Acceleration of computing and visualization processes with OpenCL for standing sea wave simulation model

Andrei Ivashchenko, Alexey Belezeko, Ivan Gankevich, Vladimir Korkhov, and Nataliia Kulabukhova

Saint Petersburg State University,
Dept. of Computer Modeling and Multiprocessor Systems,
Universitetskii prospekt 35, Petergof, Saint Petersburg, Russia 198504
aiivashchenko@cc.spbu.ru, alexey.belezeko@gmail.com, i.gankevich@spbu.ru, v.korkhov@spbu.ru, n.kulabukhova@spbu.ru

Abstract. In this paper we highlight one of the possible acceleration approaches for the standing wave model simulation model with the use of OpenCL framework for GPGPU computations. We provide a description of the waves mathematical model, an explanation for the technology selection, as well as the identification of the algorithm part that can be accelerated. The text also contains a description of solutions performance evaluation stage being compared with CPU-only program. The influence of OpenCL usage for improvements in rendering process is also shown here. Finally, possible ways of application improvement and further development are also considered.

Keywords: Computing, Mathematical modelling, OpenCL, OpenGL, Autoregressive process, Moving average process, Velocity potential field, Visualisation, Real-time simulation

1 Introduction

In most cases, visualisation of scientific data obtained during simulation or computation process is carried out separately, after all the stages of calculation are completed. This fact is connected with a rather large number of factors: computation process could be executed with CPU-only nodes (however, this should not be considered as a problem since the Mesa 3D graphics library exists); the goal to complete a task as fast as possible could have greater priority, thus, all available resources would be used for this; data could be just difficult to process and synchronise during in runtime, if such a scenario was not assumed by the software solution.

However, on the other hand, during the computation process, especially the long one, there is a need to monitor for performed operations and obtained results. In the case where such a control is possible the calculations could be suspended and the necessary tweaks made with a timely response. This kind

of action may also be needed while debugging the program or testing a new mathematical model. Thus, in this article we will consider the possibility of interactive control organisation for calculations with a terms of visualisation, which will be performed using the OpenGL API.

Another possible scenario of this approach is an interaction with simulated objects and processes in real-time. So, you can change the initial conditions, add new environmental parameters and observe the system's response immediately from the moment of the effect's influence beginning to its end. In the framework of ocean wave simulation this has educational value, as the effect of every change in the input parameter is immediately visible. In addition to this, instantaneous visualisation of ocean wavy surface brings simulation to a new level, where dynamically changing parameters to arbitrary values within predefined ranges allows to visually verify the model and its numerical code.

For the experiment we have chosen an autoregressive model of standing waves within framework of which we have accelerated velocity potential field computation with the usage of GPGPU technology through the OpenCL framework. Since, data structures that are needed for visualisation are already stored in GPU memory, we take into account that fact and remove unnecessary copying between host and device using OpenGL/OpenCL interoperability API.

2 Related work

The idea of mixing various computing APIs is not a fresh one, also including for OpenGL/OpenCL interoperability. Nvidia has announced the support for this technology in the 2011, and since then we have been able to observe the related solutions appeared on the market.

The idea of compute API usage was widely spread and adopted by entertainment industry, especially in game development sphere. Ever since the popularisation of the PhysX engine, which uses the capabilities of graphics cards to simulate a certain set of physical phenomena, it was clear that such a technology will find a place to be applied in the future [9]. So, for today, almost all heavy dynamic particle systems you can met are using one of the general compute APIs [15].

Another area of general usage where this technology was introduced is a computer vision. Industry standard library OpenCV has a special OCL module originally provided by AMD, which enables the acceleration for various algorithms, including ones for matrix transformations, image filtering and processing, object detection and many more [3].

However, the situation with scientific calculation is absolutely different. Unlike the entertainment software where the vast of the scene contents in most cases are generated in advance and the number of processes is strictly limited to ones affecting the environment at the current moment, for the scientific simulations almost everything should be calculated from scratch based only on given initial conditions, including the geometry (if it is meant by the process), the particle system, visual effects, etc. In addition, simulations by themselves are much more

complicated, and the optimisation for dynamically forming geometric structures is quite difficult to perform.

There is a way to achieve more efficient usage of resources while performing the visualisation of the computational experiment results, which are involving GPGPU to speed up the computation, and it is directly related to the accelerator exploitation. Thus, when graphics cards are used to compute, it is obvious that the data is already allocated on its memory. So, if there is a way to transform the data to the format that can be used by the graphic API, and also the way to transfer it between the compute and graphics contexts, then we will not have to copy it from the memory of GPU to the RAM and vice versa. Thus, the usage of OpenCL, CUDA, or any other compute API can boost the performance not only for the calculations themselves, but also for the results rendering. Thus, we have reviewed several papers which are referring to the stated problem to get a glance whether this approach could be used for the optimisation in our case.

One of the most interesting cases was shown by the research group from Boston Northeastern University [14]. The OpenCL/OpenGL interoperability was applied to five completely different applications related to the different study areas. For example, one of them is a Material Fault Detection program used for fault detection using wave propagation in anisotropic materials, which produces material layer surfaces. They have used a slot-based rendering technique, which means that data is precomputed for several frames in advance before passing it to the rendering context. As a result, they have been able to obtain 2.2 more frames in average with discrete AMD GPUs Radeon 7770 and Radeon 7970 and 1.9 more with AMD Fusion A8 APU.

As for the image processing, we can refer to [12], where the interop technique is used for panorama video image stitching. According to the provided metrics the best result is achieved with involving two buffers for both OpenCL computations and OpenGL rendering, and it is 12 times faster compared to the original CPU based solution. The paper is also showing that the proposed solution is scaled really well when the additional image capturing devices are attached to the system. We should also notice an another closely related case presented by Samsung at SIGGRAPH'13, which talks in general about real time video stream processing on mobile platforms captured by camera module with the usage of OpenCL and OpenGL ES interoperability [4].

All the examples of GPGPU API interoperability mentioned above are proving that the proposed approach could be applied for the various set of problems to achieve the significant results in optimisation of calculations and visualisation itself. However, it should be said that the following solutions has been applied for the particular cases and not showing any general solution, which could be treated as a specificity of the interoperability method. During the study we will try to point out the major improvements made by research groups to complete our solution in a most optimal way.

3 Compute and graphics contexts interoperability

OpenGL itself does not contain any mechanisms that could help to organise the interaction between OpenGL and OpenCL. However, despite the fact that the specified functionality is not available, OpenGL supports the data and message exchange between its own contexts, the fundamental principles of which are laid in CL/GL interoperability [1]. Thus, it would be expediently to consider on this question as it represents the basics we need to understand.

First, the graphics card, which is intended to be used for computation, should be checked for OpenCL shared context mode support. To find this out, the `clinfo` command line utility should be run on the target machine. If the required functionality is supported the `cl_khr_gl_sharing` option will be specified in the "Device Extensions" section [10]. This extension is provided by Khronos group and it is responsible for the interaction between APIs.

The following extension contains all necessary functions for OpenCL, which are defined in `cl_gl.h` header file. In general, they could be divided into three main groups:

- Memory broker functions — acquire and release allocated memory areas represented with OpenGL objects;
- Object transformation functions — create an OpenCL representation for OpenGL object;
- Info functions — provide various information about OpenGL context, like associated devices, object description, etc.

The next thing that should be discussed is data types, which could be driven by the interoperability API. This issue has been partially reviewed in [2] at the discussion about graphics API parallelization. Basically, all OpenGL object are represented with two groups. The first one is a Container Object group the specificity of which lies in the fact that they can not be shared between contexts since they contain references to other objects, and GL standard disallow transfer for objects of that type, i.e. they are not a point of our interest. Another group contains regular Objects, which could be shared between context without any limitations. Since we need to share only the data for now, we should take a look for the Buffer Object, which could actually store an array of unformatted memory allocated by the OpenGL context. Among all Buffer Objects, for the our particular case the Vertex Buffer Object should be used at first to transfer the surface representation data.

Due to the fact that graphics unit could proceed with the single context at once, we will need to manage them manually by alternating them upon request. Following this way we should be able to build a pipeline as it presented in fig. 1. Here, both contexts controlled by the main process are basically performing eight main steps to achieve the goal:

1. First, check whether the `cl_khr_gl_sharing` is supported by the target GPGPU.

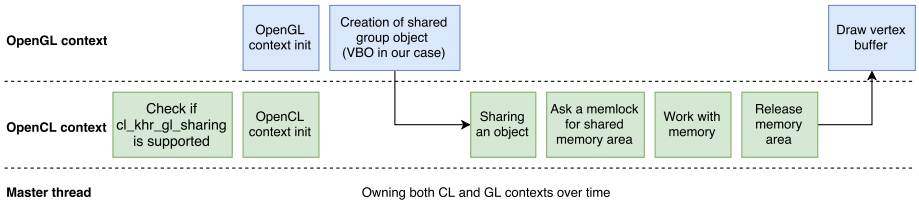

Fig. 1. OpenGL/OpenCL interoperability pipeline.

2. If the previous step succeeds, initialise both compute and graphics contexts.
3. Next, the shared object is created by OpenGL. It will be used later on to pass the data back to graphics context.
4. Then, the control passes to the OpenCL context and the object is registered here for the sharing.
5. Apply a lock for the memory area acquired by the object.
6. Perform all required computations and write the results to the shared memory.
7. Release the lock and pass control to the graphics API.
8. Finally, process and draw geometry.

4 Standing sea waves simulation model

Our approach to sea waves simulation is based on the autoregressive model—moving average (ARMA) model of sea waves [6, 7]. This model was developed as an superior alternative to existing linear Longuet—Higgins model. The new model simulates sea waves without assumptions of linear and small amplitude wave theories, i.e.

- the model generates waves of arbitrary amplitudes,
- period of wavy surface realisation equals the period of pseudo-random number generator (PRNG) and
- it requires less number of coefficients to converge compared to Longuet—Higgins model.

This model allows to generate both propagating and standing sea waves via moving average and autoregressive process respectively, but for the purpose of this paper we narrow the discussion to standing waves and autoregressive (AR) process only.

One implication of turning down the assumptions of linear wave theory is that it is not possible to use linear velocity potential field computation formulae for the new wavy surface, as they were derived under the same assumptions. As a result, the new analytic formula was derived, that determines velocity potential field under arbitrary wavy sea surface. This formula is particularly suitable for computation on GPUs:

– it contains transcendental mathematical functions (hyperbolic cosines and complex exponents);
– it is computed over large four-dimensional (t, x, y, z) region;
– it is analytic with no information dependencies between individual data points in t and z dimensions.

Moreover, for the purpose of the verification of the resulting wavy surface, it is imperative to visualise the surface and velocity potential velocity field in real-time as the computation progresses. Performing two simulations at a time with different velocity potential field formulae allows to spot the difference in computed fields, and to visually compare the size and the shape of regions where the most wave energy is concentrated.

Within the framework of autoregressive model for standing waves we investigate how GPGPU computations can be used to speed-up velocity potential field computation and make real-time visualisation of the surface as computation proceeds.

4.1 Governing equations for 3-dimensional AR process

Three-dimensional autoregressive process is defined by

$$\zeta_{i,j,k} = \sum_{l=0}^{p_1} \sum_{m=0}^{p_2} \sum_{n=0}^{p_3} \Phi_{l,m,n} \zeta_{i-l,j-m,k-n} \epsilon_{i,j,k},$$

where ζ — wave elevation, Φ — AR coefficients, ϵ — white noise with Gaussian distribution, (p_1, p_2, p_3) — AR process order, and $\Phi_{0,0,0} \equiv 0$. The input parameters are AR process coefficients and order.

The coefficients Φ are calculated from ACF via three-dimensional Yule—Walker equations:

$$\Gamma \begin{bmatrix} \Phi_{0,0,0} \\ \Phi_{0,0,1} \\ \vdots \\ \Phi_{p_1,p_2,p_3} \end{bmatrix} = \begin{bmatrix} K_{0,0,0} - \sigma_\epsilon^2 \\ K_{0,0,1} \\ \vdots \\ K_{p_1,p_2,p_3} \end{bmatrix}, \qquad \Gamma = \begin{bmatrix} \Gamma_0 & \Gamma_1 & \cdots & \Gamma_{p_1} \\ \Gamma_1 & \Gamma_0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \Gamma_1 \\ \Gamma_{p_1} & \cdots & \Gamma_1 & \Gamma_0 \end{bmatrix},$$

where $\boldsymbol{N} = (p_1, p_2, p_3)$, σ_ϵ^2 — white noise variance, and

$$\Gamma_i = \begin{bmatrix} \Gamma_i^0 & \Gamma_i^1 & \cdots & \Gamma_i^{p_2} \\ \Gamma_i^1 & \Gamma_i^0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \Gamma_i^1 \\ \Gamma_i^{p_2} & \cdots & \Gamma_i^1 & \Gamma_i^0 \end{bmatrix} \qquad \Gamma_i^j = \begin{bmatrix} K_{i,j,0} & K_{i,j,1} & \cdots & K_{i,j,p_3} \\ K_{i,j,1} & K_{i,j,0} & \ddots & x \quad \vdots \\ \vdots & \ddots & \ddots & K_{i,j,1} \\ K_{i,j,p_3} & \cdots & K_{i,j,1} & K_{i,j,0} \end{bmatrix},$$

Since $\Phi_{0,0,0} \equiv 0$, the first row and column of Γ can be eliminated. Matrix Γ is block-toeplitz, positive definite and symmetric, hence the system is solved by Cholesky decomposition. White noise variance is estimated by

$$\sigma_\epsilon^2 = K_{0,0,0} - \sum_{i=0}^{p_1} \sum_{i=0}^{p_2} \sum_{k=0}^{p_3} \Phi_{i,j,k} K_{i,j,k}.$$

4.2 Three-dimensional velocity potential field

The problem of finding pressure field under wavy sea surface represents inverse problem of hydrodynamics for incompressible inviscid fluid. System of equations for it in general case is written as [11]

$$\begin{aligned} &\nabla^2\phi = 0, \\ &\phi_t + \frac{1}{2}|\boldsymbol{v}|^2 + g\zeta = -\frac{p}{\rho}, && z = \zeta(x,y,t), \\ &D\zeta = \nabla\phi \cdot \boldsymbol{n}, && z = \zeta(x,y,t), \end{aligned} \tag{1}$$

where ϕ — velocity potential, ζ — elevation (z coordinate) of wavy surface, p — wave pressure, ρ — fluid density, $\boldsymbol{v} = (\phi_x, \phi_y, \phi_z)$ — velocity vector, g — acceleration of gravity, and D — substantial (Lagrange) derivative. The first equation is called continuity (Laplace) equation, the second one is the conservation of momentum law (the so called dynamic boundary condition); the third one is kinematic boundary condition for free wavy surface, which states that rate of change of wavy surface elevation ($D\zeta$) equals to the change of velocity potential derivative along the wavy surface normal ($\nabla\phi \cdot \boldsymbol{n}$).

Inverse problem of hydrodynamics consists in solving this system of equations for ϕ. In this formulation dynamic boundary condition becomes explicit formula to determine pressure field using velocity potential derivatives obtained from the remaining equations. So, from mathematical point of view inverse problem of hydrodynamics reduces to Laplace equation with mixed boundary condition — Robin problem.

Three-dimensional version of (1) is written as

$$\begin{aligned} &\phi_{xx} + \phi_{yy} + \phi_{zz} = 0, \\ &\zeta_t + \zeta_x\phi_x + \zeta_y\phi_y = \frac{\zeta_x}{\sqrt{1+\zeta_x^2+\zeta_y^2}}\phi_x + \frac{\zeta_y}{\sqrt{1+\zeta_x^2+\zeta_y^2}}\phi_y - \phi_z, \quad z = \zeta(x,y,t). \end{aligned}$$

Using Fourier method with some assumptions the equation is solved yielding formula for ϕ:

$$\phi(x,y,z,t) = \mathcal{F}^{-1}_{x,y}\left\{ \frac{\cosh\left(2\pi|\boldsymbol{k}|(z+h)\right)}{2\pi|\boldsymbol{k}|\cosh\left(2\pi|\boldsymbol{k}|h\right)} \mathcal{F}_{u,v}\left\{ \frac{\zeta_t}{\left(if_1(x,y) + if_2(x,y) - 1\right)} \right\} \right\}, \tag{2}$$

where $f_1(x,y) = \zeta_x/\sqrt{1+\zeta_x^2+\zeta_y^2} - \zeta_x$ and $f_2(x,y) = \zeta_y/\sqrt{1+\zeta_x^2+\zeta_y^2} - \zeta_y$.

4.3 Architecture

Incorporation of OpenCL version of velocity potential solver into the existing source code reduced to an addition of two subclasses (fig. 2):

`Realtime_solver` a subclass of abstract solver that implements computation of velocity potential field on GPU, and

`ARMA_realtime_driver` a subclass of control flow object — an object that defines the sequence of calls to subroutines — that implements real-time visualisation and stores OpenGL buffer objects that are shared with the solver. These objects are shared with the solver only if the solver is real-time.

The algorithm for computation and visualisation pipeline is presented in alg.1.

```
Initialise shared OpenCL/OpenGL context.
Generate the first wavy surface realisation time slice.
Compute corresponding velocity potential field.
while not exited do
    Visualise the current slice of wavy surface and velocity field.
    Asynchronously compute next wavy surface time slice.
    Compute its velocity potential field.
end
```

Algorithm 1: Main loop of computation and visualisation pipeline.

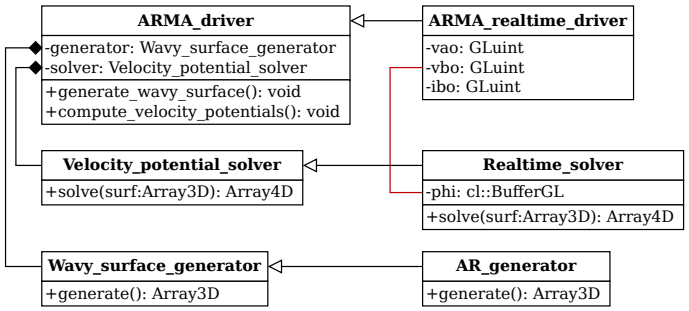

Fig. 2. Classes which implement OpenCL/OpenGL interoperability in the simulation code.

5 Evaluation

For the purpose of evaluation we use simplified version of (2):

$$\begin{aligned}\phi(x,y,z,t) &= \mathcal{F}^{-1}_{x,y}\left\{\frac{\cosh\left(2\pi|\boldsymbol{k}|(z+h)\right)}{2\pi|\boldsymbol{k}|\cosh\left(2\pi|\boldsymbol{k}|h\right)}\mathcal{F}_{u,v}\{\zeta_t\}\right\}\\ &= \mathcal{F}^{-1}_{x,y}\{g_1(u,v)\mathcal{F}_{u,v}\{g_2(x,y)\}\}\,. \end{aligned} \tag{3}$$

Since standing sea wave generator does not allow efficient GPU implementation due to autoregressive dependencies between wavy surface points, only velocity potential solver was rewritten in OpenCL and its performance was compared to existing OpenMP implementation.

For each implementation the overall performance of the solver for a particular time instant was measured. Velocity field was computed for one t point, for 128 z points below wavy surface and for each x and y point of four-dimensional (t, x, y, z) grid. The only parameter that was varied between subsequent programme runs is the size of the grid along x dimension. A total of 10 runs were performed and average time of each stage was computed.

A different FFT library was used for each version of the solver. For OpenMP version FFT routines from GNU Scientific Library (GSL) [8] were used, and for OpenCL version clFFT library [5] was used instead. There are two major differences in the routines from these libraries.

1. The order of frequencies in Fourier transforms is different and clFFT library requires reordering the result of (3) whereas GSL does not.
2. Discontinuity at $(x, y) = (0, 0)$ of velocity potential field grid is handled automatically by clFFT library, whereas GSL library produce skewed values at this point.

For GSL library an additional interpolation from neighbouring points was used to smooth velocity potential field at these points. We have not spotted other differences in FFT implementations that have impact on the overall performance.

In the course of the numerical experiments we have measured how much time each solver's implementation spends in each computation stage to explain find out how efficient data copying between host and device is in OpenCL implementation, and how one implementation corresponds to the other in terms of performance.

6 Results

The experiments showed that GPU implementation outperforms CPU implementation by a factor of 10–15 (fig. 3), however, distribution of time between computation stages is different for each implementation (fig. 4). The major time consumer in CPU implementation is computation of g_1, whereas in GPU implementation its running time is comparable to computation of g_2. GPU computes g_1 much faster than CPU due to a large amount of modules for transcendental mathematical function computation. In both implementations g_2 is computed on CPU, but for GPU implementation the result is duplicated for each z grid point in order to perform multiplication of all XYZ planes along z dimension in single OpenCL kernel, and, subsequently copied to GPU memory which severely hinders overall stage performance. Copying the resulting velocity potential field between CPU and GPU consumes $\approx 20\%$ of velocity potential solver execution time.

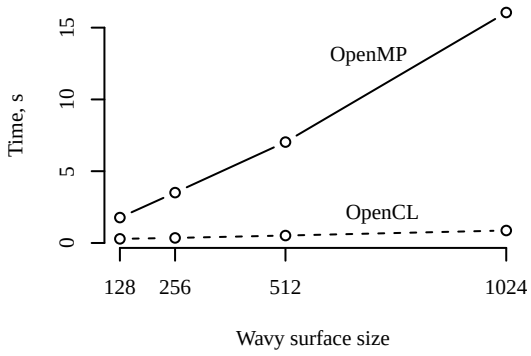

Fig. 3. Performance comparison of CPU (OpenMP) and GPU (OpenCL) versions of velocity potential solver.

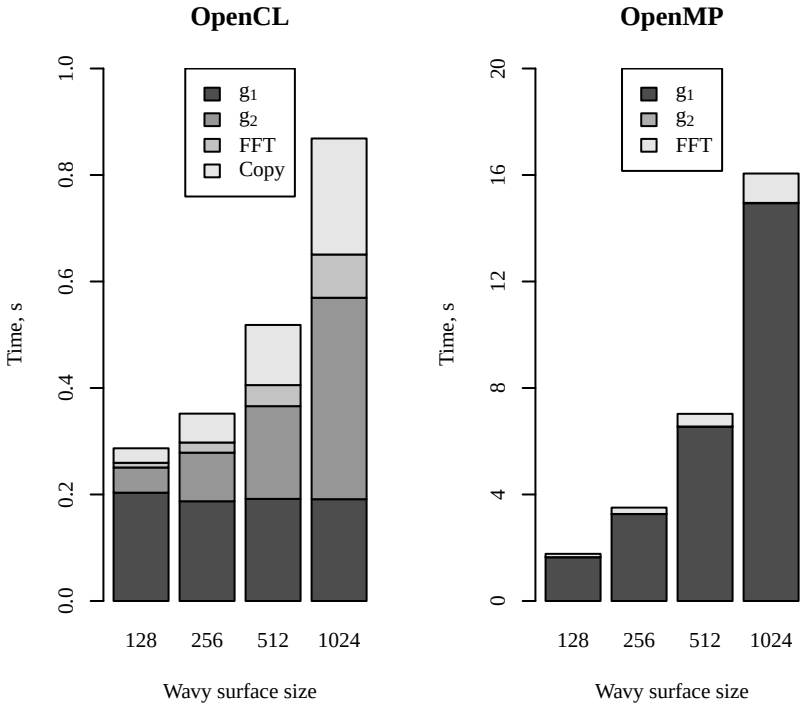

Fig. 4. Performance breakdown for GPU (OpenCL) and CPU (OpenMP) versions of velocity potential solver.

7 Discussion and future work

Simplified velocity potential formula (3) lacks $f_{1,2}$ functions which contain spatial derivatives of wavy surface ζ which are inefficient to compute on GPU. The remaining derivative ζ_t is also computed on CPU for the sake of efficiency. The future work is to find high-performance algorithm for multidimensional derivative computation on GPUs.

OpenGL is still a single-thread library and in most cases of graphics applications only one thread manages the access to the GL context. This could lead to the performance drops, thread interoperability issues and complicated application architecture. Some workarounds could be found yet, like launching multiple processes and switching contexts between them, but it is not solving the problems mentioned about really. Thus, there are some premises exist, which are making sense to investigate on the similar result achievement with the newer APIs, such as a Vulkan and DirectX 12.

Relatively the same statement could be made about the OpenCL toolkit. Despite the fact that general GPGPU computing interface allowed us to achieve some improvements through the waveform computation acceleration and and made a cross-platform execution possible, it still has some disadvantages. E.g., each core should be manually cached after the compilation, which is exactly proceeded over execution time. Moreover, it could happen that other compute APIs could show even better performance rates.

We can also consider experimenting with dual chip GPU boards. The main idea here is to provide for both computing and rendering contexts an exclusive right to use their own dedicated GPU core in every moment of time. This improvement can help for the cases where intensive rendering routines are expected to be applied for the data obtained during the computations. In that way we will not lose the performance and, probably, will achieve even better results.

And the most interesting question here is what should be done when the computing capabilities of the single node will be reached, or simply how do we scale. The part of application connected with direct computations could be handled in a common way by the one of MPI implementations. This method is compatible for both CUDA and CPU driven processes. Some concerns could appear at the stage of visualisation scaling.

The first and simplest way to achieve the desired result is to accumulate data from the nodes after each computation round on the master node to join results and visualise them on it. But if we will make it like this we will lose the benefits of using a shared buffer for OpenCL and OpenGL. In addition, as the number of nodes will increase the bandwidth requirements of the channel will grow too, and in the end we will rest against its limitations.

The second possible method of achieving the result is similar to the first one and has the same problems, but involves the usage of GPUDirect or DirectGMA technology for graphics cards depends on their vendor. The only difference here is that the exchange of data between nodes will be based on peer-to-peer protocol under CUDA context. It means that GPU will receive messages directly.

The last possible way to solve this problem is to use distributed rendering techniques. It is based basically on combination of load distribution and objects or images composition algorithms, where three main ones can be distinguished: sort-first, sort-middle and sort-last [13].

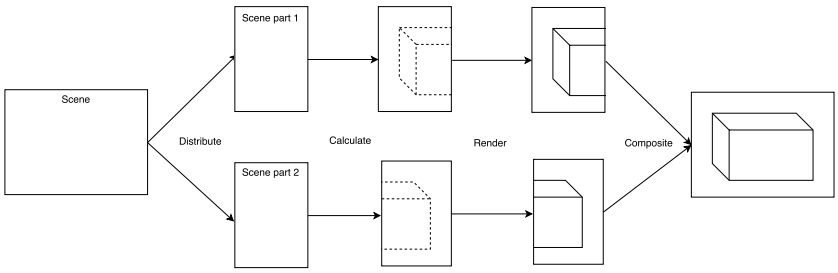

Fig. 5. Sort-first image compositing.

Sort-first (fig. 5) assumes that the scene should be divided into a number of zones, each processed on its own hardware. In fact, several cameras/viewports are created in the OpenGL context on different nodes, and then resulting frame buffers are simply merged into a single final frame. Two main problems that could be met here, but can be solved in some way, are artefacts at the joints of frame parts when using lossy compression and desynchronization of frame pieces when there is some motion on the scene.

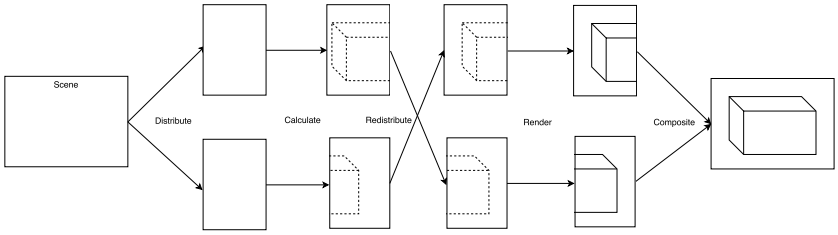

Fig. 6. Sort-middle image compositing.

Sort-middle (fig. 6) is not widely used in real time applications, due to the fact that it takes an extra time to produce and brings network overheads, but it is the most effective one. First, each node calculates geometry, and then workload is getting redistributed equally among all computing devices.

Both methods described above do not really suit us well, since each of the nodes will process either a single frame, or a part of the total surface, and we will not involve a viewport in a such way, at least for now.

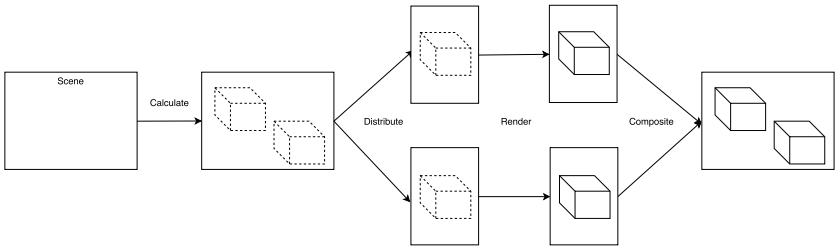

Fig. 7. Sort-last image compositing.

Actually, we are interested in the sort-last method (fig. 7) most of all. Unlike the previous two, the load distribution here is performed based on the 3D objects grouping and segmentation. Later on, alpha blending of rendered objects is performed to retain the resulting frame. This way may not be optimal in some cases, since nodes will have to render parts of objects that will be overlapped at the compositing stage, but still it allows to achieve the desired result.

We can even make some optimisation steps here. For example, to reduce the load on the network channel, we can use an N-ary compositing. In other words, instead of joining all objects on the only node, we can exchange objects between N nodes first and connect them. For example, it can be done on the principle of a binary tree. We can also use various compression algorithms, which show the greatest performance for colour and numeric data.

8 Conclusion

As a result of the investigation, we managed to achieve positive outcome in computation acceleration and its visualisation using interaction mechanisms between the graphics and compute contexts. The joint use of OpenGL and OpenCL allowed us to simultaneously use the shared areas of dedicated GPU memory for calculation and rendering, thereby saving us some time required to copy the data for RAM to the video card memory. Visualisation of real-time calculation results allowed to perform fine-grained control for the process showing us new opportunities for simulation with variable conditions.

Proposed solution could be applied not only for the ocean wavy surface simulation, but for any other iterative computations, which are producing a sane amount of data. For example, we can reuse the result in a similar manner for the costs prediction of monetary assets.

We have also determined the list of possible tasks that could be performed in the future to improve the functionality of the software solution. One of the directions is to experiment on combining various graphics and computing APIs to identify the most optimal solution.

Acknowledgements

Research was supported by grants of Russian Foundation for Basic Research (projects no. 16-07-01111, 16-07-00886, 16-07-01113).

References

1. Board, O.A.R.: Opengl Reference Manual: The Official Reference Document for Opengl, Release 1 (OTL). Addison-Wesley (C) (1993)
2. Bogdanova, A., Ivashchenko, A., Belezeko, A.: Creating distributed rendering applications. In: CEUR Workshop Proceedings. vol. 1787, pp. 130–134 (2016), http://ceur-ws.org/Vol-1787/130-134-paper-21.pdf
3. Bradsky, G.: The opencv library. Dr. Dobb's Journal of Software Tools (2000)
4. Bucur, A.: OpenCL – OpenGL ES interop: Processing live video streams on a mobile device – case study. In: ACM SIGGRAPH 2013 Mobile. p. 15. SIGGRAPH'13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2503512.2503532
5. clFFT developers: clFFT: OpenCL Fast Fourier Transforms (FFTs). https://clmathlibraries.github.io/clFFT/
6. Degtyarev, A., Reed, A.: Modelling of incident waves near the ships hull (application of autoregressive approach in problems of simulation of rough seas). In: Proceedings of the 12th International Ship Stability Workshop (2011)
7. Degtyarev, A.B., Reed, A.M.: Synoptic and short-term modeling of ocean waves. International Shipbuilding Progress 60(1-4), 523–553 (2013)
8. Galassi, M., Davies, J., Theiler, J., Gough, B., Jungman, G., Alken, P., Booth, M., Rossi, F., Ulerich, R.: GNU Scientific Library Reference Manual. Network Theory Ltd., 3 edn. (2009), eds. Brian Gough
9. Geer, D.: Vendors upgrade their physics processing to improve gaming. Computer 39(8), 22–24 (Aug 2006)
10. Group, K.: Opencl api reference (2013)
11. Kochin, N., Kibel, I., Roze, N.: Theoretical hydrodynamics [in Russian]. FizMatLit (1966)
12. Liao, W.S., Hsieh, T.J., Chang, Y.L.: Gpu parallel computing of spherical panorama video stitching. In: 2012 IEEE 18th International Conference on Parallel and Distributed Systems. pp. 890–895 (Dec 2012)
13. Molnar, S., Cox, M., Ellsworth, D., Fuchs, H.: A sorting classification of parallel rendering. IEEE computer graphics and applications 14(4), 23–32 (1994)
14. Ukidave, Y., Gong, X., Kaeli, D.: Performance evaluation and optimization mechanisms for inter-operable graphics and computation on gpus. In: Proceedings of Workshop on General Purpose Processing Using GPUs. pp. 37–45. GPGPU-7, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2576779.2576784
15. Unity Technologies: Unity - Manual: Compute Shaders. https://docs.unity3d.com/Manual/ComputeShaders.html