Numerous problems in physics require huge amount of computations, and up until current decade only supercomputers were able to provide enough computation power. During the last few years general-purpose graphic processing units (GPGPU, commonly referred to as video adapters) provide a supercomputer-class performance. We consider GPGPU application programming interfaces, and provide an example of one-dimensional linear wave equation solution with the use of OpenCL API. We also provide the results of a performance test of few GPGPU.

Introduction

Various numerical problems in physics lead to huge yet parallelizible computation. Almost every problem described via linear equations system, partial differential equations, Monte-Carlo methods is subject to parallel computation. While x86-based supercomputers are commonly used for these problems, they are quite expensive. Therefore, a cheap alternative is required.

NVidia corporation was the first who started marketing such an alternative. The introduced CUDA technology was a way to exploit computation power of video adapters for any general application, thus the term GPGPU has appeared. CUDA is available only for nVidia hardware, and possesses a mature toolkit, including a debugger, profiler, BLAS and Fourier transform libraries.

Meanwhile, OpenCL [1], an open standard for GPGPU application programming interface (API) was developed. It was adopted by AMD company, as AMD’s own API was not successful, and, up to now it has every major architecture support. OpenCL standard defines a cross-platform library, with the support for heterogeneous computing, OpenGL compatibility, however it lacks special-purpose computation packages, as those which are supplied with CUDA. Fortunately, AMD provides this software within its OpenCL API. NVidia GPGPU also supports OpenCL, while CUDA perform faster on their hardware. CPUs that feature SSE3 command set can also execute OpenCL programs, while various ARM and IBM PPC architectures have beta status of OpenCL support. OpenCL and CUDA APIs are compared in table 1. We generally believe that OpenCL is the best choice for newly developed software, however CUDA is a more matured technology.

<table>
<thead>
<tr>
<th>Feature</th>
<th>nVidia CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development facilities</td>
<td>debugger, profiler, linear algebra libraries</td>
<td>debugger</td>
</tr>
<tr>
<td>Availability on HPC</td>
<td>wide</td>
<td>uncommon</td>
</tr>
<tr>
<td>Architectures supported</td>
<td>restricted to nVidia</td>
<td>nVidia, AMD, x86 with SSE3, ARM, PPC</td>
</tr>
<tr>
<td>Heterogeneous computing</td>
<td>officially unsupported</td>
<td>supported</td>
</tr>
</tbody>
</table>

As the result, GPGPU became a cheap way for massively parallel computing. Moreover, modern top-level single-processor video adapters provide computational power of early-2000s top-level supercomputers. In the figure 1 there is a performance plot for top-level supercomputers, and top-level GPGPU. One of the fastest x86 CPUs is also shown. Finally, we compare a supercomputer to a video adapter of the same performance in the table 2.
Despite the advantages of GPGPU, they are slowly adopted for computational physics purposes and are still mainly used to speed up Monte Carlo simulations in high energy physics. Implementing Monte Carlo simulations on GPU is natural, because it is obvious how to parallelize the problem [4]. A plenty of computational physics tasks involve solution of PDE, and solving PDEs may be significantly harder. The purpose of this paper is to illustrate how one can easily implement widespread approaches to PDE solution on GPGPU with OpenCL API. We discuss explicit finite-difference method for heat equation in the first chapter. In the second chapter we consider method of lines for a wave equation.
OpenCL implementation of finite-difference method for heat equation.

We discuss the heat equation:

\[
\frac{\partial u}{\partial t} - A \frac{\partial^2 u}{\partial x^2} = 0,
\]

with imposed boundary conditions

\[
u(x,t)|_{t=0} = u_0(x), \quad u(x,t)|_{x=0,x=a} = u_0(0) = u_0(a) = u_b,
\]

where \( A \) and \( u_b \) are constants and \( u_0(x) \) is an arbitrary function within given constraints.

First and the most naive approach is to replace all derivatives with appropriate finite differences; this leads to either explicit or implicit methods. The former ones have convergence issues, the latter ones lead to many-diagonal matrix linear equation systems for every time step [5]. However, we will regard the explicit method just for a purpose of a simple example.

We introduce time step \( \Delta t \), divide the coordinate with \( N \) points, thus, coordinate step is \( \Delta x = a/(N - 1) \). We also replace \( u(x,t) \) with its grid values \( u_{i,j} = u(i\Delta x,j\Delta t) \). Finally, after replacing derivatives in (1) we get:

\[
\frac{u_{i,j+1} - u_{i,j}}{\Delta t} - A \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} = 0.
\]

Explicitly expressing the term \( u_{i,j+1} \) from (3) we obtain the scheme:

\[
u_{i,j+1} = (1 - 2\alpha)u_{i,j} + \alpha(u_{i+1,j} + u_{i-1,j}),
\]

where \( \alpha = (A\Delta t)/\Delta x^2 \).

It’s obvious that points \( u_{i,j+1} \) can be obtained from scheme (4) in any order of space index \( i \), thus this calculations can be done in parallel. We implement this idea into the OpenCL kernel source code, given on image 1.

```c
#include "task.h"
{
    const float alpha = 0.5;  // dt/(dx*dx);
    int t;
    int i = get_global_id(0);
    for (t=0; t < TIME_STEPS; t++) {
        if ( (i>0) && (i<(N-1)) )
            out[i] = in[i] * (1.0-2.0*alpha) +alpha*(in[i+1]+in[i-1]);
        else
            out[i] = u_b;
        barrier(CLK_GLOBAL_MEM_FENCE);
        in[i] = out[i];
        barrier(CLK_GLOBAL_MEM_FENCE);
    }
}
```

Figure 2 – Source code of the heat equation solver OpenCL kernel
A kernel is a function to be executed on GPU. Copies of this kernel is simultaneously executed on GPU cores. Initial values $u_{i,0}$ are passed via array $\text{in}$. There are totally $N$ copies, each gets unique number $i$ from 0 to $N-1$, considered as spatial index. Therefore, there is no need for a loop over $i$. After that, there goes a loop over time steps. Most kernels calculate new values for next slice $u_{i,j+1} = \text{out}$, based on current slice $u_{i,j} = \text{in}$. Kernels with identifier $i$ equal to 0 and $N-1$ just set boundary points in accordance with boundary conditions (2). The line \text{barrier(CLK_GLOBAL_MEM_FENCE)}; forces every kernel to stop until the layer is completely evaluated. Finally, last layer is returned to main program (not given in this paper) via $\text{out}$ array.

The results of productivity test on AMD A6-4455M APU (in stock configuration) are given on figure 3. Calculation speed is measured in million lattice points per second. Speed against number of spatial lattice points $N$ (and number of parallel kernels) is plotted. While $N$ is small the speed grows linearly, then the growth saturates. One could expect saturation at the point where $N$ is less or equal to the number of the cores in the processing unit, but for this particular case the saturation happens when $N \approx 4096$, while A6-4455M has only 256 cores.

![Figure 3 – OpenCL solver for the heat equation performance against the number of concurrent kernels. GPGPU has 256 cores.](image)

We can explain this unexpected fact by internal GPU scheduling. The more workload GPU has, the more opportunities its scheduler possesses, thus it operates more efficiently. Code, which is given on figure 2 has considerable rate of \text{barrier} calls, that decrease efficiency of scheduler, because they must be scheduled statically. We believe, that 256 kernel copies (one per core) do not provide enough movable operations, so cores waste their time in idle mode, waiting for \text{barrier} synchronization. After $N$ reaches the saturation point, processing units become completely loaded and performance stops its increase.

To sum up, we can draw a preliminary conclusion, that GPGPU usage for PDE solution is efficient in the case of massively parallel code, i.e. when number of concurrent processes is rather large.

OpenCL implementation of the method of line for wave equation.
In this chapter we discuss the parallel implementation of the method of lines for hyperbolic wave equation with one spatial dimension:

\[ \partial_{tt} \phi - \partial_{xx} \phi = 0 \]  \hspace{1cm} (5)

Explicit finite-difference schemes have issues that prohibit them from general use. Depending on discretization scheme some other methods can be constructed, for example the leap-frog and the Crank-Nicholson methods [6], symplectic integrators [7]. We follow another way of solving PDE which is also easy to implement and parallelize, namely, the method of lines.

We discretize the space while leaving the time derivatives, and obtain the system of ordinary differential equations. This system is then can be integrated by any of known methods (Runge-Kutta, Adams, etc.) [8].

We have applied the most simple form of the method of lines. Nevertheless, it uses high order schemes and provides us with robust data. The used finite difference for the second order spatial derivative is following:

\[ \frac{d^2 \phi_n}{dt^2} \approx \frac{1}{12(\Delta X)^2} \left( -\phi_{n-2} + 16\phi_{n-1} - 30\phi_n + 16\phi_{n+1} - \phi_{n+2} \right) \]  \hspace{1cm} (6)

It defines corresponding system of $2N$ first order ordinary differential equations.

We have parallelized solution of 6 in the same way as it was done for the heat equation, namely, every kernel is solving one equation from the system. The source code of the kernel is rather large and is not included in this paper. A speed test was performed on nVidia GTS 450 (192 CUDA cores), the results are presented on figure 4. The method of lines includes more operations than explicit finite-difference one, thus speed growth saturation is achieved on less number of cores.

![Figure 4 – OpenCL solver for the wave equation performance against the number of concurrent kernels. GPGPU has 192 cores.](image)
Conclusions

The both method of lines and explicit finite-difference scheme utilize the true parallel manner of PDE integrating and has wide range of applications in hyperbolic, parabolic, and mixed-type equations. Our results prove that general-purpose GPU can sufficiently speed up the process of numerical integration and open parallel computing world to everyone without a considerable expense. Moreover, shared memory model provides great scalability and allows use of lattices of size limited only by available memory.

Acknowledgments

We are grateful to Moscow nVidia division for provided hardware and technical support.

List of references


Belov Maxim Alexandrovich, Assisting professor of the Department of Physical Informatics, Atomic and Molecular physics, Belarusian State University, mpui@tut.by.

Halavanau Aliaksai Aliaksandravich, Master student of the Department of Theoretical Physics and Astrophysics, Belarusian State University, aliaksei.halavanau@gmail.com.