STUDIA INFORMATICA 2013
Volume 34 Number 1 (110)

Łukasz ROGUSKI
Polish-Japanese Institute of Information Technology
Sebastian DEOROWICZ
Silesian University of Technology, Institute of Informatics

FLUID MOTION MODELLING USING VORTEX PARTICLE METHOD ON GPU

Summary. In this paper we present the vortex-in-cell method aimed at graphic processor units. Inviscid fluid model is examined in domain with periodic boundary conditions. The leap-frogging vortex rings simulation results are presented with sample vortex rings collision visualization. At the end the GPU solver performance advantage over CPU solver is presented.

Keywords: 3D fluid motion modeling, vortex particle method, vortex-in-cell method, GPU, CUDA

MODELOWANIE RUCHU PŁYNÓW NA PODSTAWIE METODY CZĄSTEK WIROWYCH NA GPU

Strzeszczenie. W pracy prezentujemy metodę obliczeniową wir-w-komórce zaimplementowaną na układach graficznych. Za model ośrodka został przyjęty płyn nielepki wraz z periodycznymi warunkami brzegowymi. W pracy przedstawiono wyniki symulacji dla gry wirów oraz przykładowe wizualizacje z wykorzystaniem cząsteczek markerów. Pod koniec została przedstawiona analiza uzyskanego przyspieszenia algorytmu na GPU względem wersji na CPU.

Słowa kluczowe: modelowanie ruchu płynów w 3D, metoda cząstek wirowych, metoda wir-w-komórce, GPU, CUDA

1. Introduction

The computational fluid dynamics is one of the most intensive developing branches of physics in recent years, where fluid motion modeling problems can be solved virtually in computer simulations, reducing the costs of experiments. The main problem, however, remains a construction of accurate flow and fluid model descriptions, where especially for turbulent cases obtaining the exact solution is almost impossible. To better understand the turbulence phenomenon the vortex methods were developed, which focuses on the vortical characteristics of the flow.

In vortex methods we can distinguish two main branches – the direct and hybrid methods. The first, direct methods, relies on Biot-Savart law, where to compute the velocity at a given point in space one needs to sum up all the particles contributions in domain, which ends up to the n-body problem [1, 2, 3]. The second, hybrid methods, use the computational grid to solve Poisson equation and then obtain a velocity field of the computational domain [1, 4]. As both methods are computationally expensive, the usage of high-throughput computational devices is highly desirable.

In recent years we can see an increasing trend in using graphic cards processing power to scientific computations. The GPUs used to be seen as devices for processing and displaying the graphics on monitor. Nowadays, we can treat GPUs as high-throughput parallel arithmetic coprocessors, capable of handling thousands of lightweight threads. GPUs have been widely used in computational fluid dynamics field, simulating fluid flow using grid-based methods [5, 6] and smoothed particle hydrodynamics methods, thus proving to be a highly-efficient tools. However application of GPUs for solving fluid flow problems using vortex methods is a rather young practice. In [7] researchers used GPUs to speedup three-dimensional flow problems using direct methods and fast treecode or fast multipole modifications, which proved to be handled efficiently by graphics processors. The hybrid method solver was successfully implemented for GPU [8] and then improved in [9], which gave ability to simulate two-dimensional bluff body flows in inviscid fluid with very good speedup. Quite recently also the researchers [10] successfully implemented the multigrid hybrid solver on GPU obtaining interesting results.

In the current work, we present a hybrid solver designed for solving the three-dimensional incompressible and inviscid flows using Fast Fourier Transform solver on GPU. We also present the sample simulation visualizations accompanied with GPU speedup results analysis.

2. Physical background

The common basis of most fluid motion modeling problems are the Navier-Stokes equations, describing the conservation of mass, momentum and kinetic energy. As a starting point we use the conservation of momentum equation, which for incompressible and inviscid fluid model with no external forces is determined as:

$$\begin{cases} \dfrac{\partial \vec{u}}{\partial t} + (\vec{u} \cdot \nabla)\vec{u} = \dfrac{-1}{\rho} \nabla p, \\ \nabla \cdot \vec{u} = 0, \end{cases} \tag{1}$$

with velocity $\vec{u} = (x, y, z)$, pressure p, and density q.

Applying the differential rotation operator for two sides of the first equation, we obtain the Navier-Stokes equation in vorticity form and after closing it with the condition of fluid incompressibility $\nabla \cdot \vec{u} = 0$ we obtain the full system of equations describing the evolution of vorticity field in time. For three-dimensional flow with selected incompressible and inviscid fluid model the system of equations follows:

$$\begin{cases} \dfrac{\partial \vec{\omega}}{\partial t} + (\vec{u} \cdot \nabla)\vec{u} = (\vec{\omega} \cdot \nabla)\vec{u}, \\ \nabla \cdot \vec{u} = 0, \end{cases} \tag{2}$$

where vorticity $\vec{\omega} = (\omega_x, \omega_y, \omega_z)$ is described as:

$$\vec{\omega} = \nabla \times \vec{u}. \tag{3}$$

The most difficult problem lies in obtaining the velocity field. By applying the Helmholtz theorem about decomposition in relation to fluid flow and taking into account the condition of incompressibility, the velocity equation can be described:

$$\vec{u} = \nabla \times \vec{A}, \tag{4}$$

Where $\vec{A}$ is the vector potential.

The vorticity field is rotation of velocity field, so assuming additionally the non divergent potential field ($\nabla \cdot \vec{A} = 0$) [4], we obtain the vector potential $\vec{A}$, which simplifies the solution to solving:

$$\nabla^2 \vec{A} = -\vec{\omega}, \tag{5}$$

from which the obtaining the velocity field is simple.

3. Vortex-in-cell method

The vortex-in-cell method is a branch of vortex particle methods, where in order to compute the velocity field effectively, a computational grid is used. In this method, the three dimensional domain of flow is covered by a mesh of $N_x \times N_y \times N_z$ size, with equal spacing h between grid nodes. The continuous vorticity field is discretized into vortex particles (or point vortices) [4]:

$$\vec{\omega}(\vec{x}) = \sum_p \vec{\alpha}_p \delta(\vec{x} - \vec{x}_p), \tag{6}$$

where $\delta($ is the delta Dirac function, $\vec{\alpha}_p$ is the vortex particle strength and is described as:

$$\vec{\alpha}_p = \int_V \vec{\omega}_p d\vec{x} \approx \vec{\omega}(\vec{x}_p) v_p, \tag{7}$$

where v_p is the particle volume.

Evolution of vorticity field is thus defined by evolution of vortex particles carrying the vorticity. As the vortex particle motion is passive (particles are advected in the velocity field), the rate of change in their position is described as

$$\frac{d\vec{x}_p}{dt} = \vec{u}_p. \tag{8}$$

Unlike in modeling two-dimensional flow problems – where vorticity is a constant – in three-dimensional flows the rate of change in particle vorticity is described as stretching term

$$\frac{d\vec{\alpha}_p}{dt} = [\nabla \vec{u}(\vec{x}_p, t)] \cdot \vec{\alpha}_p, \tag{9}$$

where $\nabla \vec{u}(\vec{x}_p, t)$ states velocity field gradient at $\vec{x}_p$ position and in t moment.

The procedure for incompressible and inviscid fluid motion modeling using the vortex-in-cell method can be basally divided into 5 steps:

1. Distribution of particle strengths onto the mesh nodes

$$\vec{\omega}(\vec{x}_n) = \frac{1}{v_p} \sum_p \vec{\alpha}_p \vec{\phi}(\vec{x}_p), \tag{10}$$

where $\vec{\omega}(\vec{x}_n)$ is the vorticity in the specified grid node and ϕ is the interpolation (or filter) function. In the current work, the trilinear filter function is selected.

2. Solution of the second order differential Poisson equation on grid in order to obtain the vector potential $\vec{A}$ (5). In the paper we use the finite differences method with periodic boundary conditions similar in [2, 4, 9].
3. Computation of velocity field based on previously obtained vector potential $\vec{A}$ (5) and computation of velocity field gradients.
4. Interpolation of velocity values from the velocity field mesh nodes to the particles

$$\vec{u}(\vec{x}_p) = \sum_n \vec{u}_n (\vec{x}_p), \tag{11}$$

where $\vec{u}(\vec{x}_n)$ is the calculated velocity in grid node and $\phi(\vec{x})$ is the interpolation filter. Particle positions (8) and strengths (9) are updated using second order Adams-Bashfoth multi-step scheme:

$$a_{n+2} = a_{n+1} + \frac{3}{2} hf(t_{n+1}, a_{n+1}) - \frac{1}{2} hf(t_n, a_n), \tag{12}$$

where h is the size of time step.

5. Update of the free markers positions, which are advected in velocity field just as vortex particles (8).

4. GPU implementation details

To aid the computations by using GPU the NVIDIA, CUDA interface was chosen, which enables programmer to treat the GPU as an high-throughput parallel arithmetic coprocessor. Although on the market there are other available programming interfaces for parallel computing using GPUs including the open standard OpenCL and Microsoft DirectCompute (as part of DirectX API), the NVIDIA CUDA toolkit offers the largest number of additional libraries and code base for parallel computing at the cost of being limited only to NVIDIA GPU cards.

The CUDA interface describes GPU as a computational device, consisting of multiple independent processor groups called streaming multiprocessors, which can operate in parallel and handle thousands of lightweight hardware threads. Each streaming multiprocessor consists of multiple smaller arithmetic operation units – stream processors, which operating in thread groups called warps execute the same instruction.

NVIDIA CUDA interface describes also the 3-level device memory hierarchy - global memory, shared memory, and registers. The first, usually can be seen as the RAM of the GPU and can be accessed by all computational units. The global memory has the largest memory address space but also the largest memory latency access time, so it's important to design algorithms to make use of it as less as possible. The shared memory, on the other hand is per streaming multiprocessor exclusive space with size is less than 50 kB, but has access times usually an order of magnitude smaller than in the global memory case, being the best solution for e.g. swap storage between execution units. The last – registers – are exclusive per streaming processor with negligible access time latency.

Having in mind the CUDA device model, below we present the implementation details of the selected vortex-in-cell method steps.

4.1. Particle strengths distribution on mesh nodes

In the particle strength distribution step, we selected the trilinear interpolation function, where the strength of single particle is distributed between 8 grid nodes of the cell in which the particle resides. Fig. 1 presents the GPU kernel pseudocode of this step.

At the beginning, particle index in particle buffer is computed, basing on available information for worker threads that is `threadId`, `groupId` and `blockId`. Next, the particle index offset `ioff` is determined, which depends on total number of particles, threads number and processing slice size. During this stage, each multiprocessor operates on a consistent blocks of particles data, enabling coalesced memory reads. Each worker thread in a warp processes a slice of *n* particles – this process is divided into *n* iterations, giving *n* memory block read transactions initiated by multiprocessor and a minimum of *8n* memory write transactions (they cannot unfortunately be coalesced). The aim of processing particle slices by a worker thread is mainly to reduce number of active threads, when the overall number of particles is too large to be efficiently handled in 1:1 thread to particle ratio. The total number of launched threads is thus $t_{total} = \frac{particlecount}{slice\,\mathrm{size}}$.

```
Input:    P - vortex particles buffer
          M - vorticity field mesh
          n - number of particles to process

Output:   M - (processing in-place)

function DistributeStrengthsOnMesh(P, M, n)
    in ← determine the particle index based on worker thread information
    ioff ← index offset between processing particles
    for i ← 0 to n do
        p ← P[ in + i * ioff ]
        I ← compute indices of cell nodes in which particle p resides
        for all i in I do
            update the vorticity in M, node using p
        end for
        synchronize threads in group
    end for
end
```

Fig. 1. Particle strength distribution step – GPU kernel pseudocode
Rys. 1. Pseudokod programu jądra etapu dystrybucji natężenia cząsteczek na GPU

4.2. Poisson equation solution

In this paper we use the fast spectral method solver to solve the second order elliptic Poisson equation (5) in frequency domain with periodic domain boundaries. The high efficiency of spectral methods in solving partial differential equations comes from the fact, that in the frequency domain the costly differential operations on grid are replaced by simple division by wavenumber, operations reducing the computational costs. In this step we used the CUFFT library, which is a part of NVIDIA CUDA toolkit and which implements the Fast Fourier Transform on GPU. The pseudocode illustrating the main concept of this step is presented on Fig. 2.

```
Input:    M – vorticity field in time domain
          n – slice size of mesh elements to process

Output:   A – vector potential in time domain

function SolvePoissonFFT(M, n, A)
    B ← convert M from real to complex
    B ← forward FFT B
    launch SolvePoissonInFrequencyDomain(B, n) kernel
    B ← inverse FFT B
    A ← convert B from complex to real
end
```

Fig. 2. Poisson equation solution main step – GPU kernel pseudocode
Rys. 2. Pseudokod programu jądra głównego etapu rozwiązywania równania Poissona na GPU

Our solver takes vorticity mesh buffer `M` as input and places the result in vector potential `A` buffer. In order to compute the Fourier transform and solution in the frequency domain, the input data must be firstly converted from areal number of `float` type to the complex type `cufftComplex` and – analogously after obtaining solution and inverse Fourier transform – back to the real type, saving the result in output `A` vector potential buffer. Here, the auxiliary `B` buffer is used, where transformation operations are computed in-place. The `SolvePoissonInFrequencyDomain()` is the kernel function launched on all worker threads, which pseudocode is presented on Fig. 3.

```
Input:    M – vorticity field in frequency domain buffer
          n – slice size of mesh elements to process

Output:   M (processing in-place)

function SolvePoissonInFrequencyDomain(M, n)
    x, y ← determine the X, Y indices based on worker thread information
    $z_0$ ← determine Z slice start index
    for z ← $z_0$ to $z_0$ + n
        k ← compute the wavenumber for $M_{xyz}$ element
        $M_{xyz}$ ← $M_{xyz}$ / k
        synchronize threads in group
  end for
end
```

Fig. 3. Poisson equation solution in frequency domain - GPU kernel pseudocode
Rys. 3. Pseudokod programu jądra rozwiązywania równania Poissona w dziedzinie częstotliwości na GPU

At first the global `x`, `y` element indices are computed with `z0` slice start index. In the main for loop, the n elements in slice are processed – divided by previously computed wavenumber. As the elements are only read and written once, memory operations are made on single buffer `M` which works both as input and output. At the end of the `for` loop the synchronization operation between worker threads in a group is inserted, which is needed to achieve the coalesced memory reads. The overall number of threads running the kernel is $t_{total} = \frac{N_x \times N_y \times N_z}{n}$, where *Ni* corresponds to the *i*-th dimension size of vorticity field grid.

4.3. Vortex particle update

After computation of a velocity field from vector potential and velocity field gradients, the particles update step is done. The pseudocode for GPU particles update kernel is presented on Fig. 4. Analogously, like in the particle strength distribution kernel, the particle index in buffer with particle index offset for slice processing are firstly determined, where the total number of launched threads on GPU is $t_{total} = \frac{particleco\,unt}{slice\mathrm{size}}$.

```
Input:    P – vortex particles buffer
          V – velocity field mesh
          G – velocity field gradients
          h – time step size
          n – number of particles to process

Output:   P (processing in-place)

function UpdateParticlesAB2(P, V, G, h, n)
    in ← determine the particle index based on worker thread information
    ioff ← index offset between processing particles
    for i ← 0 to n do
        ip ← in + i * ioff
        p ← P[ ip ]
        v ← sample velocity from V at p.x      // p.x – position at t-1
        g ← sample velocity gradients from G at p.x
        f ← compute stretching using g and p.a // p.a – strength at t-1
        p.x ← p.x + 0.5h(3v – p.v)            // p.v – velocity at t-1
        p.a ← p.a + 0.5h(3f – p.f)            // p.f – stretching at t-1
        p.v ← v
        p.f ← f
        P[ ip ] ← p                           // save new particle value
        synchronize threads in group
    end for
end
```

Fig. 4. Vortex particles update step – GPU kermel pseudocode
Rys. 4. Pseudokod programu jądra etapu aktualizacji cząsteczek wirowych na GPU

At the beginning of the `for` iteration, the particle data with specified index is read from buffer and next – the velocity with gradients at particle position are sampled, using trilinear interpolation function. The stretching term is computed according to (9) and particle position with stretching value are updated using second order Adams-Bashforth scheme. At the end of the iteration the current sampled velocity and stretching values are saved, which will be used again in the particle update step at *t+1* time.

4.4. Visualization

To visualize the simulation results, we used OpenGL graphics interface library with supported `cuda_gl_interop` option to share GPU buffers with CUDA interface, which makes almost instant results rendering possible. Vortex (and free marker) particles are represented as simple `GL_POINTS` with positions stored in CUDA particle buffer, while particle color is set globally in order to reduce needed operations or additional GPU memory color buffer.

5. Numerical results

As a numerical vortex ring model for our simulations we chose a ring (or rather a torus) consisting of 256 slices, with 121 vortex particles on each, giving a total sum of 30976 particles per whole vortex ring. The basic idea of construction of vortex ring is presented in Fig. 5 – we chose the distribution model of particles inside the slice similar as in [4].

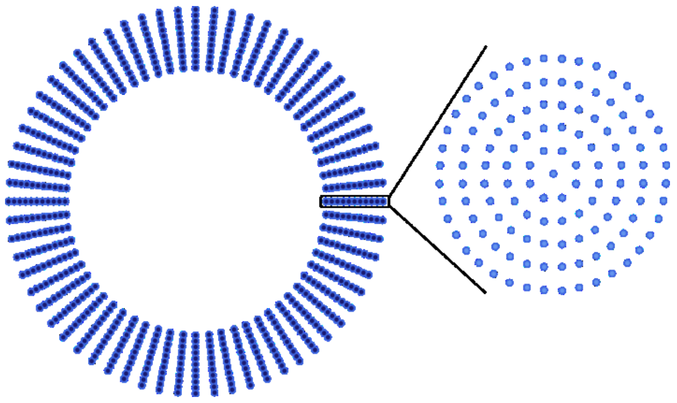

Fig. 5. Vortex ring construction idea
Rys. 5. Idea konstrukcji pierścienia wirowego

5.1. Leap frogging vortex rings simulations

As a first simulation to test the environment the vortex rings leapfrogging phenomenon, was chosen, known also as the “vortex game”. In this phenomenon we observe a specific interaction between two vortex rings moving along the common axis in the same direction. While traveling, the second ring – which is on the back towards the direction of movement – under the velocity field inducted by the heading ring starts to accelerate and deform – shrink it's size. The heading ring, however, under the velocity field induced by the back ring starts reducing it's velocity and also deform – stretches it's own size. When the distance between rings reaches zero, the ring on the back passes through the center of the heading ring and starts to accelerate with stretching it's size. The roles change and the process continues. This experiment is very hard to carry on in the laboratory, as it's stability depends on conservation of vortex rings shapes and their starting positions. Also the inviscid fluid model is required. In an idealistic situation the rings should move in that manner to infinity, but in the computer simulations, due to e.g. particle distortion or numerical diffusion the vortex rings become distorted and break into pieces or connect to one structure.

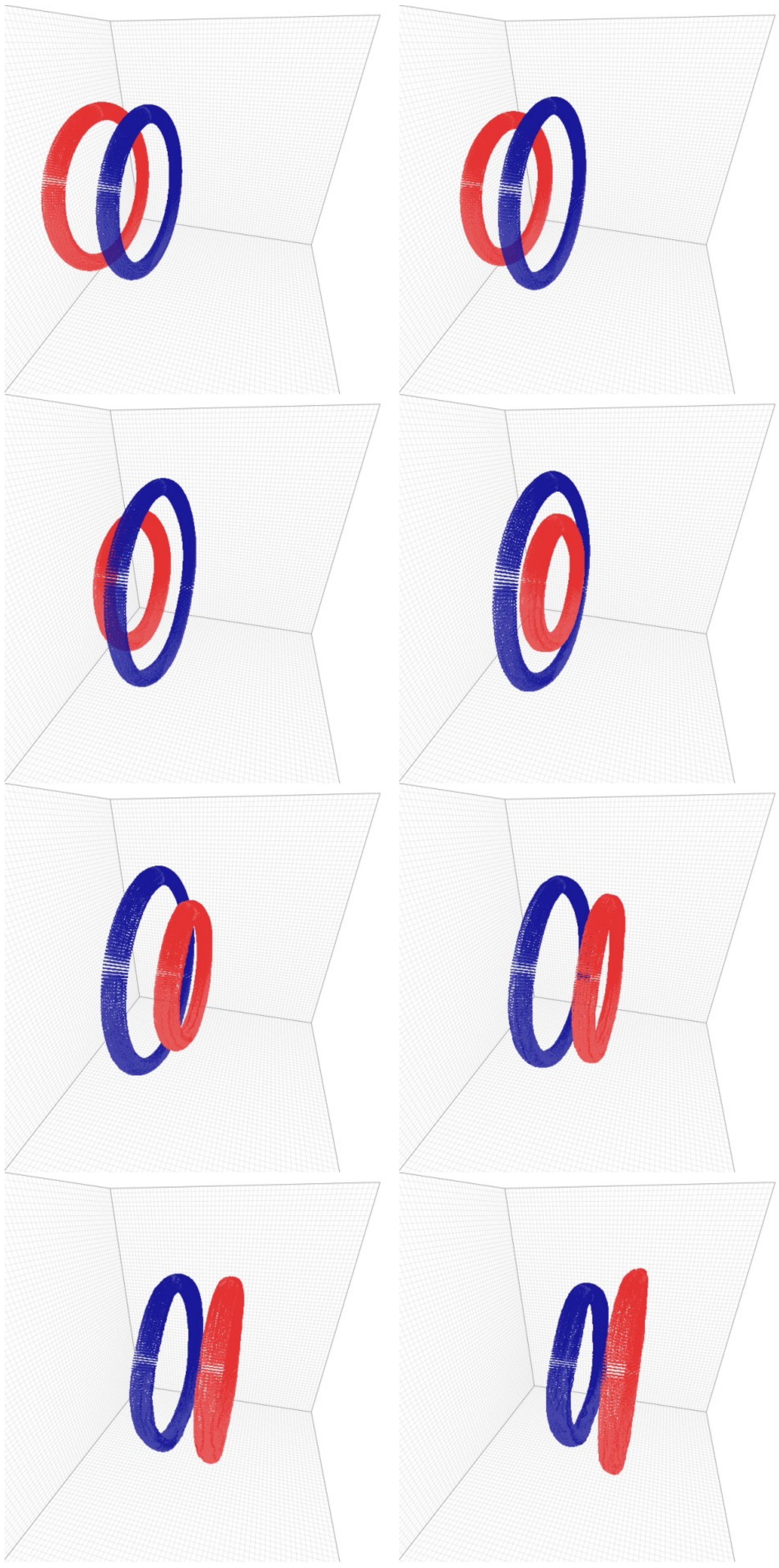

Fig. 6. Vortex game simulation
Rys. 6. Symulacja gry wirów

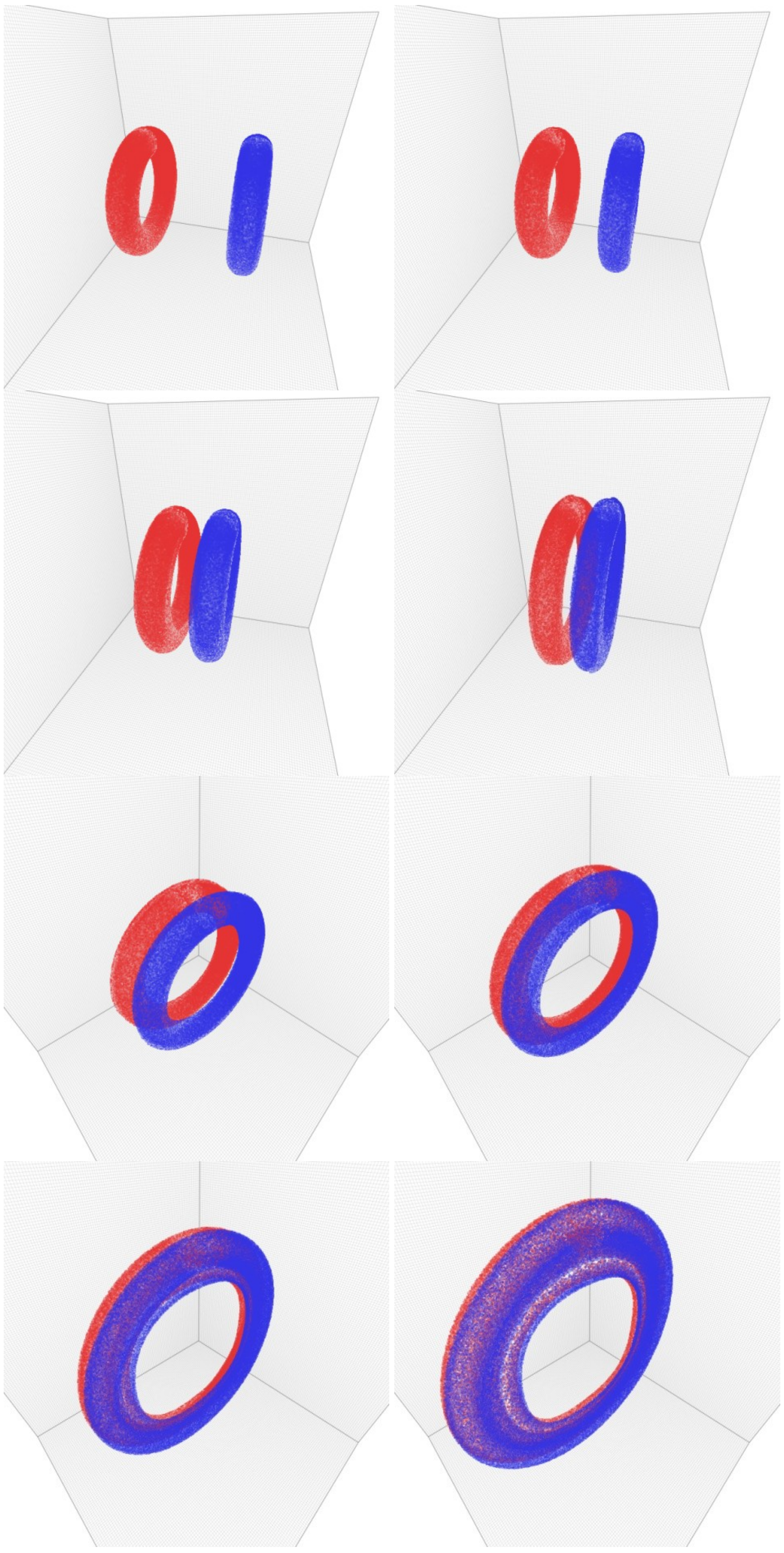

Fig. 7. Vortex ring collision simulation visualization
Rys. 7. Wizualizacja symulacji kolizji pierścieni wirowych

In our simulation we chose two identical vortex rings with radii R = 1.5 and circulation Γ = 1.0 with uniform distribution of vorticity inside the cores, which radius ε = 0.11. The rings were set to travel along *OX* axis with initial distance between their centers Δx = 0.9. The computational domain for selected simulation was a cube of 2Π length per dimension, which in next step was covered by a $64 \times 64 \times 64$ mesh – the spacing between mesh nodes h = 0.098. The time step size for time integrator was chosen as Δt = 0.01.

In Fig. 5 we present the visualization results of vortex game simulations in selected moments. The obtained results are very similar to those reported in [4]. In our work, we managed however to get the rings become less distorted in time, so that just after the t = 5s they don't slowly start to connect. This is due to the fact, that we used a vortex rings with higher resolutions – with 2.5 times more slices and nearly 3 times more particles per ring.

During simulation we unfortunately observe the influence of numerical diffusion on the rings' shape, which slowly gains quadrature distortions. The numerical diffusion is one of the main problems of computational methods that use grid to interpolate values between particles and grid. However, the reduced computational cost is very significant in comparison to the classical methods [1], where computing the velocity at given location in space comes down to the n-body problem. The next simulation example we chose was the head-on vortex rings collision phenomenon, which can be commonly observed in nature. Here we focused more on the visualization side results using free marker particles to visualize fluid elements instead of vortex particles. We used the same rings models as in previous simulation but with radius R = 1.0 and used 65535 marker particles per each vortex ring. The rings were placed on the *OZ* axis with distance Δz = 2.0 between them. As for computational we also used the box with 2Π length per side, but covered it with a $128 \times 128 \times 128$ mesh. The simulation visualization is showed on Fig. 7.

6. Performance

The CPU version was written in C/C++ and was highly optimized using SSE vector instruction set provided by SONY Vectormath library. The Fast Fourier Transform operations were used from FFTW library. Programs were run on Intel Pentium Dual-Core E5200 with 2,5GHz core clock with DDR2 RAM memory running at 800 MHz. As for GPU – the NVIDIA GeForce 460 GTX was used with 756 MB DDR5 RAM memory. The used GPU have 7 Streaming Multiprocessors, where each can handle up to 8 active thread block at time or up to 1546 active threads, running at 1350 MHz core clock. The CUDA kernels were compiled using CUDA Toolkit v.4.0. Although the application was implemented and tested in single GPU environment it can be easily extended adding the support for multi GPU environment.

To analyze the GPU algorithm speedup versus the CPU version we carried on the single vortex ring evolution simulation using different ring configurations (core radius $\varepsilon \approx 0.11$) with mesh sizes which are shown in Tables 1 and 2. The speedup factor was computed as $f = \frac{CPUtime}{GPUtime}$. We only omit here the GPU buffers initialization time and the initial RAM to GPU memory data transfer, as it is done only once at application startup. All the computations on the GPU are done without the need of exchanging the data between GPU memory and external RAM.

In the next part of this chapter we present analysis of achieved speedup results using GPU on selected processing stages – the presented timings are a truncated mean from 100 tests, where 20% of extreme results were discarded.

Table 1

Used computational mesh configurations

Symbol	Mesh size	Total number of nodes
S32	$32 \times 32 \times 32$	$2^{15} = 32768$
S64	$64 \times 64 \times 64$	$2^{18} = 262144$
S128	$128 \times 128 \times 128$	$2^{21} = 2097152$

Table 2

Used vortex ring configurations

Symbol	Number of slices	Particles / slice	Total number of particles
R128	128	25	3200
R256	256	121	30976
R512	512	4089	2093568

6.1. Particle strength distribution

To process particles on GPU we used a thread group of size `{128, 1, 1}`, where each thread processed only one particle – after many tests with different group sizes and particle processing per thread numbers this configuration performed the best. When using this configuration, the GPU multiprocessors can handle 8 active thread groups each. The satisfactory speedup results we achieved only on S128 grid with using R128 and R256 ring models. In other cases the performance difference between the CPU and GPU version was comparable, where on S32 grid and using R128 ring model the CPU version performed even better. The measured processing times are shown in table 3 and on Fig. 8 the speedup factor chart is presented.

This step turned out to be the bottleneck of our algorithm. The achieved speedup results on GPU are not fully satisfactory, which is due to the fact that we used performance expensive atomic operations to update vorticity values in the grid nodes from particles. In the pessimistic case, the atomic operations can highly reduce the GPU parallel processing efficiency, where memory write operations requests to the same memory address are serialized. In order to make better use of GPU highly parallel data processing ability we need to redesign this step and resign from atomic operations usage.

Table 3

Particle strength distribution timing results

Device	Ring	Grid		
		S32	S64	S128
CPU	R128	0.39	2.04	18.91
	R256	1.02	3.90	20.86
	R512	142.09	145.61	163.09
GPU	R128	0.74	0.93	1.23
	R256	2.56	4.23	4.75
	R512	134.33	159.95	181.78

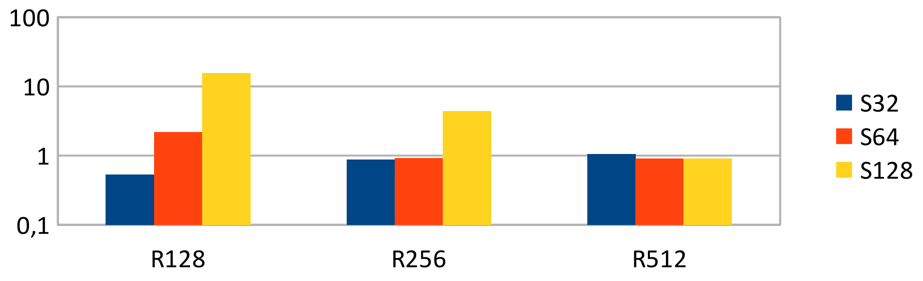

Fig. 8. Particle strength distribution speedup factor *f* chart
Rys. 8. Wykres współczynnika przyspieszenia *f* dystrybucji natężenia cząsteczek

6.2. Solving Poisson and computing velocity with gradients

During Poisson solving and velocity with gradients computing steps the processing times are only dependent on the size of mesh. Theoretically the GPU speedup factor on those steps should improve with increasing the mesh size. In every analyzed case we see a significant advantage of the algorithm on GPU versus the CPU version as expected, where in the case of solving Poisson equation on S128 mesh the GPU version outperforms the CPU version even 60x times. As for GPU configuration, here we used a thread block of size `{512, 1, 1}`, where each work thread processed 4 (S32, S64) or 8 (S128) grid nodes simultaneously. The timing results are shown in Tables 4, 5 and 6, where on Fig.9 the speedup factor chart was presented.

Table 4

Poisson equation solution timing results

Device	Grid		
	S32	S64	S128
CPU	8.29	88.31	858.04
GPU	0.78	2.29	13.48

Table 5

Velocity field computation from vector potential timings

Device	Grid		
	S32	S64	S128
CPU	0.83	6.84	54.91
GPU	0.13	0.28	1.66

Table 6

Velocity field gradients computation timings

Device	Grid		
	S32	S64	S128
CPU	1.16	9.24	76.38
GPU	0.14	0.44	2.92

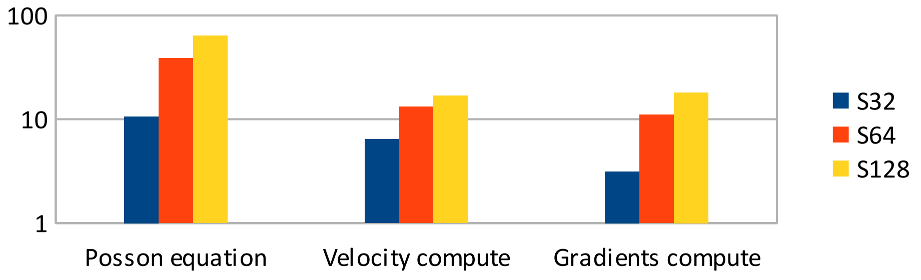

Fig. 9. Speedup factor *f* chart on analyzed steps
Rys. 9. Wykres współczynnika przyspieszenia *f* na wybranych etapach

6.3. Vortex particle update

Here we also achieved satisfactory GPU algorithm speedup results against CPU version, where in case S32 grid and R512 ring model the GPU speedup factor reached 85. As expected, the speedup factor improved with the particle number increase, which is shown on Fig. 10 with timing results in Table 7. Analogously like in distribution step, the GPU thread group size was `{128, 1, 1}` and one particle per thread processing in one time step was set.

The interesting observation was the decrease of speedup factor *f* while increasing the mesh size for analyzed ring model case. The used NVIDIA GeForce GTX 460 is based on the NVIDIA Fermi architecture, in which for the first time on the market the unified memory cache system was implemented, while the traditional graphics chipsets had only texture data memory cache. During performance analysis using NVIDIA Visual Profiler tool, we saw that when interpolating values from grid to particle locations the memory cache hit ratio factor ranged from 90% to 40% and decreased with increasing the resolution of the computational mesh. The variable cache hit ratio situation was connected with the distribution of vortex particles in space and the size of grid cell – when was a high density of particles in a grid cell, which were being processed by multiprocessor, there was a definitely greater probability that the grid node value had been already read and cached.

Table 7

Vortex particle update timings

Device	Ring	Grid		
		S32	S64	S128
CPU	R128	1.28	1.34	1.76
	R256	12.35	12.42	13.66
	R512	839.77	849.91	881.68
GPU	R128	0.31	0.32	0.41
	R256	0.46	0.64	0.98
	R512	9.79	12.69	18.94

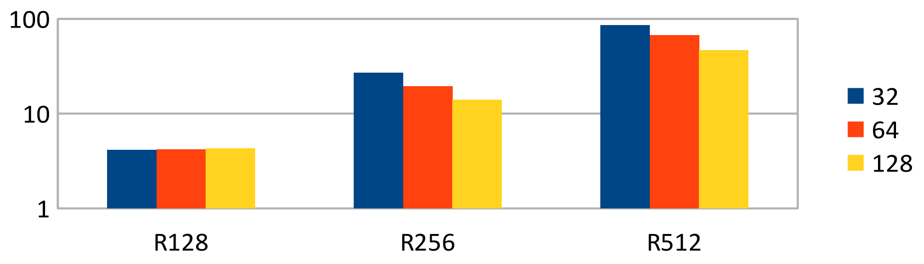

Fig. 10. Vortex particle update speedup factor *f* chart
Rys. 10. Wykres współczynnika przyspieszenia *f* na etapie aktualizacji cząsteczek

6.4. Overall

In Table 8 we present a combined sample timing and GPU speedup results for single vortex ring evolution simulation with using R256 ring model, S128 mesh with 32768 free marker particles.

Table 8

Sample simulation overall timing results

Stage	Timings		GPU speedup *f*
	CPU	GPU	
1. Distribution	20.86	4.75	4.39
2. Poisson	858.04	13.48	63.65
3. Velocity + gradients	131.29	4.58	28.67
4. Particle update	13.66	0.98	13.94
5. Free markes update	6.24	0.59	10.40
Total:	1030.09	24.38	42.25

Practically, on every step we gained a significant GPU processing time advantage over CPU gaining overall average speedup factor *f* = 42.25. In both versions, the most time took the Poisson solving step, which in CPU version represents more than 80% of overall computation time, while solving Poisson equation on the GPU took over 55% of overall processing time.

7. Conclusion

In the presented work the fluid motion modeling using vortex particle method for simulating three dimensional inviscid flows was for the first time implemented on GPU. The method has been validated on different scenarios including leapfrogging vortex rings, vortex rings head-on collision and single vortex ring evolution in time. We achieved satisfactory GPU speedup factor over the CPU version algorithm, where e.g. solving Poisson equation using GPU on $128 \times 128 \times 128$ mesh took 60 times less time than using CPU for computation. To improve the GPU solver performance more, we need to redesign the vorticity distribution algorithm, which showed to be the bottleneck of out method. Implementation of the viscous fluid model and solid boundary conditions would provide a very efficient and comprehensive solution for simulation more realistic three dimensional flows.

Publication was co-financed by the European Union from the EuropeanSocial Fund within the INTERKADRA project UDA –POKL-04.01.01-00-014/10-00.

BIBLIOGRAPHY

1. Cottet G.-H., Koumoutsakos P. D.: Vortex Methods: Theory and Practice. Measurement Science and Technology 12, 2001.
2. Cottet G.-H., Michaux B., Ossia S., VanderLinden G.: 2002. A comparison of spectral and vortex methods in three-dimensional incompressible flows. J. Comput. Phys. 175, 2, (January 2002), p. 702÷712.
3. Schlegel F., Wee D., Ghoniem A. F.: A fast 3D particle method fot the simulation of buoyant flow. J. Comput. Phys. 227, 21 (November 2008), p. 9063÷9090.
4. Kudela H., Regucki P.: Vorticity Particle Method for Simulation of 3D Flow. [in:] International Conference on Computational Science, 2004, p. 356÷363.
5. Harris M. J.: Fast fluid dynamics simulation on the GPU. [in:] Fernando R. (ed.), GPU Gems, vol. 38, Addison Wesley, 2004, p. 637÷665.
6. Crane K., Llamas I., Tariq S.: Real-Time simulation and rendering of 3D fluids. [in:] Nguyen H. (ed.): GPU Gems 3. Addison Wesley Professional, Ch. 30. Aug. 2007.

7. Stock M. J., Gharakhani A.: GPU-accelerated Boundary Element Method and Vortex Particle Method. AIAA 40th Fluid Dynamics Conference and Exhibit (July 2010), p. 1÷12.
8. Rossinelli D., Koumoutsakos P.: Vortex methods for incompressible flow simulations on the GPU. Vis. Comput. 24, 7 (July 2008), p. 699÷708.
9. Rossinelli D., Bergdorf M., Cottet G.-H., Koumoutsakos P.: GPU accelerated simulations of bluff body flows using vortex particle methods. J. Comput. Phys. 229, 9 (May 2010) .
10. Kosior A., Kudela H.: Modelowanie dynamiki pierścienia wirowego metodą cząstek wirowych z wykorzystaniem obliczeń równoległych na kartach graficznych. Modelowanie Inżynierskie, Vol. 13, (September 2012).

Omówienie

W pracy została przedstawiona obliczeniowa metoda modelowania trójwymiarowych przepływów płynów przy użyciu techniki wir-w-komórce skierowana na układy graficzne. Za analizowany ośrodek został przyjęty płyn doskonały o zerowej ściśliwości i lepkości wraz z periodycznymi warunkami brzegowymi, co umożliwiło w dalszym kroku wykorzystanie szybkich metod spektralnych do rozwiązywania różniczkowych równań ruchu płynu. Metoda została z powodzeniem zastosowana do przeprowadzenia symulacji ewolucji pojedynczego pierścienia wirowego, „gry wirów" oraz kolizji pierścieni wirowych. Dodatkowo, dla symulacji „gry wirów" oraz symulacji kolizji pierścieni wirowych zostały zaprezentowane przykładowe wizualizacje, posługując się swobodnymi cząsteczkami markerów. Symulacje zostały przeprowadzone wykorzystując osobno do obliczeń układ graficzny oraz procesor. Testy zostały przeprowadzone dla 3 siatek obliczeniowych o różnych wymiarach oraz dla 3 modeli pierścienia wirowego o różnej budowie i ilości cząsteczek wirowych. Zostały przeprowadzone szczegółowe pomiary czasu trwania symulacji z podziałem na poszczególne jej kroki, uzyskując całkowite średnie przyspieszenia na poziomie 42% na układach graficznych.

Addresses

Łukasz ROGUSKI: Polish-Japanese Institute of Information Technology, ul. Koszykowa 86, 02-008 Warszawa, lucas.roguski@gmail.com

Sebastian DEOROWICZ: Silesian University of Technology, Institute of Informatics, ul. Akademicka 16, 44-100 Gliwice, sebastian.deorowicz@polsl.pl