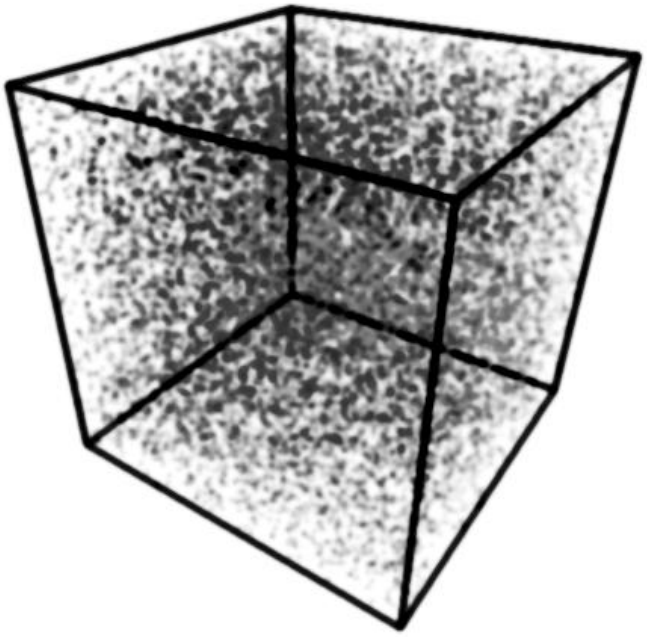
Blekinge Institute of Technology

GPU based particle system

Particle simulation using OpenCL

Author: Martin Wexö Olsson
Supervisor: Stefan Johansson
6/10/2010



GPGPU (General purpose computing on graphics processing unit) is quite common in today's modern computer games when doing heavy simulation calculations like game physics or particle systems. GPU programming is not only used in games but also in scientific research when doing heavy calculations on molecular structures and protein folding etc. The reason why you use the GPU for these kinds of tasks is that you can gain an incredible speedup in performance to your application. Previous research shows that particle systems scale very well to the GPU architecture. When simulating very large particle-system on the GPU it can run up to 79 times faster than the CPU. But for some very small particle systems the CPU proved to be faster. This research aims to compare the difference between the GPU and CPU when it comes to simulating many smaller particle-systems and to see what happen to the performance when the particle-systems become smaller and smaller.

This thesis is submitted to the School of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science. The thesis is equivalent to 10 weeks of full time studies.

Contact Information:

Author:Martin Wexö Olsson

Address: Folkparksvägen 14 Ronneby

E-mail: mawm06@student.bth.se

School of Computing
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Internet : www.bth.se/com
Phone : +46 457 38 50 00
Fax : + 46 457 271 25

Contents

1 Introduction 1
1.1 Target Group 1
2 Background 2
2.1 GPGPU programming 2
2.2 GPU Computing 2
3 Related work 3
4 Research Objectives 3
4.1 Hypothesis 3
4.2 Methodology 4
5 Implementation 4
5.1 Setting up the OpenCL environment 4
6 Test environment 6
7 Results 6
7.1 Test Results on Computer 1 7
7.2 Test Results on Computer 2 7
7.3 Discussion 8
7.4 Conclusion 8
8 Future Work 9
References 10
Published research papers 10
Websites 10
Images 10
APENDIX A – OpenCL C code for the particle-system 11
APENDIX B – OpenCL setup code. 12
APENDIX C – Particle system setup and execution code 14
APENDIX D - Test Result Data 16
Computer 1 16
Computer 2 16

1 Introduction

In modern graphic-cards there are many smaller compute units that are referred to as streaming processing units (SPU's) that can execute code in parallel independently of each other [6]. In ATI Radeon 5000 series there are up to around 700 and 800 SPU's.

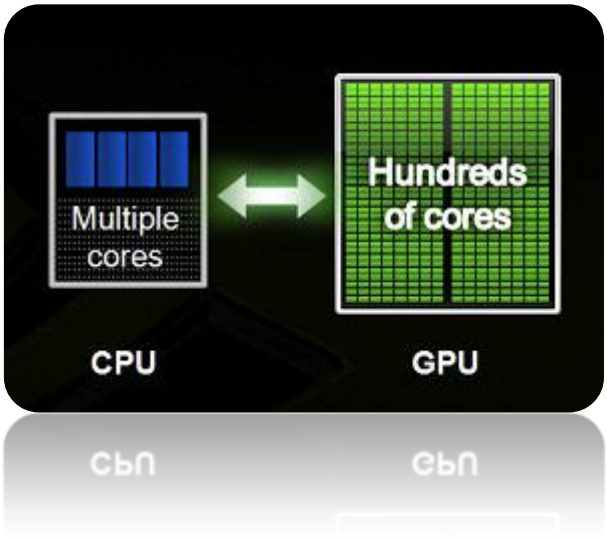


Figure 1: CPU ans GPU [7]

Taking advantage of the GPU's inherently parallel architecture to gain performance in very data-intensive tasks is quite common in applications such as games, 3D modeling software and even scientific applications [5]. These tasks that run the GPU have to be highly parallel in their structure to fit today's graphic-cards stream based architecture.

An example of such a task is particle-systems. With this stream based technology and today's graphic-cards you can simulate millions of particles in real-time. Such large particle systems are often used to create a more realistic visual effect of fuzzy objects like fire, smoke, hair, cloth and fluids.

This report focuses on these particles-systems and uses OpenCL (explained in the background section) to make it possible to run the same implementation of the particle system on both the CPU as-well as the GPU.

1.1 Target Group

The readers of this report are assumed to have some basic knowledge about computer graphics and performance optimization. This report aims to grab the attention of people with good programming background that wants to benefit from the performance gained by using the Graphic-card as a complement to the regular processors found on today's computers.

2 Background

2.1 GPGPU programming

GPGPU (General-purpose programming on graphics processors) is not unusual today and is in fact very common in modern games for physics simulations and other pre or post rasterization calculations like image composite effects. GPGPU programming is also used in many other types of applications for example in scientific simulation applications. Before the time of graphics API abstraction frameworks like OpenGL or Direct3D, developing graphics application was a very time consuming job. To do this, if you wanted your application to run on most computers despite the graphics vendor, you had to write code for all the different vendors.

Even a long time after OpenGL and Direct3D came out you only had what the fixed function pipeline for drawing 3D graphics offered, so if you wanted to do something else besides graphics you were very limited by that. To the developers there were still no god way to get around this problem until OpenGL 2.0 was released that had several programmable stages in this pipeline using GLSL (GL Shading Language) [1]. With this technology you were actually able to do some GPGPU programming. But the shaders were, and still are, optimized for doing graphics processing and not general-purpose processing which means that you were very limited by that.

On 15 February 2007 the CUDA SDK was made public, CUDA stands for Compute Unified Device Architecture and is NVIDIA based programming environment for GPGPU [2]. There is only one big problem with CUDA and it is that you have to have an NVIDIA card to use it. This dramatically limits the targeted group for marketing to the NVIDIA users only.

On the 9th December 2008 the Khronos group released OpenCL 1.0 specification [3]. OpenCL stands for Open Computing Language and was originally developed my Apple. OpenCL is one of the competitors to CUDA along with DirectCompute, developed by Microsoft, and is an open standard for any vendor company to implement for their cards. Programs written in OpenCL can be executed across heterogeneous platforms such as CPU's and GPUs and other types of processors.

2.2 GPU Computing

This new kind of programming with the CUDA and OpenCL is different from shading programming and is often referred to as GPU computing rather than GPGPU [4]. The GPGPU referees more to the general purpose use of shaders. When using shaders in a general-purpose manner the input and output is encoded into textures to be processed by the fragment or pixelshader. This method requires texture look-ups for each element that will be processed and since you cannot both read and write to the same texture buffer you must have extra output texture buffer to store the results as-well. This will double the amount of memory use and limit you to only half the memory.

But with this new technology we can pass on not only textures but normal array buffers as-well and this will negate the extra overhead with the texture look-up and double buffering since you can both read and write to an array.

3 Related work

Previous research made by Pyarelal Knowles from the RMIT University in Melbourne Australia shows that GPU based particle system has significantly higher performance over the ones processed on the CPU [4]. He uses CUDA when implementing his particle systems. His tests cover 3 different types of particles systems as shown in Figure 2, Non interacting particles (NIPS), Short-Range interactive particles (SRIPS) and Long-Ranged interactive particles (LRIPS). These types differ in computational complexities going from O(n) to O(n^2).

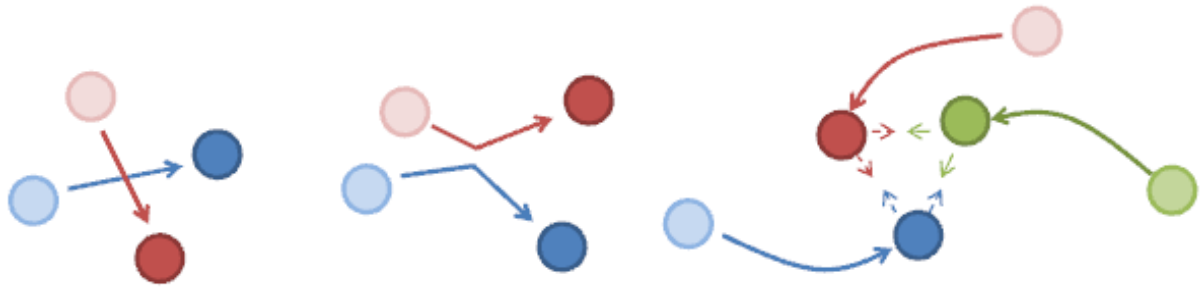


Figure 2: Three different types of interaction. [4]

His test result shows that particle-systems scale very well on the GPU's architecture and for some large particle systems the GPU can run up to 79 times faster than on the CPU. But in some cases where smaller particle systems are used the CPU proved to be faster.

4 Research Objectives

In this research I will try to answer the following questions:

- Is it faster to simulate one big system with 100, 000 particles than 100 smaller systems with 1000 particles in each on the GPU?
- How much is the performance affected when you continuously divide a huge amount of particles into many smaller particle systems?
- When is it more feasible to use the CPU for simulating the particle systems?

4.1 Hypothesis

- When the particle systems become small enough the CPU will be able to outrun the GPU.

This hypothesis are based on the fact that the CPU core is almost always much faster than a single compute unit in the GPU. There may be well over hundreds of these units in a GPU but when the particle systems are too small there are not enough particles to put all these units into work, hence unable to utilize the full potential of the GPUs processing power.

4.2 Methodology

To answer the questions presented in the research objectives I am going to implement a small bench marking application of a simple particle-system that can be simulated on both GPU and CPU using the OpenCL framework. By measuring the time it takes to compute one update-iteration of the particle-system with varying number of particles and instances of the system in each test we should be able to see how the performance is affected.

5 Implementation

For the purpose of this research a simple particle system was implemented using OpenCL. Since OpenCL allows for heterogeneous programming on multiple types of core architectures, the same implementation of the particle-system can be tested on both the CPU and the GPU. This will result in more accurate comparison between the different platforms.

The particle-system implemented for these tests is very simple, both in implementation and computational complexity. The system has only two attributes, position (x, y, z) and velocity (x, y, z). There is no collision between the particles but they do change direction when flying outside their boundary space which is a cube.

Each attribute is store in its own buffer which means that there are two buffers for each particle system, one for positions and one for velocities. These buffers are only copied to the graphics memory once in the initialization part of the program. The only parameter that is being sent to the GPU for every update-iteration is the time passed since last iteration.

For some applications there is reason to retrieve the particle information in each frame to make further simulation on that data. To cover a broader interest for this research the tests also includes doing the simulations with copy-back for each update-iteration.

5.1 Setting up the OpenCL environment

When writing OpenCL applications you must do some initializations before you can start executing you kernels on the GPU. To start with you have to create a cl context and identify the devices (CPU, GPU and etc) you want to use. You can have more than one device in the same context. This can be useful if you want to use all the computational devices in your computer.

You must also create command-queues for each device you want to use. These queues will later be used to enqueue kernels that will be executed. To create kernels we must first load the OpenCL-C code and compile it to a program, alternatively you can load pre-built binaries so you do not have to compile them during run-time. When the program is loaded and compiled you can create a kernel from it. The kernel object you receive in your host application is basically a handle that referees to the kernel function in your cl-c code.

Before enqueuing the kernels you have to set some arguments to it, arguments like buffers or images or just regular values. I use regular one dimensional buffers in my implementation to store the values of my particles. So for each particle-system I create two buffers, one for positions and one for velocities. And then I just pass the time delta as the last argument so I can get time-based movement on my particles.

I only have one instance of the kernel for my entire simulation and only one kernel can be executed at a time. So to update all the particle-systems I have to loop trough them and for each system I pass the arguments onto the kernel and, enqueues the kernel and wait for it to finish before I move on to the next system.

The actual kernel code looks like regular C code but when you are coding in it you have to keep in mind that the kernel will be executed for each work-item (element in the buffer). If it were regular C code and executed in a sequential fashion the code for updating the particles would have looked like this:

```
for(int i = 0; i < 100000; i++)
{
    pos[i] = pos[i] + vel[i]*dtime;
}
```

But the cl versoin will look like this:

```
int nIndex = get_global_id(0);
pos[nIndex] = pos[nIndex] + vel[nIndex]*dtime;
```

As you can see there is no for-loop to go through every element in this code. Instead we use a built in function that tells us the current index of the item we are working on at the moment. This code structure is well suited for stream-architectures that have many executing units because we can execute this kernel in parallel on each unit for a bunch of work items at a time.

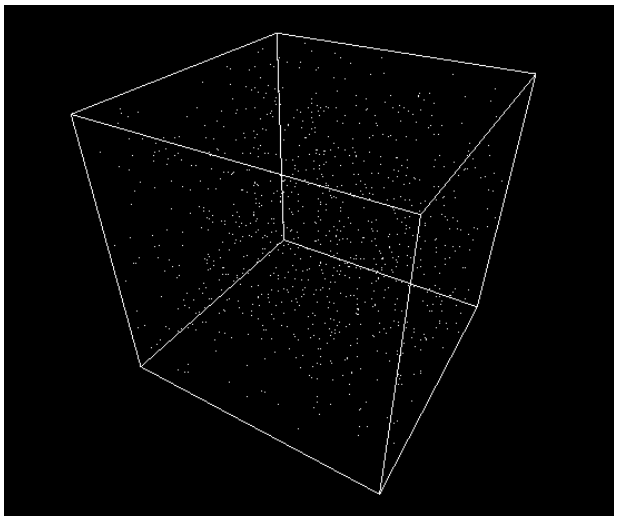


Figure 3: 1,000 particles

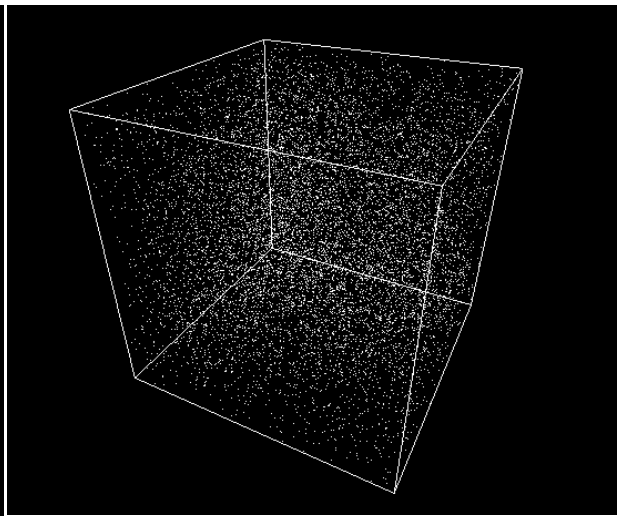


Figure 4: 10,000 particles

Figure 1 and 2 are screen-shots of the real-time visualizations of the particle-systems. The tests will not include the rendering of the systems. The visualizations are just for debugging purposes to assure the correctness of the simulations. The reason why the rendering is not included the tests is because this research aims to show the computational differences between CPU and GPU and not the extra performance gained for already having all the data in GPU memory when it is time to render.

6 Test environment

During the testing phase of my research I used two different computers witch endured the same tests. The specifications for the platform on which the tests were conducted on are the following:

Computer 1:

- CPU Intel(R) Core(TM)2 Quad CPU Q9550 @ 2.83GHz
- ATI Radeon HD 4870 X2
- MS Windows 7

Computer 2:

- Intel(R) Core(TM)2 CPU 6600 @ 2.40GHz
- ATI Radeon HD 5770
- MS Windows 7

7 Results

The time measured between every frame is in milliseconds and not in frames per seconds. This is because there is no rendering in the tests, only the simulation. For each test scenario the delta time for 100 frame iterations are sampled and the result is the average time. All the calculations during the simulation are done with single point precision.

7.1 Test Results on Computer 1

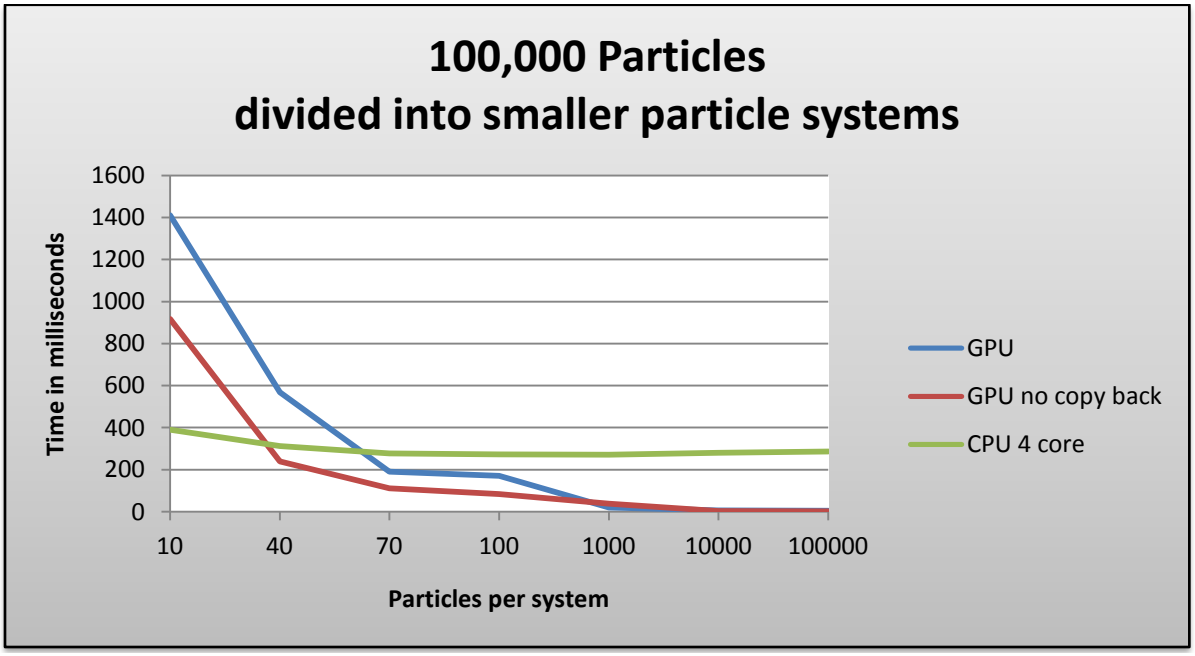


Figure 5: Test results for Computer 1

In Figure 1 we can see that the time for the simulation on the GPU decreases rapidly when we increase the number of particles per system. The CPU however follows a straight line almost the entire time except for when the particles-systems become really small.

7.2 Test Results on Computer 2

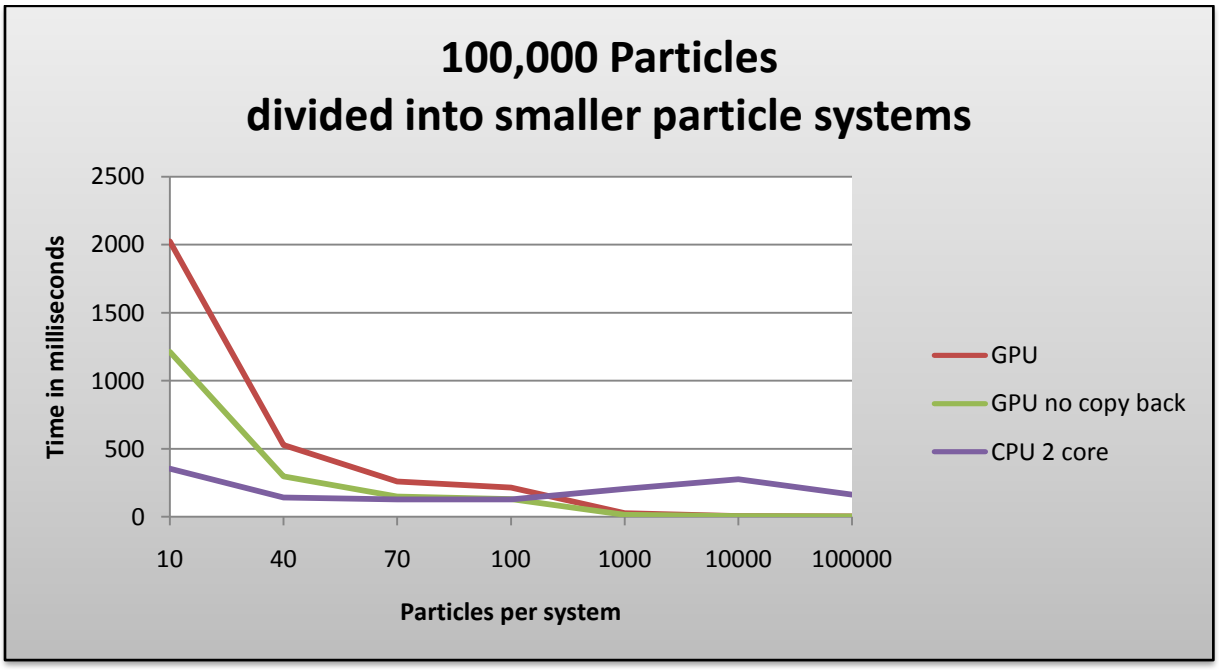


Figure 6: Test results for Computer 2

Figure 6 looks almost the same as Figure 5 and it is the test results for computer 2. The graphics-card used in this test is a later model than the one in computer 1. But the graphic-card in computer 1 has two GPUs so that is why it is faster.

7.3 Discussion

As you can see in Figure 5 and Figure 6, the gap between GPU (with copy back) and GPU with no copy back is really big when there are 10000 particle-systems with 10 particles in each and it becomes smaller when increasing the number of particles. I can think of only two factors that could play a role in this phenomenon and that would be the number of copy calls that has to be executed to retrieve all the data. The second and probably the most important factor would be that all the data I want to copy is not concentrated in one place in the memory since there are two buffers for each particle system, one for positions and one for velocities. When doing the copy back I only read the positions. So after simulating 100000 particles I only have one buffer to copy but when simulating 10 particles per system I have 10000 buffers to copy and in memory between every buffer is probably a gap allocated for velocities that I don't want to copy.

When we look at the CPUs performance the time consumption is basically constant throughout all the tests. This is especially noticeable in Figure 5. I think it is because there are so few cores that none of them ever goes unused. This may be the reason why the GPU is so much slower when the numbers of particles per system are below 40. Since the GPUs often have well over 100 smaller streaming processing units (SPU's, and in this case around 800 units), having lesser number of particles means a lot of those units maybe never gets used.

In Figure 6 the line for the CPU is a bit shakier. A factor that could play a big role in this is that MS Windows is not a real-time system and therefore the tests could have been interrupted by other programs during the tests. The operating system is the same on both computers but they have different software installed that could interfere.

There are possible optimizations that can be done for when the numbers of particles per system is really low. One way could be to rearrange the order of all the particles. For example if you have 10 particles and 10000 systems and you think of the buffers as a 10x10000 matrix when you line them up side by side. You could transpose that matrix and get 10000x10 instead by putting the first particle in each system in one buffer and so forth. So instead of having 10000 small buffers you now have 10 big buffers which will yield the same test results as having 10000 particles per system. This optimization was not implemented since it would defeat the sole purpose of this research because it requires that the kernels for all the systems are the same. For simplicity reasons I have only implemented one kernel but in the research I assume that particles can be of different types and thus have different kernels to update them. This optimization would also make it impossible to have collisions or any other type of particle interaction among particles within the same system.

7.4 Conclusion

To conclude I think it is safe to say that my hypothesis is confirmed. The results clearly show that for smaller particle-systems the CPU is much faster than the GPU. One peculiar thing about the CPU simulation is that it almost never matter how many systems you divide your particles in, the performance stay the same.

The size of the particle-systems can have a huge impact on the performance of the GPU, especially when talking about smaller systems in the range between 1 and 1000 particles. The numbers of particles per system has an exponential effect on the time consumed for simulating them. The smaller the system, the more time it takes for the simulation.

To answer the question: When is it more feasible to use the CPU for simulating the particle systems? I would have to say that it greatly depends on the hardware you are using and whether you need to have the particle data in the CPU memory or not. For the hardware used in these tests the CPU is better than the GPU when the number of particles drop below 40.

8 Future Work

The tests in this research do not take rendering into consideration and is more focused on GPU vs CPU performance. But in games and other more visual applications there is often a mix of both simulation and rendering. In these cases it would be preferable to take advantage of the fact that all the particles are already in GPU memory when simulated and thus no extra data transfer from CPU memory to GPU memory is needed to render it. Future research could involve finding out exactly how much that would affect the performance compared to doing the simulation on the CPU and then sending it to the GPU for rendering. Could it suddenly become more efficient to have even the smaller particle-systems on the GPU?

One thing about OpenCL is that it runs on many different platforms which make it an ideal programming environment for combining the processing power of all the cores in your computer. As they say: “Two heads are always better than one”. Well how much better performance can we get if we for example add the CPU to aid the GPU when simulating a really large particle system?

This report only covers the computation of a very simple particle system with no inter-particle collision or other effects that can alter the time complexity of the simulation algorithm. It might be possible that when you introduce these additional factors to the particle system that the simulation will still be faster on the GPU with even smaller particle systems. For example if you have a particle-system where every particle affects each-other with turbulence or gravitational pull. In this case you have a time complexity of O(n^2).

References

Published research papers

[1] Mark Segal and Kurt Akeley. The OpenGLR Graphics System: A Specification (Version 2.1 - December 1, 2006). opengl.org

[2] Jun Ni. CUDA and OpenCL --- Development Interfaces for Multicore Programming. December 23 2009. uiowa.edu

[3] Khronos Press Releases. The Khronos Group Releases OpenCL 1.0 Specification December 9th 2008. khronos.org

[4] Pyarelal Knowles. GPGPU Based Particle System Simulation. School of Computer Science and Information Technology RMIT University Melbourne, AUSTRALIA November 12, 2009

[5] A. Leist, D. P. Playne and K. A. Hawick†. Exploiting Graphical Processing Units for Data-Parallel Scientific Applications. Computer Science, Institute of Information and Mathematical Sciences, Massey University, Albany, Auckland, New Zealand. December 2008. http://www.massey.ac.nz/

Websites

[6] http://nostasolutions.com/2008/11/03/stream-processing-units-implementation-nvidia-vs-amdati/ last visited 100523

Images

[7] http://developer.download.nvidia.com/pix/tools/cpu_and_gpu_med.png last visited 100524

APENDIX A – OpenCL C code for the particle-system

```
/**
* This is the OpenCL kernel that will update the particles.
*/
__kernel void partsys(  __global float4 * pos,
                        __global float4 * vel,
                        float dtime)
{
   // Get index for our current work object
   int nIndex = get_global_id(0);

   // update position of the particle
   pos[nIndex] = pos[nIndex] + vel[nIndex]*dtime;

   /*
   * Test Collision with the bounding walls of the cube.
   * If position is outside the cube, change the direction
   * so i looks like it bounces of the walls.
   */
   if(pos[nIndex].x<-1.0f)
   {
      pos[nIndex].x = -1.0f;
      vel[nIndex].x = -vel[nIndex].x;
   }
   if(pos[nIndex].x>1.0f)
   {
      pos[nIndex].x = 1.0f;
      vel[nIndex].x = -vel[nIndex].x;
   }
   if(pos[nIndex].y<-1.0f)
   {
      pos[nIndex].y = -1.0f;
      vel[nIndex].y = -vel[nIndex].y;
   }
   if(pos[nIndex].y>1.0f)
   {
      pos[nIndex].y = 1.0f;
      vel[nIndex].y = -vel[nIndex].y;
   }
   if(pos[nIndex].z<-1.0f)
   {
      pos[nIndex].z = -1.0f;
      vel[nIndex].z = -vel[nIndex].z;
   }
   if(pos[nIndex].z>1.0f)
   {
      pos[nIndex].z = 1.0f;
      vel[nIndex].z = -vel[nIndex].z;
   }
}
```

APENDIX B – OpenCL setup code.

```
int OCL_Module::initialize(cl_device_type deviceType, bool log)
{
  if(OCL_Module::mpInstance != NULL)
    return false;

  OCL_Module::mpInstance = new OCL_Module();
  OCL_Module *ocl_module = OCL_Module::getInstance();

  //Initialize OpenCL context.
  cl_int err;
  cl::vector<cl::Platform> platforms;
  err = cl::Platform::get(&platforms);
  if(err != CL_SUCCESS)
  {
    cout << "Unable to retrieve platforms" << endl;
    return SDK_FAILURE;
  }

  cl::vector<cl::Platform>::iterator i;
   if(platforms.size() > 0)
   {
    for(i = platforms.begin(); i != platforms.end(); ++i)
    {
      if(log)
      {
        logPlatformInfo((*i));
      }
      if(!strcmp((*i).getInfo<CL_PLATFORM_VENDOR>(&err).c_str(),
                                        "Advanced Micro Devices, Inc."))
      {
        break;
      }
    }
   }
  if(err != CL_SUCCESS)
  {
    cout << "Platform::getInfo() failed (" << err << ")" << endl;
    return SDK_FAILURE;
  }

  cl_context_properties cps[3] =
    { CL_CONTEXT_PLATFORM,(cl_context_properties)(*i)(), 0 };

  cout<<"Creating a context AMD platform\n";
  ocl_module->mpContext =
       new cl::Context(deviceType, cps, NULL, NULL, &err);
  if (err != CL_SUCCESS)
  {
    cout << "Context::Context() failed (" << err << ")\n";
    return SDK_FAILURE;
  }
  if(log)
  {
    logContextInfo(*ocl_module->mpContext);
  }
    cout<<"Getting device info\n";

  ocl_module->mDevices =
       ocl_module->mpContext->getInfo<CL_CONTEXT_DEVICES>();
  if (err != CL_SUCCESS) {
```

```
    cout << "Context::getInfo() failed (" << err << ")\n";
    return SDK_FAILURE;
  }
  if (ocl_module->mDevices.size() == 0) {
    cout << "No device available\n";
    return SDK_FAILURE;
  }

  if(log)
  {
    cout << "Number of devices found: "
         << ocl_module->mDevices.size() << endl;
  }
  for(size_t deviceIndex=0;
    deviceIndex<ocl_module->mDevices.size();
    ++deviceIndex)
  {
    //log info about the devices.
    if(log)
    {
      logDeviceInfo(ocl_module->mDevices[deviceIndex]);
    }

    // Create a CommandQueue for each device.
    cl::CommandQueue queue(*ocl_module->mpContext,
                   ocl_module->mDevices[deviceIndex],
                   0,
                   &err);
    if(err)
    {
      cout << "Unable to create commandqueue for device[" << deviceIndex
      << "]. ErrCode: " << err << endl;
      return SDK_FAILURE;
    }
    else
    {
      ocl_module->mCommandQueues.push_back(queue);
      if(log)
        cout << "CommandQueue for device[" << deviceIndex
             << "] was successfuly created." << endl;
    }
  }
  return SDK_SUCCESS;
}
```

APENDIX C – Particle system setup and execution code

```
int ParticleSystem::initialize()
{
   cl_int err;
   OCL_Module *ocl_module = OCL_Module::getInstance();
   this->mpPositions =
      (cl_float*)malloc(this->mNumParticles * sizeof(cl_float4));
   this->mpVelocities =
      (cl_float*)malloc(this->mNumParticles * sizeof(cl_float4));

   // Set initial positions.
   for(int i=0; i<this->mNumParticles; i++)
   {
      int index = 4*i;

      for(int j=0; j<3; j++)
      {
         this->mpPositions[index+j] = this->random(-1.0f, 1.0f);
         this->mpVelocities[index+j] = this->random(-1.0f, 1.0f);
      }
   }

   // Create mem buffers
   mPosBuff = new cl::Buffer(*ocl_module->getContext(),
      CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
      this->mNumParticles*sizeof(cl_float4), this->mpPositions, &err);

   if(err != CL_SUCCESS)
   {
      cout << "unable to create mPosIn" << endl;
      return FAILURE;
   }
   mVelBuff = new cl::Buffer(*ocl_module->getContext(),
      CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
      this->mNumParticles*sizeof(cl_float4), this->mpVelocities, &err);

   if(err != CL_SUCCESS)
   {
      cout << "unable to create mPosOut" << endl;
      return FAILURE;
   }

   return SUCCESS;
}

int ParticleSystem::updateKernel(cl::Kernel &kernel, cl_float dtime)
{
   cl_int err;
   err = kernel.setArg(0, sizeof(cl_mem), mPosBuff);
   if(err != CL_SUCCESS)
   {
      cout << print_cl_errstring(err) << endl;
      cout << "Unable to set kernel arg 0" << endl;
      return FAILURE;
   }
   err = kernel.setArg(1, sizeof(cl_mem), mVelBuff);
   if(err != CL_SUCCESS)
   {
      cout << "Unable to set kernel arg 1" << endl;
      return FAILURE;
   }
```

```
    err = kernel.setArg(2, sizeof(cl_float), (void*)&dtime);
    if(err != CL_SUCCESS)
    {
        cout << "Unable to set kernel arg 4" << endl;
        return FAILURE;
    }
    return SUCCESS;
}

int ParticleSystem::executeKernel(cl::Kernel &kernel)
{
    cl_int err;

    // Execute kernel.
    cl::Event progEvent[2];
    cl::NDRange globalWorkgroup(this->mNumParticles);
    cl::NDRange localWorkgroup(1);
    err = this->mCommandQueue.enqueueNDRangeKernel(kernel,
                                cl::NullRange,
                                globalWorkgroup,
                                localWorkgroup,
                                NULL,
                                &progEvent[0]);
    //queue.enqueueNDRangeKernel
    if(err != CL_SUCCESS)
    {
        cout << "Unable to enqueue kernel: err = " << err << endl;
        return FAILURE;
    }

    // wait until finnished and read result.
    err = progEvent[0].wait();
    if(err != CL_SUCCESS)
    {
        cout << "Failed to wait" << endl;
        return FAILURE;
    }

    err = this->mCommandQueue.enqueueReadBuffer(*this->mPosBuff,
                               1,
                               0,
                               this->mNumParticles*sizeof(cl_float4),
                               this->mpPositions,
                               0,
                               &progEvent[1]);
    if(err != CL_SUCCESS)
    {
        cout << "failed to enqueu ReadBuffer: err = " << err << endl;
        return FAILURE;
    }

    // Wait for the read buffer to finish execution.
      err = progEvent[1].wait();
    if(err != CL_SUCCESS)
    {
        cout << "Failed to wait for ReadBuffer" << endl;
        return FAILURE;
    }

    return SUCCESS;
}
```

APENDIX D - Test Result Data

The particles column describes the numbers of particles per particle-system. The time is measured in milliseconds.

Computer 1

Intel(R) Core(TM)2 Quad CPU Q9550 @ 2.83GHz
ATI Radeon HD 4870 X2
MS Windows 7

Particles	GPU	GPU no copy back	CPU 4 core
10	1410,04	917,158	390,085
40	569,028	240,252	311,908
70	190,942	112,394	277,941
100	170,523	84,783	273,214
1000	20,234	38,644	271,126
10000	6,5	3,433	280,309
100000	5,249	2,645	285,619

Computer 2

Intel(R) Core(TM)2 CPU 6600 @ 2.40GHz
ATI Radeon HD 5770
MS Windows 7

Particles	GPU	GPU no copy back	CPU 2 core
10	2025,167	1212,686	354,58
40	529,582	297,444	142,749
70	260,237	148,841	129,975
100	214,93	129,873	128,812
1000	27,073	14,294	205,846
10000	6,058	3,812	275,907
100000	4,344	2,577	163,9