GPGPU PROCESSING FOR PARTICLE MOVEMENT IN OPENCL ARCHITECTURE

Matthew Dee Co, Tomoyuki Maehara

University of the Philippines, Diliman

Abstract

General-purpose computing on GPUs (GPGPU) is currently an area of interest for its uses in scientific research and various engineering applications. With the increase in capacity of storage media, the amount of data that needs to be processed also increases. By harnessing the computational power of graphics processing units, GPUs, the processing time of large amounts of data can be potentially made faster.

In this paper, we demonstrate the capabilities of OpenCL in simulating the movement of a varying number of particles. OpenCL is a cross-vendor and cross-platform framework that allows for the utilization of the GPU for parallel execution. The actual rendering of particles is performed using OpenGL. Aside from the GPU approach, we developed the same simulation algorithm using single-threaded CPU, and CPU with threads. By comparing the frame rates of the three simulations, we are able to see a fundamental difference in their performances

Introduction

In recent years, the graphics processing unit (GPU), has been rapidly gaining recognition as a powerful tool for parallelizing computationally expensive calculations. While CPUs are only composed of a few cores, the GPU has hundreds of cores in it, and thus usually performs faster. GPU was originally developed as a computing tool for 3D graphical applications. This is because the computations involved in those types of applications are too heavy for a single-chip processor (like the CPU), and do not have to be performed sequentially.

We took a look at the GPU in order to utilize its power for simulating particle movement. Particles are used in a wide variety of simulation techniques, such as Smoothed Particle Hydrodynamics (SPH), which is a computational method for fluid flow simulation [3]. SPH is applied in real life applications, such as video games, which demand more and more complex and realistic game physics. When compared to grid-based simulation techniques, particle-based techniques are considered to be easier to program and understand, and as such, many computer graphics researchers are focusing on this particular technique. However, with the way particle-based techniques work, millions of particle positions need to be calculated in a second, which would cause the processor to do a lot of heavy lifting.

The motivation behind this project is to gain an idea of the possible improvement in machine performance when using code that runs on the GPU compared to using single, or multi-threaded code running on the CPU. The code to be implemented is a particle simulation, where particles move according to a differential equation.

Implementation

In implementing the project, OpenGL was used for rendering the particles onto the 3D coordinate system, and OpenCL was used for computing the positions of thousands of particles in a parallel fashion for each frame.

The software is designed such that one can freely toggle among three types of operation: GPU, CPU, and CPU-threaded, simply by pressing the left and right arrow keys; likewise, one can double or half the number of particles by pressing the up and down arrow keys respectively. The initial number of rendered particles is 1024. The number of particles, or the global work size, must be a multiple of the local work size, which is 512, so that it can be queued for execution on the kernel using the function *clEnqueueNDRangeKernel*.

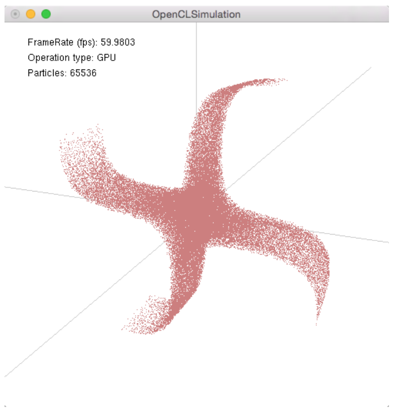
The local work size is, in a nutshell, a *compute unit* to be executed simultaneously in the kernel. A local work size that is divisible by 32 is said to be best for NVIDIA GPUs, and 64 for AMD GPUs.

The compute unit size, called *warp* in NVIDIA, and *wavefronts* in AMD, differs depending on the compute device. In order to eliminate such vendor-specific definitions, OpenCL accommodates various compute devices. In an OpenCL, work items are grouped by the local work size and clustered into warps or wavefronts to be executed at the same time [4]. We chose the local work size of 512, which is a multiple of both 32 and 64, not just to be able to adjust to any device, but also to avoid causing some threads to be idle, which is bad device utilization.

The simulation consists of N particles, each with its own position, moving at some velocity inside a 3D space. In each frame, the position of each particle is updated using a velocity-valued function. The change in position of a particle depends on the elapsed time, and its current position x and y. All particles are moving along the path depending on the function described as:

$$u = -2\cos\frac{t}{8}\pi \sin x\pi \sin x\pi \cos y\pi \sin y\pi \tag{1}$$
$$v = 2\cos\frac{t}{8}\pi \cos x\pi \sin x\pi \sin y\pi \sin y\pi \tag{2}$$

These functions from [2] are convergent, meaning the particles remain in a certain area, and eventually return to their initial positions. This is so that we can easily observe the movement, as it is completely contained inside the window. In the vector field, the velocity of each particle over time is expressed as: $\frac{dx}{dt} = u$, $\frac{dx}{dt} = v$.


OpenCLSimulation

FrameRate (fps): 59.9803
Operation type: GPU
Particles: 65536

Figure 1: GPU Simulation

The task of the kernel function is to solve this differential equation in each time frame for every single particle, in order to find the change in position over time to be added to the previous positions. For the method to be used to compute the differential equation, we used the fourth order Runge-Kutta numerical integration [1], which is said to have an acceptable accuracy. To illustrate, we will demonstrate how the equation with respect to the X axis can be derived.

Initially, we have

$$\dot{x} = u(x, y, t) \tag{3}$$
$$x(t_0) = x_0 \tag{4}$$

The next step value in an infinitely small interval time is determined by the following equation:

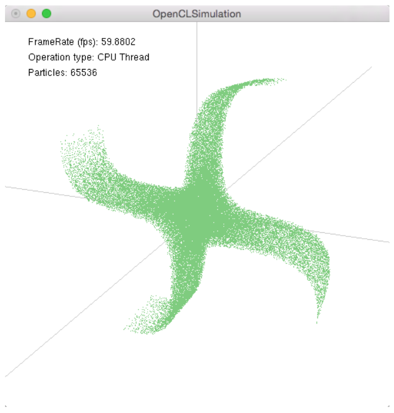
$$x_{n+1} = x_n + \frac{dt}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{5}$$
$$t_{n+1} = t_n + dt \tag{6}$$

,

where k_1, k_2, k_3, k_4 are:

$$k_1 = u(x_n, y_n, t_n) \tag{7}$$
$$k_2 = u(x_n + \frac{dt}{2}k_1, y_n + \frac{dt}{2}k_1, t_n + \frac{dt}{2}) \tag{8}$$
$$k_3 = u(x_n + \frac{dt}{2}k_2, y_n + \frac{dt}{2}k_2, t_n + \frac{dt}{2}) \tag{9}$$
$$k_4 = u(x_n + dt \cdot k_3, y_n + dt \cdot k_3, t_n + dt) \tag{10}$$

After solving the equations, we now have the N particles move according to the change in position over time since the last time frame, the calculation of which is executed by the kernel function independently in a thread.


OpenCLSimulation

FrameRate (fps): 59.8802
Operation type: CPU Thread
Particles: 65536

Figure 2: CPU Thread Simulation

In the GPU environment, shown in (Figure 1), the function *clEnqueueNDRangeKernel* is used to call, from the host code running on the CPU, the kernel function to be run on the GPU device. This function enqueues a command for execution on the associated device with certain argument values that we set beforehand using the function *clSetKernelArg*. The arguments passed on to the device, such as the total number of particles, their respective positions, the current time, and the elapsed time, are specific to the kernel function. The kernel function then calculates the new

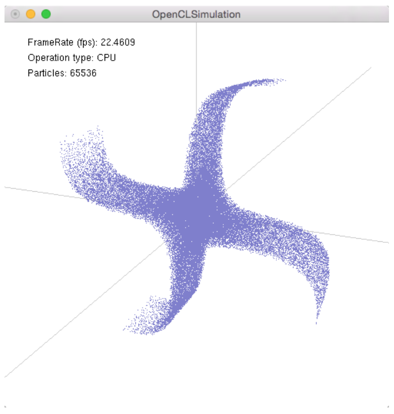


Figure 3: CPU Simulation

position for each particle, independent of all other particles. The results are ultimately stored in the device buffer, which we can extract from the host code using the function *clEnqueueReadBuffer*. The host code running on the CPU then retrieves the calculated positions and updates the actual particles accordingly.

Both the CPU-thread (shown in Figure 2), and the CPU (shown in Figure 3) modes of operation compute the particle positions completely using only the CPU. They only differ in how the kernel function is executed. While the CPU simulation goes through N iterations (one iteration for every particle), the CPU-thread approach first divides the particles into blocks of 512 particles each, and for each block, creates a thread to compute positions of particles in that block. Thus, there are a total of N/512 threads, where N is the total number of particles. Two factors went into this decision. First, creating too many CPU threads, i.e., having as many threads as there are particles, could have a high overhead and thus become a bottleneck in the performance. Second, by choosing a block size of 512, we can directly compare this solution to the GPU approach, which also has a local work size of 512 in our implementation.

Data and Analysis

The simulations were performed using the visualization software we developed. We measured the performance of the three modes using the frame rate, expressed in fps (frames per second), which describes the frequency at which a rendering device produces frames. For instance, a frame rate of 60 fps means that the kernel function is capable of finishing all the processes for each frame under 1/60 second, and putting too much burden in the function could possibly reduce the frame rate. The maximum frame rate is limited by the screen refresh rate of the PC monitor, which is usually 60 Hz in regular PC monitors. The machine we ran the simulations on also has a screen refresh rate of 60 Hz. Hence, we can assume that 60 fps is the highest frame rate we can achieve for any mode of operation. This can go lower if the kernel function recomputes particle positions too slowly, which is often the case when there are too many particles.

We plotted the average frame rate for each mode upon changing the number of particles. As the result in Figure 4 suggests, although the three modes of operation start out with the same frame rate of 60 fps, the CPU and CPU-thread modes quickly drop in frame rate even just for a relatively small number of particles (below 17,000). Meanwhile, the GPU mode can render at the maximum frame rate for almost 8x the number of particles (around 131,000) before its performance starts to go down. This result implies that, for the GPU, the number of particles does not impact the performance as much, when compared to the other two modes of operation.

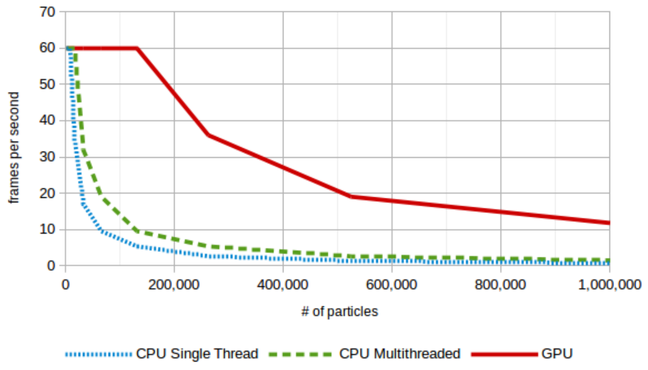


Figure 4: FPS on rendering up to 1,000,000 particles
GPU outperforms the CPU modes by a large margin. Measurements were taken with an Intel 3rd Gen Core graphics card, a quadruple-core Intel i3 processor, and 4GB memory.

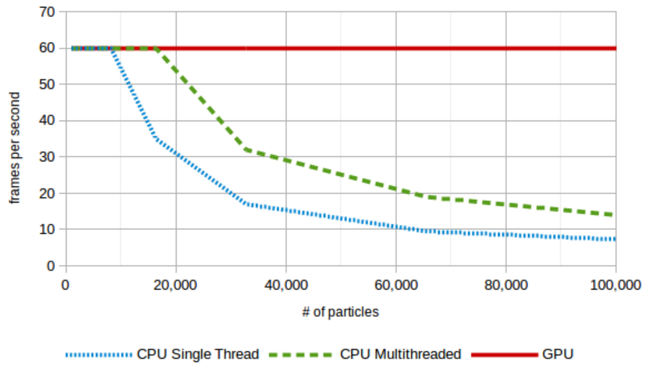


Figure 5: FPS on rendering up to 100,000 particles
A blown-up version of the first graph. The CPU modes quickly drop to 10-15 fps for as low as 100,000 particles. Meanwhile, the GPU can still render at the maximum frame rate of 60 fps.

Conclusion and Future Work

Given the results of the simulation, we saw that the GPU approach outperformed the other two approaches, CPU and CPU-thread. Running code on the GPU, as opposed to the CPU, greatly speeds up calculation of expensive, but highly parallelizable computations. However, this project still has a lot of room for improvement. We can achieve a higher level of performance for the GPU by applying some refinements, such as optimization of the work group sizes in OpenCL. We can try out the simulation using several different values for it while considering vendor-specific hardware features, such as warps or wavefronts, in order to write an efficient OpenCL kernel for each GPU.

Appendix: GPU and CPU Code Differences

Most programmers are familiar with writing code that will run on the CPU. This knowledge can be easily transferred to writing code for the GPU with the help of frameworks like OpenCL, which simplify the process. However, some parts of the code would need to be written differently.

Set-up code for OpenCL

First, you would need to get the platform information to see if the client has installed at least one implementation of OpenCL. If there is none, then you cannot run code with OpenCL.

```
cl_platform_id platform;
cl_uint platformCount;
outcome = clGetPlatformIDs(
            1, &platform, &platformCount);
if (platformCount == 0) {
        cerr << "No platform.\n";
        return EXIT_FAILURE;
}
```

You also have to check if the platform supports GPU devices, since some platforms only accommodate CPU. If no platform has a GPU device, then you cannot run code on the GPU using OpenCL.

```
cl_int outcome = 0;
cl_device_id device;
cl_uint deviceCount;
outcome = clGetDeviceIDs(
            platform, CL_DEVICE_TYPE_GPU,
            1, &device, &deviceCount);
if (deviceCount == 0) {
    cerr << "No device.\n";
    return EXIT_FAILURE;
}
```

Once you have verified that the client can run code on the GPU, you would need to create an OpenCL context. Contexts are used for managing OpenCL objects (such as the command queue), and for executing kernels. After that, you can now create the command queue within the context.

```
context = clCreateContext(
            0, 1, &device, NULL,
            NULL, &outcome);
queue = clCreateCommandQueue(
            context, device, 0, &outcome);
```

Next, you would need to build the kernel program you have written, and then create the kernel. The program source is loaded from a string, but in C++, there is no multi-line string, so to make the code less cluttered, we put the kernel code in a separate file called `kernel.cl`, and simply load its contents into a string on run-time. In creating the kernel, you would need to pass the name of the function you wish to run. This function should be in the program you built.

```
ifstream path("kernel.cl");
istreambuf_iterator<char> vdataBegin(path);
istreambuf_iterator<char> vdataEnd;
string source(vdataBegin,vdataEnd);
const char *c_source = source.c_str();
const size_t source_l = source.length(

program = clCreateProgramWithSource(
            context, 1, &c_source,
            &source_l, &outcome);
clBuildProgram(program, 1, &device,
               NULL, NULL, NULL);
kernel = clCreateKernel(
            program, "d_rungekutta",
            &outcome);
```

If your program needs to read or write from the GPU memory, you would need to create a buffer object. Here, `n` is the number of particles. A `cl_float3` object is a point in 3D space, so it has three attributes: `x`, `y`, and `z`. Each particle is a point in 3D space, so we need to allocate for `n` `cl_float3` objects.

While `d_mem` is stored in the GPU memory, `h_mem` is the same object stored in CPU memory. Since it is easier to assign values to objects using native C++, we decided to initialize `h_mem` with the starting particle positions, and then simply copy the same object to `d_mem` using the command `clEnqueueWriteBuffer`.

```
cl_mem d_mem = NULL;
cl_float3 *h_mem = NULL;

d_mem = clCreateBuffer(
          context, CL_MEM_READ_WRITE,
            sizeof(cl_float3)*n, NULL,
            &outcome);
h_mem = new cl_float3[n];

// initalize h_mem

clEnqueueWriteBuffer(queue, d_mem, CL_TRUE,
            0, sizeof(cl_float3)*n,
            h_mem, 0, NULL, NULL);
```

In the next section, we will show how the computation can be done on the GPU, and compare it to the code that runs on CPU.

runGPU()

Now that the set-up part is done, we can finally use the GPU to perform some computations using the function we wrote inside `kernel.cl`. Here is the signature of the function we wrote for recomputing particle positions:

```
__kernel void d_rungekutta(
      int num_particles,
      __global float3 *position,
      float a_time,
      float a_dt) {
    // omitted for brevity
}
```

As you can see, it takes four arguments: the particle count, the positions of all particles, the current time, and the change in time. In order to pass arguments to the kernel function, you would need to use the command `clSetKernelArg`. You need to specify the argument position, size and pointer. Finally, to actually execute the kernel, you would need to enqueue it on the command queue we created earlier. The global work size is the number of particles, while the local work size can be chosen by the programmer. However, each device has its own preference on what this should be a multiple of. We set it to 512.

```
size_t global_work_size[1] = {n};
size_t local_work_size[1] = {512};

clSetKernelArg(kernel, 0,
        sizeof(cl_int), (void*)&n);
clSetKernelArg(kernel, 1,
        sizeof(cl_mem), (void*)&d_mem);
clSetKernelArg(kernel, 2,
        sizeof(cl_float), (void*)&a_time);
clSetKernelArg(kernel, 3,
        sizeof(cl_float), (void*)&a_dt);

clEnqueueNDRangeKernel(
        queue, kernel, 1, NULL,
        global_work_size, local_work_size,
        0, NULL, NULL);
```

After the kernel is queued for running, we want to follow it with a read command, in order to read back the recomputed particle positions. In particular, we will read the contents of the GPU memory, `d_mem`, into the CPU memory, `h_mem`. This is done using the command `clEnqueueReadBuffer`. Finally, we update the time and then go to the next iteration.

```
outcome = clEnqueueReadBuffer(
        queue, d_mem, CL_TRUE, 0,
        sizeof(cl_float3)*n, h_mem,
        0, NULL, NULL);
a_time += a_dt;
```

runCPU()

Running on the CPU is much easier as we do not need to call any of the `clXXX` functions described above. We also do not need to copy back and forth between `d_mem` and `h_mem` as computations are performed directly on `h_mem`. However, we need to implement the kernel function in native C++. Here is the signature of the function we wrote for that:

```
void cpu_d_rungekutta(
        int num_particles,
        int index,
        cl_float3 *position,
        float a_time,
        float a_dt);
```

It takes an additional parameter, `index`, which tells us which particle we are currently recomputing the position of. In the GPU version, we did not need the index as particles were not being computed sequentially. The kernel function can be run on the particles concurrently and in any order.

After defining the function, we can now call this on each particle index sequentially, like this:

```
for (int i=0; i<n; i++) {
    cpu_d_rungekutta(
        n, i, h_mem, a_time, a_dt);
}
a_time+=a_dt;
```

As `n` grows exponentially, this eventually becomes too slow to be of use. Hence, we also tried making use of threads to speed up the computation.

runCPUThreads()

In order to help with passing variables to threads, we created a new structure called `CpuParam`, which contains all the information a thread needs to do its work, and its work only (i.e. not computing particle positions outside its scope).

```
struct CpuParam{
    int num_particles;
    int index;
    int block_s;
    cl_float3 *position;
    float a_time;
    float a_dt;
};
```

Next, we created the function that will be called in each thread. It determines the indices of the particle positions that will be recomputed in the thread, and calls the “kernel” function on the particles.

```
void* tfunction(void * x){
    CpuParam *p = (CpuParam *)x;
    int i = p->index * p->block_s;
    int l = i + p->block_s;
    while (i < l) {
        cpu_d_rungekutta(
            p->num_particles,
            i,
            p->position,
            p->a_time,
            p->a_dt);
        i++;
    }
    return 0;
}
```

The main job of `runCPUThreads()`, then, is to delegate its work to threads. Here, `block_s` refers to the block size, i.e. the number of particles that will be grouped together, and recomputed in the same thread. We chose a block size of 512, hence, there are a total of `n / 512` threads. Finally, when a thread is finished, we join it back to the parent process.

```
int block_s = 512;
int l = n / block_s;

vector<CpuParam> params(l);
vector<pthread_t> threads(l,0);

for (int i=0; i<l; i++) {
    params[i].num_particles = n;
    params[i].index = i;
    params[i].block_s = block_s;
    params[i].position = h_mem;
    params[i].a_time = a_time;
    params[i].a_dt = a_dt;
}
for (int i=0; i<l; i++) {
    int s = pthread_create(
        &(threads[i]), 0,
        &tfunction, &(params[i]) );
    if (s != 0) {
        threads[i] = 0;
    }
}
for (int i=0; i<l; i++){
    if (threads[i]==0)
        continue;
    pthread_join(threads[i], 0);
}
a_time+=a_dt;
```

Although it might be comparatively more difficult to get code to run on the GPU, we believe that the effort is worth it, because as we have seen earlier, there is a big payoff. Note that the GPU code we wrote is still not completely optimized. It could possibly run even faster than it does now with some improvements, like minimizing the communication between CPU and GPU, or, like stated before, tweaking some OpenCL parameters.

References

Runge–kutta methods. `http://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods`.

Hajimete no CUDA Programming. Kougakusha, 2009.

Adrian Sandu Colin Braley. Fluid simulation for computer graphics: A tutorial in grid based and particle based methods.

Yusuke Arai Kentaro Koyama Hiroyuki Takizawa Hiroaki Kobayash Kazuhiko Komatsu, Katsuto Sato. Evaluating performance and portability of opencl programs, 2010.