22

GPGPU Cloth Simulation Using GLSL, OpenCL, and CUDA

Marco Fratarcangeli

Taitus Software Italia

22.1 Introduction

This chapter provides a comparison study between three popular platforms for generic programming on the GPU, namely, GLSL, CUDA, and OpenCL. These technologies are used for implementing an interactive physically-based method that simulates a piece of cloth colliding with simple primitives like spheres, cylinders, and planes (see Figure 22.1). We assess the advantages and the drawbacks of each different technology in terms of usability and performance.

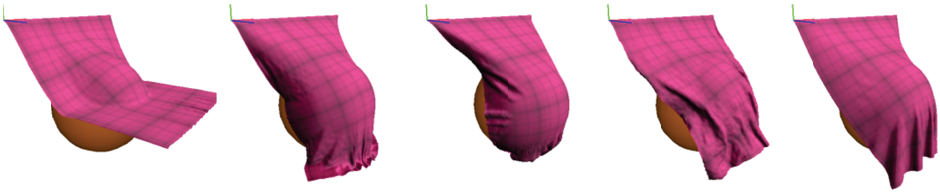

Figure 22.1. A piece of cloth falls under the influence of gravity while colliding with a sphere at interactive rates. The cloth is composed of 780,000 springs connecting 65,000 particles.

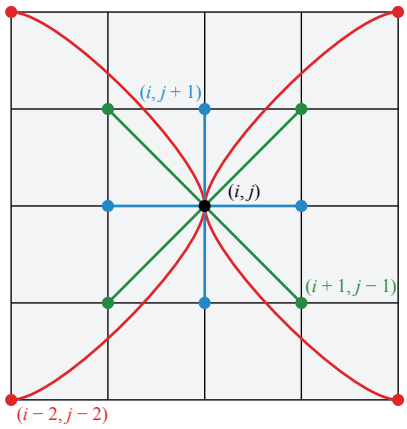

Figure 22.2. A 4×4 grid of particle vertices and the springs for one of the particles.

22.2 Numerical Algorithm

This section provides a brief overview of the theory behind the algorithm used in computing the cloth simulation. A straightforward way to implement elastic networks of particles is by using a mass-spring system. Given a set of evenly spaced particles on a grid, each particle is connected to its neighbors through simulated springs, as depicted in Figure 22.2. Each spring applies to the connected particles a force $\mathbf{F}_{\text{spring}}$:

$$\mathbf{F}_{\text{spring}} = -k\,(\mathbf{l} - \mathbf{l}_0) - b\dot{\mathbf{x}},$$

where $\mathbf{l}$ represents the current length of the spring (i.e., its magnitude is the distance between the connected particles), $\mathbf{l}_0$ represents the rest length of the spring at the beginning of the simulation, k is the stiffness constant, $\dot{\mathbf{x}}$ is the velocity of the particle, and b is the damping constant. This equation means that a spring always applies a force that brings the distance between the connected particles back to its initial rest length. The more the current distance diverges from the rest length, then the larger is the applied force. This force is damped proportionally to the current velocity of the particles by the last term in the equation. The blue springs in Figure 22.2 simulate the stretch stress of the cloth, while the longer red ones simulate the shear and bend stresses.

For each particle, the numerical algorithm that computes its dynamics is schematically illustrated in Figure 22.3. For each step of the dynamic simulation,

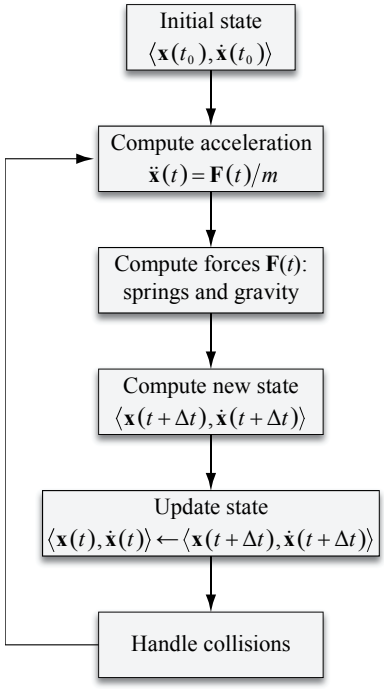

Figure 22.3. Numerical algorithm for computing the cloth simulation.

the spring forces and other external forces (e.g., gravity) are applied to the particles, and then their dynamics are computed according to the Verlet method [Müller 2008] applied to each particle in the system through the following steps:

1. $\dot{\mathbf{x}}(t) = [\mathbf{x}(t) - \mathbf{x}(t - \Delta t)]/\Delta t$.
2. $\ddot{\mathbf{x}}(t) = \mathbf{F}(\mathbf{x}(t), \dot{\mathbf{x}}(t))/m$.
3. $\mathbf{x}(t + \Delta t) = 2\mathbf{x}(t) - \mathbf{x}(t - \Delta t) + \ddot{\mathbf{x}}(t)\Delta t^2$.

Here, $\mathbf{F}(t)$ is the current total force applied to the particle, m is the particle mass, $\ddot{\mathbf{x}}(t)$ is its acceleration, $\dot{\mathbf{x}}(t)$ is the velocity, $\mathbf{x}(t)$ is the current position, and Δt is the time step of the simulation (i.e., how much time the simulation is advanced for each iteration of the algorithm).

The Verlet method is very popular in real-time applications because it is simple and fourth-order accurate, meaning that the error for the position computation is $O(\Delta t^4)$. This makes the Verlet method two orders of magnitude more precise than the explicit Euler method, and at the same time, it avoids the compu-

tational cost involved in the Runge-Kutta fourth-order method. In the Verlet scheme, however, velocity is only first-order accurate; in this case, this is not really important because velocity is considered only for damping the springs.

22.3 Collision Handling

Generally, collision handling is composed of two phases, collision detection and collision response. The outcome of collision detection is the set of particles that are currently colliding with some other primitive. Collision response defines how these collisions are solved to bring the colliding particles to a legal state (i.e., not inside a collision primitive). One of the key advantages of the Verlet integration scheme is the easiness of handling collision response. The position at the next time step depends only on the current position and the position at the previous step. The velocity is then estimated by subtracting one from the other. Thus, to solve a collision, it is sufficient to modify the current position of the colliding particle to bring it into a legal state, for example, by moving it perpendicularly out toward the collision surface. The change to the velocity is then handled automatically by considering this new position. This approach is fast and stable, even though it remains valid only when the particles do not penetrate too far.

In our cloth simulation, as the state of the particle is being updated, if the collision test is positive, the particle is displaced into a valid state. For example, let's consider a stationary sphere placed into the scene. In this simple case, a collision between the sphere and a particle happens when the following condition is satisfied:

$$\|\mathbf{x}(t+\Delta t)-\mathbf{c}\|-r<0,$$

where $\mathbf{c}$ and r are the center and the radius of the sphere, respectively. If a collision occurs, then it is handled by moving the particle into a valid state by moving its position just above the surface of the sphere. In particular, the particle should be displaced along the normal of the surface at the impact point. The position of the particle is updated according to the formula

$$\mathbf{d}=\frac{\mathbf{x}(t+\Delta t)-\mathbf{c}}{\|\mathbf{x}(t+\Delta t)-\mathbf{c}\|},$$

$$\mathbf{x}'(t+\Delta t)=\mathbf{c}+\mathbf{d}r,$$

where $\mathbf{x}'(t+\Delta t)$ is the updated position after the collision. If the particle does not penetrate too far, $\mathbf{d}$ can be considered as an acceptable approximation of the normal to the surface at the impact point.

22.4 CPU Implementation

We first describe the implementation of the algorithm for the CPU as a reference for the implementations on the GPU described in the following sections.

During the design of an algorithm for the GPU, it is critical to minimize the amount of data that travels on the main memory bus. The time spent on the bus is actually one of the primary bottlenecks that strongly penalize performance [Nvidia 2010]. The transfer bandwidth of a standard PCI-express bus is 2 to 8 GB per second. The internal bus bandwidth of a modern GPU is approximately 100 to 150 GB per second. It is very important, therefore, to minimize the amount of data that travels on the bus and keep the data on the GPU as much as possible.

In the case of cloth simulation, only the current and the previous positions of the particles are needed on the GPU. The algorithm computes directly on GPU the rest distance of the springs and which particles are connected by the springs. The state of each particle is represented by the following attributes:

1. The current position (four floating-point values).
2. The previous position (four floating-point values).
3. The current normal vector (four floating-point values).

Even though the normal vector is computed during the simulation, it is used only for rendering purposes and does not affect the simulation dynamics. Here, the normal vector of a particle is defined to be the average of the normal vectors of the triangulated faces to which the particle belongs. A different array is created for storing the current positions, previous positions, and normal vectors. As explained in later sections of this chapter, for the GPU implementation, these attributes are loaded as textures or buffers into video memory. Each array stores the attributes for all the particles. The size of each array is equal to the size of an attribute (four floating-point values) multiplied by the number of particles. For example, the position of the i-th particle $\mathbf{p}_i$ is stored in the positions array and accessed as follows:

```
pos_i ← vec3(in_pos[i * 4], in_pos[i * 4 + 1], in_pos[i * 4 + 2],
             in_pos[i * 4 + 3])
```

The cloth is built as a grid of $n \times n$ particles, where n is the number of particles composing one side of the grid. Regardless of the value of n, the horizontal and the vertical spatial dimensions of the grid are always normalized to the range

[0,1]. A particle is identified by its array index i, which is related to the row and the column in the grid as follows:

$$row_i = \lfloor i/n \rfloor,$$
$$col_i = i \bmod n.$$

From the row and the column of a particle, it is easy to access its neighbors by simply adding an offset to the row and the column, as shown in the examples in Figure 22.2.

The pseudocode for calculating the dynamics of the particles in an $n \times n$ grid is shown in Listing 22.1. In steps 1 and 2, the current and previous positions of the the i-th particle are loaded in the local variables $\mathbf{pos}_i^t$ and $\mathbf{pos}_i^{t-1}$, respectively, and then the current velocity $\mathbf{vel}_i^t$ is estimated in step 3. In step 4, the total force $\mathbf{force}_i$ is initialized with the gravity value. Then, the `for` loop in step 5 iterates over all the neighbors of $\mathbf{p}_i$ (steps 5.1 and 5.2), spring forces are computed (steps 5.3 to 5.5), and they are accumulated into the total force (step 5.6). Each neighbor is identified and accessed using a 2D offset $(x_{\text{offset}}, y_{\text{offset}})$ from the position of $\mathbf{p}_i$ within the grid, as shown in Figure 22.2. Finally, the dynamics are computed in step 6, and the results are written into the output buffers in steps 7 and 8.

```
for each particle p_i
1. pos_i^t ← x_i(t)
2. pos_i^{t-1} ← x_i(t − Δt)
3. vel_i^t = (pos_i^t − pos_i^{t-1}) / Δt
4. force_i = (0, -9.81, 0, 0)
5. for each neighbor (row_i + x_offset, col_i + y_offset)
       if (row_i + x_offset, col_i + y_offset) is inside the grid
           5.1. i_neigh = (row_i + y_offset) * n + col_i + x_offset
           5.2. pos_neigh^t ← x_neigh(t)
           5.3. d_rest = ||(x_offset, y_offset)|| / n
           5.4. d_curr = ||pos_neigh^t − pos_i^t||
           5.5. force_spring = −((pos_neigh^t − pos_i^t) / ||pos_neigh^t − pos_i^t||) (d_curr − d_rest) · k − vel_i^t · b
           5.6. force_i += force_spring
6. pos_i^{t+1} = 2 · pos_i^t − pos_i^{t-1} + (force_i / m) · Δt^2
7. x_i(t) ← pos_i^{t+1}
8. x_i(t − Δt) ← pos_i^t
```

Listing 22.1. Pseudocode to compute the dynamics of a single particle i belonging to the $n \times n$ grid.

22.5 GPU Implementations

The different implementations for each GPGPU computing platform (GLSL, OpenCL, and CUDA) are based on the same principles. We employ the so-called "ping-pong" technique that is particularly useful when the input of a simulation step is the outcome of the previous one, which is the case in most physically-based animations. The basic idea is rather simple. In the initialization phase, two buffers are loaded on the GPU, one buffer to store the input of the computation and the other to store the output. When the computation ends and the output buffer is filled with the results, the pointers to the two buffers are swapped such that in the following step, the previous output is considered as the current input. The results data is also stored in a vertex buffer object (VBO), which is then used to draw the current state of the cloth. In this way, the data never leaves the GPU, achieving maximal performance. This mechanism is illustrated in Figure 22.4.

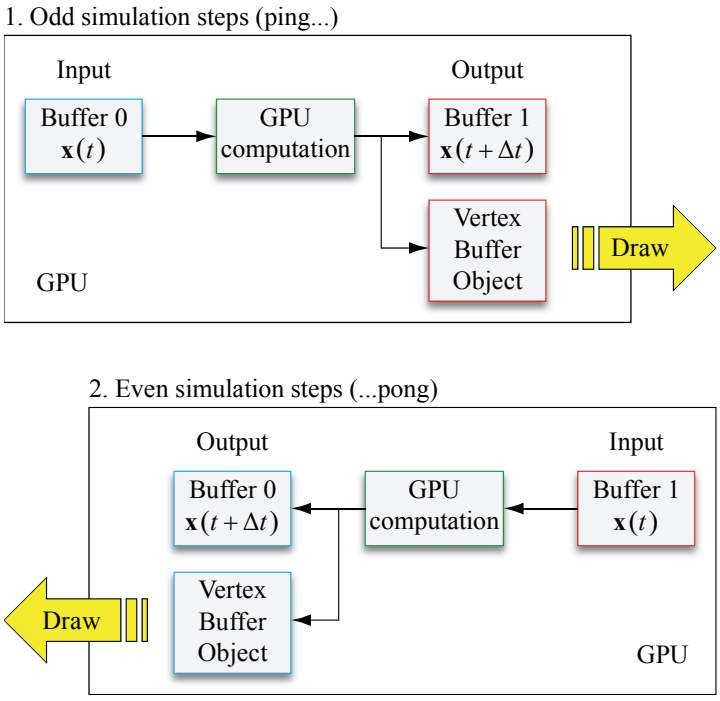

Figure 22.4. The ping-pong technique on the GPU. The output of a simulation step becomes the input of the following step. The current output buffer is mapped to a VBO for fast visualization.

22.6 GLSL Implementation

This section describes the implementation of the algorithm in GLSL 1.2. The source code for the vertex and fragment shaders is provided in the files `verlet_cloth.vs` and `verlet_cloth.fs`, respectively, on the website. The position and velocity arrays are each stored in a different texture having $n \times n$ dimensions. In such textures, each particle corresponds to a single texel. The textures are uploaded to the GPU, and then the computation is carried out in the fragment shader, where each particle is handled in a separate thread. The updated state (i.e., positions, previous positions, and normal vectors) is written to three distinct render targets.

Frame buffer objects (FBOs) are employed for efficiently storing and accessing the input textures and the output render targets. The ping-pong technique is applied through the use of two frame buffer objects, FBO1 and FBO2. Each FBO contains three textures storing the state of the particles. These three textures are attached to their corresponding FBOs as color buffers using the following code, where `fb` is the index of the FBO and `texid[0]`, `texid[1]`, and `texid[2]` are the indices of the textures storing the current positions, the previous positions, and the normal vectors of the particles, respectively:

```
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fbo->fb);
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
    GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_2D, texid[0], 0);
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
    GL_COLOR_ATTACHMENT1_EXT, GL_TEXTURE_2D, texid[1], 0);
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
    GL_COLOR_ATTACHMENT2_EXT, GL_TEXTURE_2D, texid[2], 0);
```

In the initialization phase, both of the FBOs holding the initial state of the particles are uploaded to video memory. When the algorithm is run, one of the FBOs is used as input and the other one as output. The fragment shader reads the data from the input FBO and writes the results in the render targets of the output FBO (stored in the color buffers). We declare the output render targets by using the following code, where `fb_out` is the FBO that stores the output:

```
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb_out);
GLenum mrt[] = {GL_COLOR_ATTACHMENT0_EXT,
    GL_COLOR_ATTACHMENT1_EXT, GL_COLOR_ATTACHMENT2_EXT};
glDrawBuffers(3, mrt);
```

In the next simulation step, the pointers to the input and output FBOs are swapped so that the algorithm uses the output of the previous iteration as the current input.

The two FBOs are stored in the video memory, so there is no need to upload data from the CPU to the GPU during the simulation. This drastically reduces the amount of data bandwidth required on the PCI-express bus, improving the performance. At the end of each simulation step, however, position and normal data is read out to a pixel buffer object that is then used as a VBO for drawing purposes. The position data is stored into the VBO directly on the GPU using the following code:

```
glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
glBindBuffer(GL_PIXEL_PACK_BUFFER, vbo[POSITION_OBJECT]);
glReadPixels(0, 0, texture_size, texture_size,
    GL_RGBA, GL_FLOAT, 0);
```

First, the color buffer of the FBO where the output positions are stored is selected. Then, the positions' VBO is selected, specifying that it will be used as a pixel buffer object. Finally, the VBO is filled with the updated data directly on the GPU. Similar steps are taken to read the normals' data buffer.

22.7 CUDA Implementation

The CUDA implementation works similarly to the GLSL implementation, and the source code is provided in the files `verlet_cloth.cu` and `verlet_cloth_kernel.cu` on the website. Instead of using FBOs, this time we use memory buffers. Two pairs of buffers in video memory are uploaded into video memory, one pair for current positions and one pair for previous positions. Each pair comprises an input buffer and an output buffer. The kernel reads the input buffers, performs the computation, and writes the results in the proper output buffers. The same data is also stored in a pair of VBOs (one for the positions and one for the normals), which are then visualized. In the beginning of the next iteration, the output buffers are copied in the input buffers through the `cudaMemcpyDeviceToDevice` call. For example, in the case of positions, we use the following code:

```
cudaMemcpy(pPosOut, pPosIn, mem_size, cudaMemcpyDeviceToDevice);
```

It is important to note that this instruction does not cause a buffer upload from the CPU to the GPU because the buffer is already stored in video memory. The

output data is shared with the VBOs by using `graphicsCudaResource` objects, as follows:

```
// Initialization, done only once.
cudaGraphicsGLRegisterBuffer(&cuda_vbo_resource, gl_vbo,
    cudaGraphicsMapFlagsWriteDiscard);

// During the algorithm execution.
cudaGraphicsMapResources(1, cuda_vbo_resource, 0);
cudaGraphicsResourceGetMappedPointer((void **) &pos,
    &num_bytes, cuda_vbo_resource);
executeCudaKernel(pos, ...);
cudaGraphicsUnmapResources(1, cuda_vbo_resource, 0);
```

In the initialization phase, we declare that we are sharing data in video memory with OpenGL VBOs through CUDA graphical resources. Then, during the execution of the algorithm kernel, we map the graphical resources to buffer pointers. The kernel computes the results and writes them in the buffer. At this point, the graphical resources are unmapped, allowing the VBOs to be used for drawing.

22.8 OpenCL Implementation

The OpenCL implementation is very similar to the GLSL and CUDA implementations, except that the data is uploaded at the beginning of each iteration of the algorithm. At the time of this writing, OpenCL has a rather young implementation that sometimes leads to poor debugging capabilities and sporadic instabilities. For example, suppose a kernel in OpenCL is declared as follows:

```
__kernel void hello(__global int *g_idata);
```

Now suppose we pass input data of some different type (e.g., a `float`) in the following way:

```
float input = 3.0F;
cfloatlSetKernelArg(ckKernel, 0, sizeof(float), (void *) &input);
clEnqueueNDRangeKernel(cqQueue, ckKernel, 1, NULL,
    &_szGlobalWorkSize, &_szLocalWorkSize, 0, 0, 0);
```

When executed, the program will fail silently without giving any error message because it expects an `int` instead of a `float`. This made the OpenCL implementation rather complicated to develop.

22.9 Results

The described method has been implemented and tested on two different machines:

- A desktop PC with an Nvidia GeForce GTS250, 1GB VRAM and a processor Intel Core i5.
- A laptop PC with an Nvidia Quadro FX 360M, 128MB VRAM and a processor Intel Core2 Duo.

We collected performance times for each GPU computing platform, varying the numbers of particles and springs, from a grid resolution of 32×32 (1024 particles and 11,412 springs) to 256×256 (65,536 particles and approximately 700,000 springs). Numerical results are collected in the plots in Figures 22.5 and 22.6.

From the data plotted in Figures 22.5 and 22.6, the computing superiority of the GPU compared with the CPU is evident. This is mainly due to the fact that this cloth simulation algorithm is strongly parallelizable, like most of the particle-based approaches. While the computational cost on the CPU keeps growing linearly with the number of particles, the computation time on the GPU remains relatively low because the particle dynamics are computed in parallel. On the GTS250 device, this leads to a performance gain ranging from 10 to 40 times, depending on the number of particles.

It is interesting to note that in this case, GLSL has a much better performance than CUDA does. This can be explained by considering how the memory is accessed by the GPU kernels. In the GLSL fragment program, images are employed to store particle data in texture memory, while in CUDA and OpenCL, these data is stored in the global memory of the device. Texture memory has two main advantages [Nvidia 2010]. First, it is cached, and thus, video memory is accessed only if there is a cache miss. Second, it is built in such a way as to optimize the access to 2D local data, which is the case because each particle corresponds to a pixel, and it must have access to the positions of its neighbors, which are stored in the immediately adjacent texture pixels. Furthermore, the results in GLSL are stored in the color render targets that are then directly mapped to VBOs and drawn on the screen. The data resides in video memory and does not need to be copied between different memory areas. This makes the entire process extremely fast compared with the other approaches.

The plots also highlight the lower performance of OpenCL compared with CUDA. This difference is caused by the fact that it has been rather difficult to tune the number of global and local work items due to causes requiring further

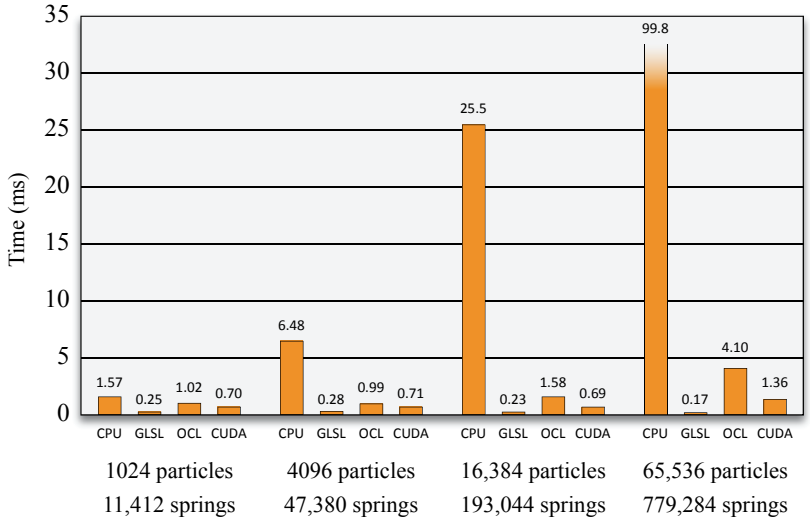

Figure 22.5. Computation times measured on different computation platforms using a GeForce GTS 250 device (16 computing units, 128 CUDA cores).

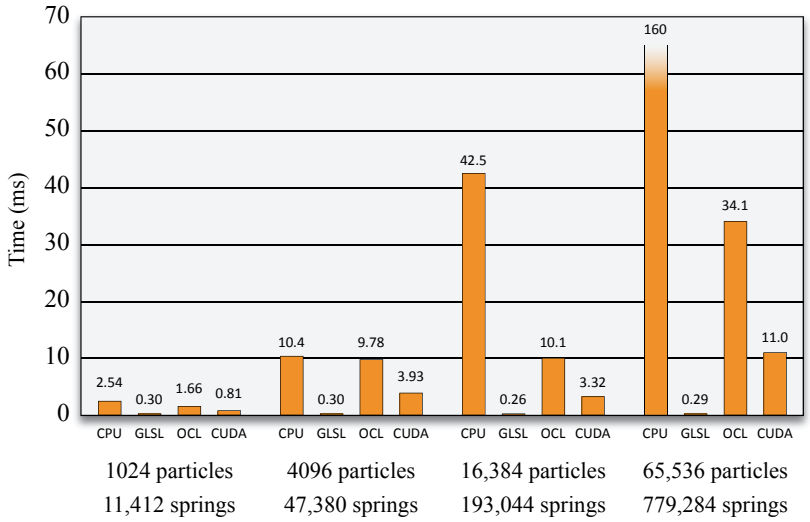

Figure 22.6. Computation times measured on different computation platforms using a Quadro FX 360M device (2 computing units, 16 CUDA cores).

investigation. OpenCL is a very young standard, and both the specification and the driver implementation are likely to change in the near future in order to avoid such instabilities.

The GLSL program works on relatively old hardware, and different from CUDA, it does not require Nvidia hardware. CUDA on the other hand, is a more flexible architecture that has been specifically devised for performing computing tasks (not only graphics, like GLSL), which is easier to debug and provides access to hardware resources, like the shared memory, allowing for a further boost to the performance. OpenCL has the same features as CUDA, but its implementation is rather naive at the moment, and it is harder to debug. However, different from CUDA, it has been devised to run on the widest range of hardware platforms (including consoles and mobile phones), not limited to Nvidia ones, and thus, it is the main candidate for becoming the reference platform for GPGPU in the near future.

The main effort when dealing with GPGPU is in the design of the algorithm. The challenging task that researchers and developers are currently facing is how to redesign algorithms that have been originally conceived to run in a serial manner for the CPU, to make them parallel and thus suitable for the GPU. The main disadvantage of particle-based methods is that they require a very large number of particles to obtain realistic results. However, it is relatively easy to parallelize algorithms handling particle systems, and the massive parallel computation capabilities of modern GPUs now makes it possible to simulate large systems at interactive rates.

22.10 Future Work

Our algorithm for cloth simulation can be improved in many ways. In the CUDA and OpenCL implementations, it would be interesting to exploit the use of shared memory, which should reduce the amount of global accesses and lead to improved performance.

For future research, we would like to investigate ways to generalize this algorithm by introducing connectivity information [Tejada 2005] that stores the indexes of the neighbors of each particle. This data can be stored in constant memory to hide as much as possible the inevitable latency that using this information would introduce. By using connectivity, it would be possible to simulate deformable, breakable objects with arbitrary shapes, not only rectangular pieces of cloth.

22.11 Demo

An implementation of the GPU cloth simulation is provided on the website, and it includes both the source code in C++ and the Windows binaries. The demo allows you to switch among the computing platforms at run time, and it includes a hierarchical profiler. Even though the source code has been developed for Windows using Visual Studio 2008, it has been written with cross-platform compatibility in mind, without using any Windows-specific commands, so it should compile and run on *nix platforms (Mac and Linux). The demo requires a machine capable of running Nvidia CUDA, and the CUDA Computing SDK 3.0 needs to have been compiled. A video is also included on the website.

Acknowledgements

The shader used for rendering the cloth is "fabric plaid" from RenderMonkey 1.82 by AMD and 3DLabs. The author is grateful to Professor Ingemar Ragnemalm for having introduced him to the fascinating world of GPGPU.

References

[Müller 2008] Matthias Müller, Jos Stam, Doug James, and Nils Thürey. "Real Time Physics." ACM SIGGRAPH 2008 Course Notes. Available at http://www.matthiasmueller.info/realtimephysics/index.html.

[Nvidia 2010] "NVIDIA CUDA Best Practices Guide," Version 3.0, 2010. Available at http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide.pdf.

[Tejada 2005] Eduardo Tejada and Thomas Ertl. "Large Steps in GPU-Based Deformable Bodies Simulation." *Simulation Modelling Practice and Theory* 13:8 (November 2005), pp. 703–715.