].

4. Implementation on GPU

The GPU contains a large number of processors working all together applying the same operation over different elements. In order to program using this paradigm, NVIDIA has developed CUDA (Compute Unified Device Architecture) [22] that abstracts some aspects of the hardware, allowing programmers to develop general purpose programs efficiently.

There are two main points to understand the performance of GPUs by means of CUDA. The first is based on the way CUDA applications are developed. The basic element to be processed is called Thread. Threads are identified by labels ranging between 0 and blockDim. The group of Threads is called Block, and it contains a (recommended) 32 multiple number of Threads. Finally any group of Blocks is called Grid. The second aspect of interest is the hardware architecture. The minimum unit is the Streaming Processor (SP), where a single Thread is executed. A group of SP’s form the Streaming Multiprocessor (SM), typically with 32 SP’s. Finally, a GPU is composed by between 2 and 16 SM’s.

NVIDIA GPUs are generally formed by Streaming Multiprocessors (Typically 14-16 in the Tesla Series) which contain Streaming Processors (32 in the case of the Tesla) [23]. The GPU distributes the Blocks among the SMs. The SMs in turn assigns the Threads to the SP’s. All SP’s inside the multiprocessor perform the same operations at the same time, but each of them applies it to a different element inside a vector. The designing of the GPU is the reason of the recommendation of configure blockDim multiple of 32. The set of 32 threads processed in a SM is called warp.

CUDA for C is an extension of the C Standard programming language which includes the possibility of specifying three key abstractions in the execution: hierarchy of thread groups, shared memories and barrier synchronization. These are exposed to the programmer as a set of extensions that may be introduced in the
code. The most significant change in the syntax is the necessary parameters for
the execution of the functions. In particular, the functions include:

```c
// Function calling with blocksPerGrid blocks of size
blocksPerGrid, // Number of Blocks
threadsPerBlock, // Number of Threads within a block
0, // This parameter must be included if stream
// ID is specified. It is 0 by default.
streamId // Identifier of the Execution Stream

<<< // This is always included in CUDA kernels
functionName
<<< blocksPerGrid, // Number of Blocks
threadsPerBlock, // Number of Threads within a block
0, // This parameter must be included if stream
// ID is specified. It is 0 by default.
streamId // Identifier of the Execution Stream

>>>(parameters);

where blocksPerGrid is the number of blocks of size threadsPerBlock
launched in the stream streamId. Moreover, inside each function, each thread
may establish its identifier taking into account its own threadIdx.x and blockIdx.x
(in the case of 2D or 3D blocks, it is possible to get access to the .y, .z ID). This
provides each thread the way of establishing a unique access as, for instance, the
value of a vector V:

```c
float V[size];
int ID;
float velocity;
```

// Each thread may build its own unique Identifier as
ID = threadIdx.x+(blockIdx.x*blockDim.x);
// Each thread wants a position of vector v
v[ID]=velocity
```

More details of the standard can be found in [22].

The implementation of numerical models using GPU requires to take into ac-
count four important aspects:

- **Number of elements to be processed**: The number of Blocks and the num-
ber of Threads within each Block are parameters to be tuned by the pro-
gramer. They determine the maximum number of elements the GPU can
process (Blocks times Threads per Block). This number must be greater
than or equal to the number of elements to be processed.

- **Bottlenecks**: In order to process all the operations following the GPU paradigm,
special attention must be paid to the shared information between the process-
ing elements.
• **Data transfer reduction**: The communication between CPU and GPU is very slow. In general, all the operations must take place inside the GPU, otherwise the overhead caused by data transfers may generate such a cost that the global performance of the implementation can be lower than on CPU.

• **Floating Point data precision**: The GPU arithmetic performance is halved when using double precision data. Many applications require double precision because of numerical aspects but there exist many others for which simple precision is enough to develop the calculations. When single precision is acceptable, performance can be almost doubled on GPU.

The main loop of our implementation is shown in Listing 1 where the fundamental of the programming and the general aspect of the simulation code are shown.

Listing 1: Overview of the CUDA implementation.

```c
...  // Configuration of the parameters
Threads=256;
edgeBlocks=nEdge/Threads;
cellBlocks=nCell/Threads;
while(t<tmax){
    // Calculate the fluxes
    calculateEdgeFluxes<<<edgeBlocks,Threads,0,executionStream>>>(...);
    // Establish the minimum dt obtaining the ID of the
    // minimum dt
    (*^) Explained at Listing 3
    cublasIdamin(...,nEdge,vDt,1,id);
    // And assign it
    newDt<<<1,1,0,executionStream>>>(dt,vDt,id);
    // Update the elapsed time (in GPU)
    updateT<<<1,1,0,executionStream>>>(cuda_t,dt);
    // Retrieves the value of t to CPU
    cudaMemcpy(..., cudaMemcpyDeviceToHost);
    // Verify whether it is neccessary to dump data and
    // if so, process it.
    (*^) Detailed in Listing 4
    if(t<t_dump){
        // Copy of cell variables to a GPU
        // stored buffer
        cudaMemcpy(..., cudaMemcpyDeviceToDevice);
        // Establishing the barrier to ensure the copy of the
        // data to the buffer
```
cudaStreamSynchronize(copyStream);
// Copy the data to the CPU buffer
cudaMemcpyAsync(..., cudaMemcpyDeviceToHost,copyStream);
// Create a parallel CPU stream in order to control the disk-transfer
pThread_create( &diskThread, ...);
}
}

First, the number of elements of the Blocks is defined statically at the beginning attending to the criterion of occupancy of the streaming multiprocessors (See CUDA GPU Occupancy Calculator [23]). For our purposes, the amount of 96 Threads/Block and 256 Threads/Block has been identified in general as the best configuration for meshes with less than 100000 elements and more than 100000 elements respectively.

Bottlenecks appear when reduction patterns are present in the algorithms. The necessity of using reduction functions during the computation can be implemented using cublas. cublas library has high-level functions that work retrieving results to GPU or to CPU. When interested in using them without taking out the data from the GPU, thus must be specified as in Listing 2

Listing 2: Cublas configuration
```
cublasSetPointerMode_v2(handle, CUBLAS_POINTER_MODE_DEVICE)
```

stating that all results have to be returned to the GPU memory.

![Figure 1: Gathering minimum $\Delta t$ for all the domain](image)

The proposed model formulated as in (14) requires the computations of the minimum global $\Delta t$ that verifies the stability condition 16 when running along all the cell edges. It is achieved by means of the functions cublasIdamin. Figure 1 illustrates an example of the behaviour of this function obtaining the minimum among all of them. Details are shown in Listing 3.

Listing 3: Gathering $\Delta t$ operation
```
__global__ void newDt(double *dt, double *vDt, int *id){
```
```
257 :    // As cublasIdamin returns it value following
258 :    // 1-based indexing, we must substract 1
259 :    *dt=vDt[*id-1];
260 : }...
261 :    cublasIdamin(handle,*npared,vDt,1,id);
262 :    newDt<<<1,1>>>(dt,vDt,d_id);
263 :    ...
264 :    As the calculation is controlled by CPU, it is necessary to transfer the updated
265 :    \( t^{n+1} \). After \( \Delta t \) is calculated, the updating operation can be performed as in Listing
266 :    4 hence the updated value of \( t^{n+1} \) is transferred to the CPU.
267 :
268 :    Listing 4: Updating \( t \)
269 :    __global__ void updateT(double *dt,double *t){
270 :      *t=*t+*dt;
271 :    }
272 :    \( \Delta t \) is the only variable that has to be transferred every time-step due to the
273 :    control of the global calculation by the CPU. When data dumping is required, all
274 :    variables at the cells must be transferred to the CPU. This transfer is slow. Although
275 :    out of the scope of this work, this particular operation may be optimized making
276 :    use of concurrent execution and asynchronous data transfer.
277 : 5. Issues related to implementation on unstructured meshes
278 :    Structured meshes have proved suitable for GPU computations [11]. Indeed,
279 :    the GPUs have been developed optimizing the accesses to memory when this kind
280 :    of structures (grids) are used. The main question when calculating with structured
281 :    meshes to solve the Shallow Water equations is whether they provide good results
282 :    when the topography is complicated. It is more difficult to apply local refinement
283 :    of the mesh if necessary and, when solving river basins, the angulosity of the mesh
284 :    could lead to artificial viscosity near the shores. Moreover, when the simulation
285 :    requires to respect some kind of structure, the flexibility of the mesh allows fitting
286 :    to represent it properly [15]
287 :    The main advantage of structured meshes is the inherent order existing in their
288 :    creation. Neighbouring cells are usually near in memory. When the solver works
289 :    by edges, this point is very important in order to get the coalesced access to the
290 :    main memory in GPU [24].
291```
The way the calculations are made over unstructured meshes is not the same as that when using structured meshes. The work in [25] provides a way to make the calculations using the NSEW scheme, implying that the order of the cells is inherent to the manner of accessing the data. However, when using unstructured meshes, the most common way of performing the calculation is by solving the fluxes by edges rather than by cells because the number of operations is nearly halved. In the present work three main aspects have been identified as relevant to construct an optimal solver in GPU when working with unstructured meshes:

- **Cell ordering:** The way cells are ordered is important when accessing data from two neighboring cells.
- **AoS vs SoA:** The choice of using Array of Structures or Structure of Arrays may improve the performance of the solver
- **Edges ordering:** From the previous two points, when calculating by edges, the ordering of these shows relevant results.

In consequence, there is a limitation because of the variable wet-dry boundaries that appear in unsteady cases and may decrease the overall performance of the code. When ordering in the cells is not proximal, Warp divergence is probable. Warp divergence occurs when two threads have different evaluation in a control flow structure. Within a warp, the hardware is not capable of executing if and else statements and serialization of execution paths is required. More details about warp divergence can be found in [26]. This implies that when two elements to be processed must apply different operations by means of a flow control structure (i.e. if... else... such in the case of wet-dry frontier), first those threads that satisfy the condition will apply the first operation and after this, those Threads which enter in the else condition, will apply the other operations. This special issues may have important impact on the performance of the application. The order of the cells, will avoid partially this limitation and it is next discussed.

### 5.1. Cell Ordering

Cell ordering has relevant weight on the way the cells are connected. This is controlled by the connectivity matrix, defined as:

$$m_{ij} = \begin{cases} 1 & \text{if cell } i \text{ and cell } j \text{ are neighbours and } i < j \\ 0 & \text{otherwise} \end{cases}$$  \hspace{1cm} (18)

Coalesced memory access or memory coalescing refers to combining multiple memory accesses into a single transaction. Every successive 128 bytes (32 single
precision words) memory can be accessed by a warp (32 consecutive threads) in a single transaction. Among the conditions that may result in uncoalesced load, i.e., memory access becomes serialized, the more problematic is when memory access is sparse [23]. Structured meshes, in general, have coalesced-pattern implicit in their construction and this allows not only to know implicitly the index of the cell given the edge but also an ordered manner of performing the memory access. This is illustrated by the sketch in Figure 2.

![Figure 2: Access pattern for most common structured meshes](image)

On the other hand, unstructured meshes require to have an auxiliar index vector to perform the calculations by edges. Taking this into account, the coalescence for the access to the cells given the edge depend on their design. An example of the access pattern is described in Figure 3. In contrast to Figure 2, unstructured meshes require an the connectivity between edges and cells.

![Figure 3: Access pattern for unstructured meshes](image)

Two 32-triangular element meshes are defined in Figure 4. The first is a triangular structured mesh and the second is a triangular unstructured mesh constructed using Triangle [27]. Also Figure 4 shows the connectivity matrix of both meshes. That of the structured mesh is very close to a banded matrix whereas that of the unstructured mesh does not have that pattern. This implies that, for the structured mesh, the memory will be more ordered and the memory transactions will be performed faster.

When dealing with unstructured meshes, some manipulations can be applied to change the connectivity matrix to make it become closer to a tridiagonal matrix. The RCM (Reverse Cutchil-McKee) algorithm transforms a sparse matrix into a banded matrix form with a small bandwidth. This bandwidth can be measured as:
\[ \varphi = \max \{|f(v_i) - f(v_j)| : v_iv_j \in E\} \]  

where \(|f(v_i) - f(v_j)|\) defines the distance between elements \(v_i\) and \(v_j\). For our purpose, \(f(v_i)\) is the index of the element \(v_i\) and the difference of the indexes of two neighboring elements \(v_i\) and \(v_j\) represents the distance in memory allocation for those elements.

In terms of matrices, the graph bandwidth is the bandwidth of the symmetric matrix, which represents the adjacency matrix of the graph. Applying this transformation to the unstructured mesh of Figure 4, the bandwidth goes from \(\varphi = 29\) (Original bandwidth) to \(\varphi = 5\) (Figure 5), so, in the worst case, two variables...
related to the same edge, will be allocated at most 6 memory positions apart.

Figure 5: Unstructured RCM processed mesh (left) and connectivity matrix (right)

5.2. AoS vs SoA

A very frequent question when working with arrays is the use of Structure of Arrays (SoA) or Arrays of Structures (AoS). This is more important when working with GPU. In Figure 6 the difference between both options is displayed. Taking into account that the considered variables are $U = \{h, hu, hv\}$, it is necessary to store them ordered by cells. While SoA stores first all $h_{1...NC}$ for the NC mesh cells and then $hu_{1...NC}$ and $hv_{1...NC}$, AoS stores $\{h, hu, hv\}_1$ then $\{h, hu, hv\}_2$ and so on up to $NC$.

Figure 6: Example of configuration of GPU Main Memory for Structure of Arrays and Array of Structures for 5 cells
Contrary to the CPU, the GPU has a very small cache and, moreover, the accesses are improved in groups of 32 elements. Therefore, when a single Thread accesses any memory address, it will move a group of 32 elements starting in that memory address to the cache. If the contiguous Threads access the near memory addresses, there is no need to access the memory again, because the GPU have already brought the following 31 elements to the cache.

Listing 5: Cell access example function. Detailed the access to the variables of interest

```c
__global__ void cellAccess(double *h, double *hu, ..., double *w)
{
    int i = ThreadIdx.x+(BlockIdx.x*BlockDim.x);
    ...
    // SoA
    h=h[i];
    hu=hu[i];
    hv=hv[i];
    ...
    // AoS
    h=w[3*i];
    hu=w[3*i+1];
    hv=w[3*i+2];
    ...
}
```

When the GPU performs the memory load in order to obtain the value of $h$ (for instance) for a given identifier, if the memory is ordered as AoS (Listing 5, lines 10-12), the accesses pattern is 3-displaced. This is so because, given a Thread accessing to position $i$, the neighbouring Thread in the Warp will need to access position $i+3$. However, if the memory is ordered as SoA (Listing 5, lines 5-7) the access to position $i$, will be made concurrently to the access to position $i+1$ by the next Thread and the coalesced access will be automatically produced. It is important to note that if, as in the example, data types are `double`, internally two cycles are required for a Warp. This is so because the 16 first Threads will access the first 128 bytes and the second 16, the next 128 bytes, covering each 16 the amount of bandwidth allowed by the Caching GPU Load Operation. For the case of AoS, the first 6 Threads will hit in the access, but the 6 next Threads will need to load 128 bytes again because of the structure of the memory space. Although AoS is clearer conceptually, SoA improves memory load operations in this type of problems and is thus recommended in GPU implementations.

Bearing in mind that the default mode of the load operation in the GPU is Caching, it attempts to hit the load in L1 and L2 cache with load granularity of
128-Byte. If the data type used is double, it takes 8 bytes to store data and for a Warp, if the data are contiguous, it is necessary to access twice the main memory. On the other hand, when using float only one access is required. For our case, it was decided to write operations as appears in Listing 1. It is worth stressing that the cache mechanism will not be sufficient to compensate a bad coalescence.

5.3. Edge Ordering

The third point deals with edge ordering. Edges are defined by the label of their neighbouring cells $cid_1$ and $cid_2$, and by their own label $i$. When operating on edges, functions like the one in Listing 6 are used. Here, the cell identifiers are stored in two vectors using an SoA approach to improve memory access. The way that accesses are made are the last point where coalescence is necessary.

Listing 6: Edges access example function. Detailed the access to the variables of interest

```c
__global__ void edgeAccess(int *connectivityMatrix1, int *connectivityMatrix2...)
{
    int i = ThreadIdx.x+(BlockIdx.x*BlockDim.x);
    ...cid1=connectivityMatrix1[i];
    ...cid2=connectivityMatrix2[i];
    ...
    h1=h[cid1];
    h2=h[cid2];
    ...}
```

The lack of coalescence happens because the Threads in a Block will be assigned consecutive edge identifiers, but will access variables which have been indexed by cells. See an example in Listing 6 where a function needs to operate on a variable at both sides of the edge. One of the Threads will access edge $i$, retrieve the neighboring cells $cid_1[i]$ and $cid_2[i]$, then access the cell-indexed depth vector: $h[cid_1[i]]$. The next Thread, operating on $i + 1$ will retrieve $cid_1[i + 1]$ and $cid_2[i + 1]$ in a coalesced way. There is no guarantee however, that $cid_1[i]$ and $cid_1[i + 1]$ are close to one another, and thus coalescence problems happen when this Thread requires access $h[cid_1[i + 1]]$.

Listing 7 is presented as a way to force coalescence by means of edges ordering. Note that the matrix is symmetric and an element $\{cid_1, cid_2\}$ appears in $\{cid_2, cid_1\}$. To avoid this replication, it is necessary to establish the matrix as a triangular matrix. Here only the element that verifies $cid_1 < cid_2$ has been kept.
Listing 7: Ordering method for the connectivity matrix based on *qsort* algorithm

```c
struct connectivityMatrix{
  int cid1; int cid2;
};

// The necessary comparison operator for the qsort operation
int cmp(const connectivityMatrix *idx1, const connectivityMatrix *idx2){
  if(idx1->cid1<idx2->cid1){
    return(0);
  } else {
    if(idx1->cid1>idx2->cid1){
      return(1);
    } else {
      if(idx1->cid2<=idx2->cid2){
        return(0);
      } else{
        return(1);
      }
    }
  }
}

void createEdgeIdentifiers(int *connectivityMatrix1, int *connectivityMatrix2){
  // It is very useful to remove elements under the main diagonal. It means, remove duplicated edges in order to avoid replication of the edges. After this operation has been applied, there exist nEdges elements not replicated.
  // NOTE: This operation has been ommited
  edges=(connectivityMatrix*)malloc(sizeof(connectivityMatrix)*nEdges);
  for(i=0;i<nEdges;i++){ edges[i].cid1=connectivityMatrix1[i]; edges[i].cid2=connectivityMatrix2[i]; }
  // *qsort* is defined in stdlib. *cmp* is the function previously defined which defined the criterion of order
  qsort(edges,nEdges,sizeof(connectivityMatrix),cmp);
  for(i=0;i<nEdges;i++){
    connectivityMatrix1[i]=edges[i].cid1;
    connectivityMatrix2[i]=edges[i].cid2;
  }
}
```
With this ordering, defining the $k^{th}$ edge as $w_k = (cid_1, cid_2), 1 < k < nEdges$, and $cid_1$ and $cid_2$ being the identifiers of the neighbouring cells for that edge, the following property is enforced:

$$\{w_{k-1} = (cid_a, cid_b)|cid_a < cid_1 \lor [(cid_a = cid_1) \land (cid_b \leq cid_2)]\}$$  \hfill (20)

Therefore, when Thread $i$ in a Warp accesses position $\text{idx}$ and Thread $i + 1$ to the position $\text{idx} + 1$, the cell data that these Threads must load are as close as possible. Applying this technique to the unstructured mesh presented earlier, it is possible to observe in Figure 7 how the proximity of the numbering can be obtained.

![Figure 7: Unstructured mesh RCM processed edges ordering. Original ordering (left), qsort post-processed ordering (right)](image)

These three optimization strategies, cell ordering, the use of SoA and the edges ordering, help to achieve the proximity in the memory space, allowing Threads the accessing to the nearest possible taking the advantage of the coalescence within a Warp.

6. Results

In order to analyze the performance of our strategy, two test cases have been selected. Both test cases have been run on an Intel Core 2 Quad CPU Q9550 @ 2.83 GHz using an OpenMP shared memory parallelized implementation. In order to explode the Multi-threading of the processor, the comparisons include the
simulation using 1 core and 4 cores. Furthermore, the GPU used for our tests is
a NVIDIA Tesla C2070. The code of the GPU version is the same for all the
combinations, implemented with SoA similarly to Listing 1. On the other hand,
four combinations of GPU calculations have been tested for unstructured meshes.
The first one (called GPU) makes an standard preprocess of the mesh, allocating
data ordered by cell numbering and building the edges by node connectivity (i.e.
following cells ordering). The second one (called GPU+RCM) have the cell num-
bering reordered by the RCM method. The third (GPU+edgeOrd) orders only the
dges. The fourth applies both RCM and Edge Ordering (GPU+RCM+edgeOrd).
For Squared structured meshes there are two tests. One without ordering and the
second one applying the edge ordering. As the meshes have been built as presented
on the examples, the bandwidth, as defined in (19) is proportional to the square root
of the number of cells.

6.1. Test Case 1

Test Case 1 is oriented to demonstrate the influence of the mesh when solving
the shallow water equation and to analyze the behaviour of the GPU in squared
structured meshes and triangular unstructured meshes. This test case is fully ex-
plained in [28] and it is related to the oscillating evolution of the free surface in
a radially symmetrical frictionless paraboloid. The solution is defined with the
parameters \( r_0 = 10m, a = 25.0m, h_0 = 2.5m \) and \( L = 100m \). Squared and
triangular unstructured meshes are processed using RCM. The main properties of
the triangular unstructured meshes and the reduction of their bandwidth are shown
in Table 1. The results for this case are displayed for \( t=T/4, T/2, 3T/4 \) and \( T \)
taking into account that \( \omega = 2\pi/T \) and then, \( T = 11.2192s \) in Figure 8 and Figure
9. Results show differences between both discretizations. The quadrilateral mesh
creates preferential directionality along the axis. This has special interest with the
evolution of the problem and much more when flooding cases are simulated.

Table 3 and Table 4 show the total time \( t \) required by the complete simulation in
all the implementations together with the time used per time step \( (t/t_s) \). According
to Table 3 quadrilateral grids require less steps due to the CFL condition. In that
mesh, \( \chi \) is three times bigger than the smallest one for the unstructured mesh and
then the time step is three times larger. This is the main reason that explains that,
for this case, the absolute execution time \( t \) for the squared mesh is smaller than
for the unstructured mesh with the same number of cells. When the computational
cost of each time step is analyzed, there is another point to have in consideration.
For the same number of cells, the implicated edges in the calculus is bigger for the
Comparing CPU and GPU time-step execution cost, the speed-up goes from

structured mesh.
Figure 9: Comparison of Velocity for structured (lower-left) and unstructured (upper-right) meshes

15x to 30x comparing the mono-core implementation and from 6-11x comparing with the 4 core version. As many authors mention (e.g [29]), squared meshes have better results because of their implicit order in the creation of the mesh but applying our strategies with unstructured meshes, the performance may be doubled.
compared against the non-optimized version going from 30x to nearly 60x for comparison respect to the 1 core run time and from 8x to 20x compared with OpenMP version using 4 core.

<table>
<thead>
<tr>
<th>Mesh</th>
<th>10000</th>
<th>40000</th>
<th>200000</th>
<th>400000</th>
</tr>
</thead>
<tbody>
<tr>
<td>nCells</td>
<td>10015</td>
<td>40030</td>
<td>200797</td>
<td>400680</td>
</tr>
<tr>
<td>$\varphi_D$</td>
<td>9525</td>
<td>39794</td>
<td>100401</td>
<td>200735</td>
</tr>
<tr>
<td>$\varphi_{RCM}$</td>
<td>118</td>
<td>243</td>
<td>372</td>
<td>522</td>
</tr>
</tbody>
</table>

Table 1: Bandwidth analysis for the triangular unstructured meshes used in Test Case 1

It is very important to note that the size of the computational domain varies along the simulation. In the code, there are some control flow statements in order to apply calculations when a pair of cells are dry. This implies that there exists Thread divergence because no calculations are made when two neighboring cells are dry and, as explained in section 5, this may reduce the GPU performance. By means of the purposed technique, this limitation is minimized as ordering the cells
improves locality making more probable the application of the same calculation for all the Threads in the Warp, and the Thread divergence limitation is strongly reduced. In other words, the reordering leads to the warp to process cells with the same properties and then the if or else action will be applied in the whole warp.

6.2. Test Case 2

In the second test case, a dam break problem in a channel with two bumps is considered, and numerical simulations are compared with experimental data. The experimental data were obtained by Aureli et al. [30]. The geometry leads to highly unsteady flow, including shock formation, strong surface curvatures and reverse flow. Therefore, this case is useful to simulate it using a uniform refined mesh and because of its character of rapidly propagation, a good compromise between the different meshes is obtained, allowing good measurements in the comparison of a full computational load case. The flume was rectangular in section, 1.0 m wide, 0.5 m high and 7.0 m long. The dam was placed at \( x=2.25 \) m from the head of the reservoir \( (x=0) \). The water surface elevation in the reservoir is 0.45m. Dry conditions are considered downstream the gate. The slopes of both obstacles are symmetric. Friction losses are computed using the Manning roughness coefficient \( n = 0.007 \text{sm}^{-1/3} \). As in the previous case, the computational domain is represented in unstructured meshes with cell count ranging from 10000 to 400000 cells, setting \( CFL=0.9 \) in all cases. Measurements of water depth versus time were made based on video recordings of the flow. There are three gauging points: \( P_1 \) at \( x=1.40m \), \( P_2 \) at \( x=2.25m \) and \( P_3 \) at \( x=4.5m \) Also, velocity measurements were done at gauging point \( P_1 \).

<table>
<thead>
<tr>
<th>Mesh 10000</th>
<th>40000</th>
<th>100000</th>
<th>200000</th>
<th>400000</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n_{Cells} )</td>
<td>10171</td>
<td>40033</td>
<td>100138</td>
<td>200440</td>
</tr>
<tr>
<td>( \varphi_0 )</td>
<td>10094</td>
<td>36613</td>
<td>100098</td>
<td>199658</td>
</tr>
<tr>
<td>( \varphi_{RCM} )</td>
<td>39</td>
<td>86</td>
<td>129</td>
<td>204</td>
</tr>
</tbody>
</table>

Table 2: Bandwidth analysis for the triangular unstructured meshes used in Test Case 2

Computed and measured data are in good agreement (Figure 13). Differences can be explained by the limits of the mathematical model, that considers purely hydrostatic forces along the flow. On the other hand, computational effort is also reduced by means of this techniques. When applying \( RCM \) (Table 2, Figure 11), \( \varphi \) varies 3 orders of magnitude, leading to a good coalesced pattern for this mesh.
The behaviour of the technique is very similar to the one found in the previous test case. Those domains containing less than 50000 cells, offer gain values of around 15% when applying all the proposed techniques. When using more than 100000 computational cells, the speed-up reaches values around 60x representing an increase of around a 23%. It is very important to note that, as in the previous case, applying only RCM is not enough. In fact, the sole application of the RCM could lead to a worse performance because although the cell access pattern will be coalesced, the edge access pattern could be less coalesced as a consequence of RCM. In that case, the speed-up could be lower than for computing without any modification in the mesh.

Contrary to the previous test case where the wet domain changes all the time, in this second test case all the domain is included almost immediately after the gate removal at the beginning in the calculation. This avoids Thread divergences and hence the speed-up for the not optimized version is higher.

7. Conclusions

A general implementation of a finite volume scheme on Graphic Processors Unit has been analyzed in this work. Unstructured meshes are useful for many fields of the engineering but GPU designing is oriented to structured meshes. The results have been compared with an OpenMP implementation. In this work, some
improvements on the way of preprocessing the meshes have demonstrated good results, improving the performance of the GPU implementation when unstructured meshes are used. Specifically the use of SoA for the definition of variables, and the application of cell and edge ordering algorithms have been shown to have a noticeable influence on GPU performance. When not all the domain is used, these strategies allow coalesced memory to avoid Thread divergences. When almost all the domain is used, the profits of the coalesced memory access is obtained increasing in both cases the performance between 10% − 25% with small meshes and with no moving boundaries and 60% − 100% with more than 100000 cells and variable domain.

These results provide a very useful technique that lead to have a very efficient implementation on modern GPUs. It is noting worth that OpenMP parallel version is really easy to be implemented in a reasonable time while the CUDA version require more time as well as to be careful with details related to the architecture.

The proposed strategy can be extended to other solvers based on calculations by edges and also to other methods where calculations and storage are performed by nodes to simulate unsteady flows on unstructured meshes.
Figure 13: Water level evolution in time. Experimental data (−•−) and numerical results (−□−) for Test Case 2
8. Acknowledgments

This work has been partially supported by the European Social Fund (Aragon Government). We also thank NVIDIA for the donation of hardware facilities.
References


<table>
<thead>
<tr>
<th></th>
<th>t (ms)</th>
<th>t/t_s</th>
<th>vs. CPU 1 Core</th>
<th>vs. CPU 4 Core</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>10000 Cells</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU1Core - Unst. Mesh</td>
<td>4991.81</td>
<td>5.57</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU1Core - Sq. Mesh</td>
<td>3953.16</td>
<td>7.40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU4Core - Unst. Mesh</td>
<td>1783.99</td>
<td>1.99</td>
<td>2.79</td>
<td></td>
</tr>
<tr>
<td>CPU4Core - Sq. Mesh</td>
<td>1470.98</td>
<td>2.75</td>
<td>2.69</td>
<td></td>
</tr>
<tr>
<td>GPU - Unst. Mesh</td>
<td>295.56</td>
<td>0.33</td>
<td>16.89</td>
<td>5.43</td>
</tr>
<tr>
<td>GPU+RCM - Unst. Mesh</td>
<td>306.89</td>
<td>0.34</td>
<td>16.27</td>
<td>5.23</td>
</tr>
<tr>
<td>GPU+edgeOrd - Unst. Mesh</td>
<td>283.04</td>
<td>0.32</td>
<td>17.64</td>
<td>5.67</td>
</tr>
<tr>
<td>GPU+RCM+edgeOrd - Unst. Mesh</td>
<td>267.63</td>
<td><strong>0.30</strong></td>
<td><strong>18.65</strong></td>
<td><strong>6.00</strong></td>
</tr>
<tr>
<td>GPU - Sq. Mesh</td>
<td>178.28</td>
<td>0.33</td>
<td>22.17</td>
<td>8.25</td>
</tr>
<tr>
<td>GPU+edgeOrd - Sq. Mesh</td>
<td>177.29</td>
<td>0.33</td>
<td>22.30</td>
<td>8.67</td>
</tr>
<tr>
<td><strong>40000 Cells</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU1Core - Unst. Mesh</td>
<td>58105.63</td>
<td>28.98</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU1Core - Sq. Mesh</td>
<td>35323.78</td>
<td>32.53</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU4Core - Unst. Mesh</td>
<td>21599.27</td>
<td>10.77</td>
<td>2.69</td>
<td></td>
</tr>
<tr>
<td>CPU4Core - Sq. Mesh</td>
<td>14054.87</td>
<td>12.94</td>
<td>2.51</td>
<td></td>
</tr>
<tr>
<td>GPU - Unst. Mesh</td>
<td>1921.82</td>
<td>0.96</td>
<td>30.23</td>
<td>8.99</td>
</tr>
<tr>
<td>GPU+RCM - Unst. Mesh</td>
<td>2120.12</td>
<td>1.06</td>
<td>27.41</td>
<td>8.15</td>
</tr>
<tr>
<td>GPU+edgeOrd - Unst. Mesh</td>
<td>1656.17</td>
<td>0.83</td>
<td>35.08</td>
<td>10.43</td>
</tr>
<tr>
<td>GPU+RCM+edgeOrd - Unst. Mesh</td>
<td>1401.03</td>
<td><strong>0.70</strong></td>
<td><strong>41.47</strong></td>
<td><strong>12.33</strong></td>
</tr>
<tr>
<td>GPU - Sq. Mesh</td>
<td>839.92</td>
<td>0.77</td>
<td>42.06</td>
<td>16.74</td>
</tr>
<tr>
<td>GPU+edgeOrd - Sq. Mesh</td>
<td>841.08</td>
<td>0.77</td>
<td>42.00</td>
<td>16.71</td>
</tr>
<tr>
<td><strong>100000 Cells</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU1Core - Unst. Mesh</td>
<td>264078.60</td>
<td>80.05</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU1Core - Sq. Mesh</td>
<td>160182.18</td>
<td>88.21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU4Core - Unst. Mesh</td>
<td>98068.16</td>
<td>29.72</td>
<td>2.69</td>
<td></td>
</tr>
<tr>
<td>CPU4Core - Sq. Mesh</td>
<td>66227.14</td>
<td>36.47</td>
<td>2.42</td>
<td></td>
</tr>
<tr>
<td>GPU - Unst. Mesh</td>
<td>8346.91</td>
<td>2.53</td>
<td>31.63</td>
<td>10.57</td>
</tr>
<tr>
<td>GPU+RCM - Unst. Mesh</td>
<td>9174.20</td>
<td>2.78</td>
<td>28.78</td>
<td>9.62</td>
</tr>
<tr>
<td>GPU+edgeOrd - Unst. Mesh</td>
<td>6750.18</td>
<td>2.05</td>
<td>39.12</td>
<td>13.08</td>
</tr>
<tr>
<td>GPU+RCM+edgeOrd - Unst. Mesh</td>
<td>5267.17</td>
<td><strong>1.60</strong></td>
<td><strong>50.14</strong></td>
<td><strong>16.76</strong></td>
</tr>
<tr>
<td>GPU - Sq. Mesh</td>
<td>3392.74</td>
<td>1.87</td>
<td>47.21</td>
<td>19.52</td>
</tr>
<tr>
<td>GPU+edgeOrd - Sq. Mesh</td>
<td>3384.75</td>
<td>1.86</td>
<td>47.32</td>
<td>19.56</td>
</tr>
<tr>
<td><strong>200000 Cells</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU1Core - Unst. Mesh</td>
<td>840913.36</td>
<td>162.46</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU1Core - Sq. Mesh</td>
<td>455589.91</td>
<td>176.59</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU4Core - Unst. Mesh</td>
<td>332028.52</td>
<td>64.14</td>
<td>2.53</td>
<td></td>
</tr>
<tr>
<td>CPU4Core - Sq. Mesh</td>
<td>170974.87</td>
<td>66.27</td>
<td>2.66</td>
<td></td>
</tr>
<tr>
<td>GPU - Unst. Mesh</td>
<td>26533.18</td>
<td>5.13</td>
<td>31.69</td>
<td>11.26</td>
</tr>
<tr>
<td>GPU+RCM - Unst. Mesh</td>
<td>28943.84</td>
<td>5.59</td>
<td>29.05</td>
<td>10.32</td>
</tr>
<tr>
<td>GPU+edgeOrd - Unst. Mesh</td>
<td>20562.27</td>
<td>3.97</td>
<td>40.90</td>
<td>14.53</td>
</tr>
<tr>
<td>GPU+RCM+edgeOrd - Unst. Mesh</td>
<td>15639.76</td>
<td><strong>3.02</strong></td>
<td><strong>53.77</strong></td>
<td><strong>19.11</strong></td>
</tr>
<tr>
<td>GPU - Sq. Mesh</td>
<td>8998.99</td>
<td>3.49</td>
<td>50.63</td>
<td>19.00</td>
</tr>
<tr>
<td>GPU+edgeOrd - Sq. Mesh</td>
<td>8992.23</td>
<td>3.49</td>
<td>50.66</td>
<td>19.01</td>
</tr>
<tr>
<td><strong>400000 Cells</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU1Core - Unst. Mesh</td>
<td>322224.69</td>
<td>337.04</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU1Core - Sq. Mesh</td>
<td>1299827.96</td>
<td>352.73</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU4Core - Unst. Mesh</td>
<td>1411913.42</td>
<td>10.02</td>
<td>2.35</td>
<td></td>
</tr>
<tr>
<td>CPU4Core - Sq. Mesh</td>
<td>525174.53</td>
<td>142.51</td>
<td>2.48</td>
<td></td>
</tr>
<tr>
<td>GPU - Unst. Mesh</td>
<td>109699.40</td>
<td>11.13</td>
<td>30.28</td>
<td>11.58</td>
</tr>
<tr>
<td>GPU+RCM - Unst. Mesh</td>
<td>117526.78</td>
<td>11.92</td>
<td>28.27</td>
<td>10.81</td>
</tr>
<tr>
<td>GPU+edgeOrd - Unst. Mesh</td>
<td>83056.90</td>
<td>8.43</td>
<td>40.00</td>
<td>15.30</td>
</tr>
<tr>
<td>GPU+RCM+edgeOrd - Unst. Mesh</td>
<td>56543.50</td>
<td><strong>5.74</strong></td>
<td><strong>58.76</strong></td>
<td><strong>22.47</strong></td>
</tr>
<tr>
<td>GPU - Sq. Mesh</td>
<td>24195.92</td>
<td>6.57</td>
<td>53.72</td>
<td>21.70</td>
</tr>
<tr>
<td>GPU+edgeOrd - Sq. Mesh</td>
<td>24180.67</td>
<td>6.56</td>
<td>53.75</td>
<td>21.72</td>
</tr>
</tbody>
</table>

Table 3: Test Case 1: CPU and GPU total execution time (t) and per time step (t/t_s) together with performance of the GPU version using different configurations. CPU1Core is a sequential version on an Intel Core 2 Quad CPU Q9550 @ 2.83 GHz and CPU4Core is the OpenMP implementation using 4 Cores.
Table 4: Test Case 2: CPU and GPU total execution time (t) and per time step (t/t_s) together with performance of the GPU version using different configurations. CPU_1Core is a sequential version on an Intel Core 2 Quad CPU Q9550 @ 2.83 GHz and CPU_4Core is the OpenMP implementation using 4 Cores.

<table>
<thead>
<tr>
<th></th>
<th>t (ms)</th>
<th>t/t_s</th>
<th>vs. CPU_1Core</th>
<th>vs. CPU_4Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000 Cells</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU_1Core</td>
<td>41532.30</td>
<td>7.49</td>
<td>2.51</td>
<td></td>
</tr>
<tr>
<td>CPU_4Core</td>
<td>36165.99</td>
<td>2.94</td>
<td></td>
<td>2.51</td>
</tr>
<tr>
<td>GPU</td>
<td>1969.88</td>
<td>0.36</td>
<td>21.08</td>
<td>7.36</td>
</tr>
<tr>
<td>GPU+RCM</td>
<td>2013.51</td>
<td>0.36</td>
<td>20.63</td>
<td>7.20</td>
</tr>
<tr>
<td>GPU+edgeOrd</td>
<td>1882.64</td>
<td>0.34</td>
<td>22.06</td>
<td>7.70</td>
</tr>
<tr>
<td>GPU+RCM+edgeOrd</td>
<td>1738.89</td>
<td>0.31</td>
<td>23.88</td>
<td>8.34</td>
</tr>
<tr>
<td>40000 Cells</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU_1Core</td>
<td>505401.30</td>
<td>33.40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU_4Core</td>
<td>417096.00</td>
<td>15.51</td>
<td></td>
<td>2.15</td>
</tr>
<tr>
<td>GPU</td>
<td>13762.87</td>
<td>0.91</td>
<td>36.72</td>
<td>11.63</td>
</tr>
<tr>
<td>GPU+RCM</td>
<td>15138.44</td>
<td>1.00</td>
<td>33.39</td>
<td>10.58</td>
</tr>
<tr>
<td>GPU+edgeOrd</td>
<td>12613.27</td>
<td>0.83</td>
<td>40.07</td>
<td>12.69</td>
</tr>
<tr>
<td>GPU+RCM+edgeOrd</td>
<td>11102.60</td>
<td>0.73</td>
<td>45.52</td>
<td>14.42</td>
</tr>
<tr>
<td>100000 Cells</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU_1Core</td>
<td>2295837.22</td>
<td>88.48</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU_4Core</td>
<td>1835517.78</td>
<td>36.50</td>
<td></td>
<td>2.42</td>
</tr>
<tr>
<td>GPU</td>
<td>51589.11</td>
<td>1.99</td>
<td>44.50</td>
<td>12.46</td>
</tr>
<tr>
<td>GPU+RCM</td>
<td>58088.87</td>
<td>2.24</td>
<td>39.52</td>
<td>11.07</td>
</tr>
<tr>
<td>GPU+edgeOrd</td>
<td>46221.68</td>
<td>1.78</td>
<td>49.67</td>
<td>13.91</td>
</tr>
<tr>
<td>GPU+RCM+edgeOrd</td>
<td>41278.49</td>
<td>15.57</td>
<td>55.62</td>
<td>44.47</td>
</tr>
<tr>
<td>200000 Cells</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU_1Core</td>
<td>4949436.05</td>
<td>180.39</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU_4Core</td>
<td>3912997.63</td>
<td>79.13</td>
<td></td>
<td>2.27</td>
</tr>
<tr>
<td>GPU</td>
<td>103658.95</td>
<td>3.78</td>
<td>47.75</td>
<td>17.65</td>
</tr>
<tr>
<td>GPU+RCM</td>
<td>120357.69</td>
<td>4.39</td>
<td>41.12</td>
<td>15.20</td>
</tr>
<tr>
<td>GPU+edgeOrd</td>
<td>93125.04</td>
<td>3.39</td>
<td>53.15</td>
<td>19.64</td>
</tr>
<tr>
<td>GPU+RCM+edgeOrd</td>
<td>82877.25</td>
<td>3.02</td>
<td>59.72</td>
<td>22.06</td>
</tr>
<tr>
<td>400000 Cells</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU_1Core</td>
<td>14448850.66</td>
<td>359.22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU_4Core</td>
<td>11717537.88</td>
<td>169.62</td>
<td></td>
<td>2.11</td>
</tr>
<tr>
<td>GPU</td>
<td>306335.43</td>
<td>7.62</td>
<td>47.17</td>
<td>17.43</td>
</tr>
<tr>
<td>GPU+RCM</td>
<td>350376.72</td>
<td>8.71</td>
<td>41.24</td>
<td>15.24</td>
</tr>
<tr>
<td>GPU+edgeOrd</td>
<td>269668.53</td>
<td>6.70</td>
<td>53.58</td>
<td>19.80</td>
</tr>
<tr>
<td>GPU+RCM+edgeOrd</td>
<td>234851.10</td>
<td>5.84</td>
<td>61.52</td>
<td>22.74</td>
</tr>
</tbody>
</table>