GPGPU Based Particle System Simulation*

Pyarelal Knowles

Supervisor: Geoff Leach

School of Computer Science and Information Technology
RMIT University
Melbourne, AUSTRALIA

November 12, 2009

Abstract

General purpose computing on graphics processing units, known as GPGPU but now often referred to as GPU computing, is the approach of performing computation on the GPU instead of the CPU. GPU computing has been made possible by the increasing programmability and performance of GPUs. The programmability of GPUs is accessed via shader programs, typically written in a C like language. Until recently shader programs and shader languages have been targeted (naturally) towards graphics applications. However there has been increasing interest in using GPUs for non graphics computationally intensive tasks. This has lead to new languages and tools for GPU development which are more general than shaders. For example, Nvidia released its compute unified device architecture (CUDA) in 2007. CUDA allows what is essentially general purpose C code to be executed on GPU processors, without the graphics orientation found in shader languages. More recently Open Compute Language (OpenCL) has been proposed for GPU computing, and more generally for CPU-GPU hybrid computing.

This research aims to compare performance of particle system computation between CPU and GPU based implementations. Particle systems differ in their types of inter-particle interaction. The types of interaction between particles has a direct effect on the computational complexity of the particle system and therefore affects performance. As such, for the purpose of this research, we identify three distinct categories of particle systems: no interaction, short range interaction (collisions) and long range interaction. We implemented each of these types of particle systems for the CPU (including multicore versions) and the GPU (using CUDA). We have found that a significant performance increase may be obtained from GPGPU based compared with CPU based particle system computation.

*Honours research paper

Contents

1 Introduction

Figure 1: Star Trek II: The Wrath of Kahn - The genesis effect.

The term particle system in the context of computer graphics was coined in 1983 by William T. Reeves [Ree83] when he used particle systems in the movie "Star Trek II: The Wrath of Kahn" to make the genesis effect (Figure 1). Before that, even though not referred to as such, particle systems were used in some of the very first video games, for example "Spacewar!" in 1962 used particles to display explosions (Figure 2). Since then the use of particle systems in games has continued to increase and they are used in most games today in some way. Games are required to have fast execution in order to have smooth animation and responsive interaction. Thus any particle systems used must be able to be computed quickly which is why games today typically use non interacting, or independent particles. These are considerably less computationally intensive than systems of interacting particles which can have more realism. Not all particle systems run in "real-time" such as in games. For example movies such as "Star Trek II" calculate the system and pre-render each frame of animation.

Figure 2: Early video game, Spacewar!

Particle systems can be used to model an object as a cloud of primitives that define its volume [Ree83]. Each particle has a position in space and moves based on initial position, velocity and acceleration values along with forces. The collection of particles come together as a whole to represent shapes. There are many uses in games and graphics for particle systems including water, smoke, dust, fire or cloud and even hair and cloth simulation, as shown in Figure 3.

While games and graphics are the primary interest in our investigation, particle systems also have many uses in science which this research also applies to. For example cosmological simulations use particle systems of more than tens of millions of particles [Ber98] to study theories about the creation of the universe among many others. Particle simulations are used in research for large and costly fusion reactors [TZ99]. Medical training simulations use particle

(a) Fire (FarCry2) (b) Smoke (NFS Pro-Street) (c) Water[1]

(d) Cloth[2] (e) Dust/Smoke (ID's Rage) (f) Hair [VMT04]

Figure 3: Particle System Examples

systems to simulate blood in virtual surgery [MST04]. Particle systems are used in molecular modeling [LHvdS01]. Many of these examples require particle systems of higher computational complexity and larger numbers for accurate simulation and realism. The computational complexity of a particle system has a fundamental impact on performance and hence limits the size of the particle system.

There is a need for larger and more realistic particle systems, which will be more computationally intensive. As the performance of particle systems increases with both algorithm and hardware, particle system applications will improve. For example, games today typically do not perform fluid simulation using particle systems, yet with the introduction of GPGPU computation, the game physics engine PhysX can now perform substantial real-time fluid simulation [Har08], which means this approach will begin to emerge.

Research Questions

Particle systems have traditionally been implemented on CPUs. However due to a recent improvement in the evolution of GPU architecture they have begun to be computed using GPUs. It is important to know what the implications are in moving to a GPU processing environment. Previous research shows that the GPU has the potential to process some types of particle systems much faster than some CPU implementations. However before we start processing particle systems on the graphics card we should be aware of the intricacies and consequences in different situations, including performance trade-offs.

We investigate CPU and GPU particle system computation in order to answer the following questions:

1. What are the performance improvements, if any, from the use of GPUs instead of CPUs for particle systems computation?

[1]From: `http://people.csail.mit.edu/acornejo/html/cloth.htm`
[2]From: `http://area.autodesk.com/blogs/duncan/pouring_water_with_nparticles`

2. How does the introduction of a graphics rendering load (common to games and graphics) affect these results?

3. What are the programming issues involved with GPU computing for particle systems in particular?

Primarily, this research seeks to determine the circumstances under which it is more appropriate to compute particle systems using the GPU and when use of the CPU is preferable. Results will be focused towards games and graphics however it is intended that findings will also be relevant to broader applications.

2 Background

2.1 Particle Systems

The simulation of a particle system involves processing every particle's movement in discrete timesteps. In a graphics environment each timestep is referred to as a frame, in which the scene is recalculated and rendered. The processing of each particle generally involves integrating the equations of motion. This includes integration of velocity and acceleration from initial or applied forces. Performing this integration based on time in discrete steps requires the use of numerical integration. Methods for this are described later in Section 2.17

There are many different types of particle systems. We categorize them into three types based on interactions as shown in Figure 4. The types of interactions in a particle system have direct implications for the computational complexity of the algorithm.

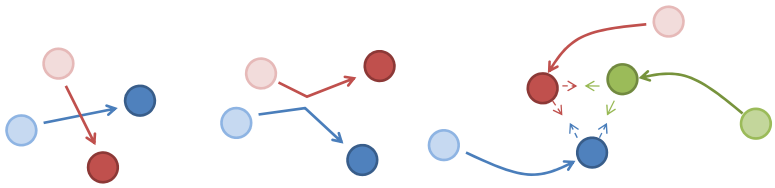


Figure 4: Different types of interaction.

2.1.1 NIPS

Non-interacting particle systems (NIPS) have no interaction between particles, that is, particles are independent of each other. These particle systems have a computational complexity of $O(n)$ per step (Algorithm 1). These are the least computationally complex particle systems.

for $i = 0$ to $N - 1$ **do**
  move particle i
end for

Algorithm 1: NIPS

2.1.2 SRIPS

Short range interacting particle systems (SRIPS) have particles that interact with their neighbours, that is, other particles within a short range. One form of these interactions, and the

one we have used, is collisions, where the particles are hard spheres which bounce off each other. The brute force method for the computation of this type of particle system is given in Algorithm 2.

for $i = 0$ to $N - 2$ **do**
  move particle i
  for $j = i + 1$ to $N - 1$ **do**
    check collision between particle i and j
  end for
end for

Algorithm 2: SRIPS brute force

Spatial data structures (discussed later in Section 2.16) can be used in this case to quickly determine close or neighbouring particles, reducing the computational complexity from the brute force $O(n^2)$ method to $O(n)$, shown in Algorithm 3. The inner loop of the Algorithm 3 should, depending on the type of spatial data structure and distribution of particles, take a constant time, and hence reduce the complexity to $O(n)$, discussed in Section 2.16.

for $i = 0$ to $N - 1$ **do**
  move particle i
  for all j in neighbour(particle[i]) **do**
    check collision between particle i and j
  end for
end for

Algorithm 3: SRIPS using a spatial data structure

2.1.3 LRIPS

Long range interacting particle systems (LRIPS) are where every particle is affected by every other particle as occurs with a gravitational or electrostatic force. These systems then have computational complexity of $O(n^2)$, or more specifically $\Omega(n^2)$, and require the brute force approach as given in Algorithm 4.

for $i = 0$ to $N - 1$ **do**
  for $j = 0$ to $N - 1$ where $j \neq i$ **do**
    compute interaction between j and i
  end for
  move particle i
end for

Algorithm 4: LRIPS

2.2 Rendering

Applications, such as games, may require the particle system to be rendered graphically as well as computed. The rendering may happen after the calculation step in each frame or after multiple frames. The rendering approach can range between simple unshaded opaque points to transparent billboards that require depth sorting or surfaces that need geometry generated

from the particle cloud. This extra rendering load added to the particle system computation is also an important factor as it will affect the performance of applications involving display.

2.3 Computer Architecture

Investigating and optimizing performance generally requires understanding computer and graphics architecture. To make processes and algorithms efficient they should be shaped to fit the architecture. The process of computing and rendering particle systems may include data transfer from hard disk to system memory to video memory and vice-versa. Some parts of the process may be computed on the CPU and others on the GPU. It is important to know how each of these components relate to each other and how they are interconnected.

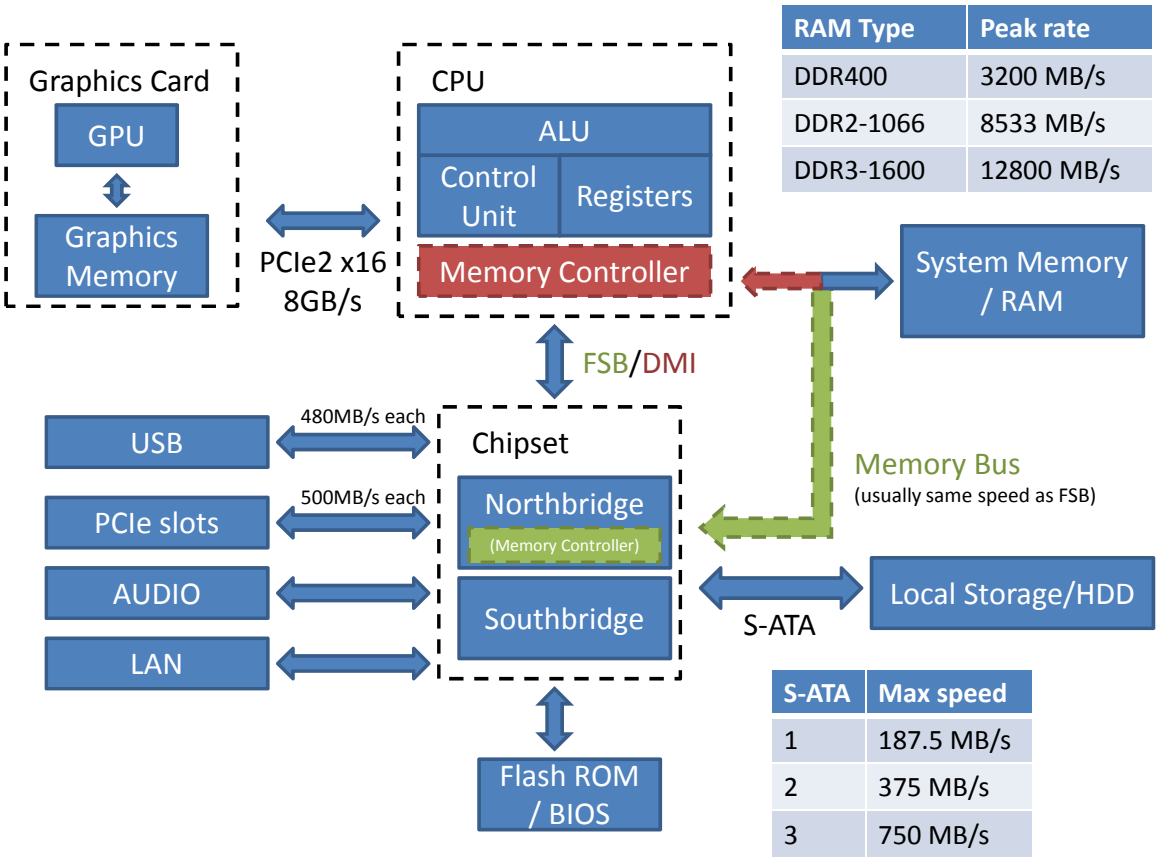


Figure 5: Simplified computer architecture.[3]

Figure 5 shows the basic architecture of today's desktop computers. An important aspect of this architecture is the different bus speeds between components. Previously the northbridge included the memory controller (shown in green) and also handled data transfer to the graphics card via the PCIe bus. Recently both memory controller and the graphics card bus have been relocated to the CPU itself (shown in red). These changes are designed to improve performance and reduce latency between the CPU, system memory and the graphics card. The southbridge, also known as the I/O (input/output) controller hub, bridges the lower performance components of a computer and the northbridge. This includes data transfer to hard drives, lower speed PICe

[3]RAM speeds from: `http://www.crucial.com/support/memory_speeds.aspx`

slots, LAN (local area network) and USB host.

In the example of a particle system that is computed on the CPU which is then rendered, the data in system memory must constantly be transferred into CPU registers, operated on and sent back to system memory. The resulting data then needs to be transferred to graphics card. Currently computers use the PCIe (Peripheral Component Interconnect Express) bus to transfer data to the GPU. While this is a huge improvement from older AGP or PCI buses this is still a potential bottleneck. Figure 6 shows transfer speeds of PCIe versions for differing numbers of lanes. As shown, doubling the number of lanes in a PCIe bus doubles its speed.

PCIe Version	1 lane	8 lanes (x8)	16 lanes (x16)
v1.x	250 MB/s	2 GB/s	4 GB/s
v2.0	500 MB/s	4 GB/s	8 GB/s
v3.0	1 GB/s	8 GB/s	16 GB/s

Figure 6: PCIe transfer speeds.

2.4 Graphics Pipeline

The graphics pipeline, as shown in Figure 8(a), relates to the process of rendering graphics. Typically, this takes a 3D scene and generates a 2D image representation. Commonly a scene is made up of models or geometry made from a mesh of polygons. The polygons' 3D coordinates are transformed into screen coordinates and rasterized to pixels. The pixels are then coloured and the result is then displayed or saved depending on the render target. This process is shown in Figure 7.

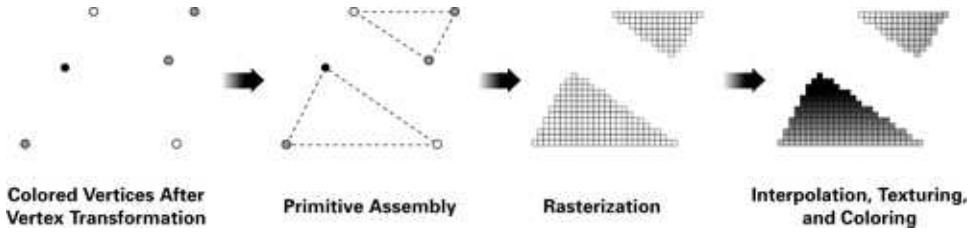


Figure 7: Basic rendering process.[4]

The rendering process starts with some geometry specified as vertices along with connectivity to form graphics primitives, such as points, lines and polygons. This represents the geometry in object space. This data is then sent to the graphics card. PCIe version 1 transfers data at a rate of 256MB/s per lane. Most graphics cards use 16 lane PCIe giving a total of 4GB/s. PCIe version 2 which is now becoming common is double that at 8GB/s.

The next step is to transform the geometry into eye space so it is relative to the viewer or virtual camera. This transformation includes world space and camera space transformations that move the object to its position in the scene and then that position relative to the camera. This transformation is referred to as the modelview transform.

At this point vertices may be coloured, usually by performing lighting calculations. Two common methods for applying lighting calculations are Phong [Pho75] and Blinn-Phong [Bli77]

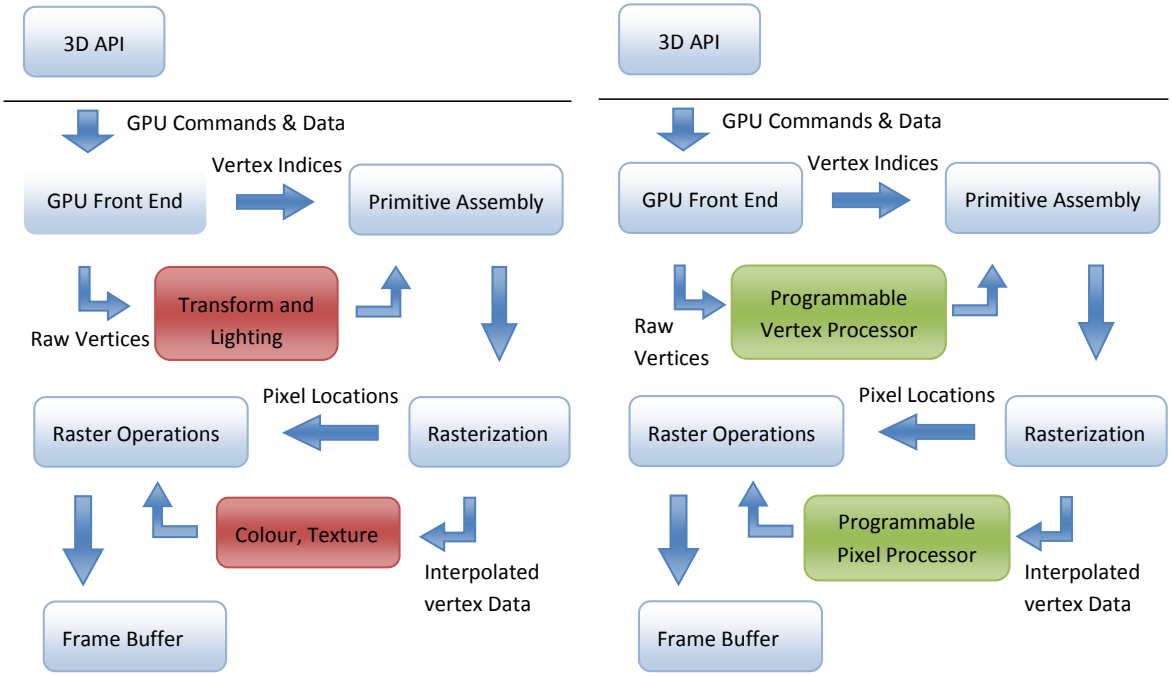
[4]From: `http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter01.html`

lighting.

The projection transformation is then applied. This applies a projection such as perspective to the model, resulting in the object being transformed into screen space. Clipping is performed to remove primitives outside the 2D image area.

The object in screen space is essentially 2D vector based graphics. This geometry must be rasterized to determine the correct pixels for each primitive. At this point each pixel is given a colour based on interpolated data from the vertex colours. Pixels may be further colored using a texture.

Finally, raster operations may be applied, for example depth testing so objects are not drawn over the top of each other, that is to say the hidden surface problem is solved, and blending to draw transparent objects.



(a) Fixed pipeline (b) Programmable pipeline

Figure 8: Graphics rendering

2.5 CPU Evolution

Ever since the first CPU, hardware has increased in power at an exponential rate, as predicted by Moore's law [Moo65]. The CPU, designed for general purpose processing, has previously evolved as a faster single core processor, largely through increasing clock rates. More recently in order to keep increasing performance, CPUs have shifted from faster single cores towards increasingly multicore processors. Current generation CPUs have 4 cores but 6 and 8 core CPUs are expected to be widely available within a year or so.

Some games have taken advantage of multicore CPUs, implementing multithreaded engine components. For example the Source engine includes a multithreaded particle systems [5].

[5]Reference: http://source.valvesoftware.com/SourceBrochure.pdf

2.6 GPU Evolution

Graphics acceleration hardware technology has been around for 30 years but the first GPU, as we know them now, is generally regarded to be the Nvidia Geforce 256, released in 1999. This was the first GPU to perform hardware transform and lighting calculations which were previously performed on the CPU. Moreover it was the first single chip GPU to accelerate the entire (fixed) graphics pipeline, shown in Figure 8(a), in hardware.

In 2001 Nvidia released its Geforce 3 GPUs. These GPUs were the first to include programmable vertex and pixel shaders units (Figure 8(b)). Similar to the previous fixed function GPUs (Figure 8(a)) these GPUs have separate vertex and pixel units, except they are now programmable. The shader programs that ran on the Geforce 3 series GPUs were written in assembly language. This made writing shaders a complex task without access to high level programming tools (further discussed in Section 2.8).

The unified shader architecture combines vertex and pixel shader units into one processor. The advantage of this is a scene with imbalanced vertex to pixel operations will not suffer a performance loss. ATI were the first to produce a GPU with this capability with the Xenos GPU used in Microsoft's Xbox 360. Soon to follow was Nvidia's G80 series GPUs (released in 2006) which were the first GPUs with unified shaders for the PC. They were also the first with unified shaders to fully support DirectX 10. The G80 series were also the first GPUs with CUDA support. The G90 series GPUs were released in 2008 and are very similar to G80 GPUs. Nvidia's GT200 GPUs (released in 2008) are currently the latest generation and include more advanced CUDA functionality.

Intel's recent high end quad core CPU, the i7, has an estimated 731 million transistors, which may be compared with Nvidia's GT200 GPU which has 1.4 billion transistors and 240 "cores". Nvidia have just announced their latest GPU: the GT300 or "Fermi" with 3 billion transistors which is expected to be available in late 2009 or early 2010. GPUs are increasing in their number of transistors and similarly computational power at greater than the exponential rate Moore's law according to Matt Pharr [PF05].

Intel, which has not been in the discrete GPU market, has announced its intention to enter with a new chip called Larrabee, marketed as a GPU with 1.7 billion transistors. Its design is targeted both toward GPGPU and graphics rendering, however much hardware based, graphics specific functionality has been removed in preference the idea of software based rendering on a fully programmable GPU.

2.7 Stream Processing

The rendering process involves many repetitive transformations and colour operations on vertices and pixels. Due to the independent nature of these operations it is possible to divide and stream these operations amongst many processors. Stream processing is a limited form of parallel processing where separate processors do not need to communicate or synchronize. To exploit this attribute GPUs are built with many processors (now well into the hundreds range).

2.8 Shaders

The introduction of programmable shaders was a major step in GPU evolution. Shaders are small C-like programs that can be run on the shader units in a GPU. Since this development, many graphics techniques stopped depending on specific fixed functionality and use of obscure hacks to work, and more generally, has made possible a whole range of new rendering approaches.

In 2002 the OpenGL Architecture Review Board (ARB) created OpenGL shading language (GLSL) as a high level shading language to give access to the programmable graphics pipeline without the use of low level assembly language. GLSL was added as an extension to OpenGL 1.4. Later in 2004 GLSL was formally included as part of OpenGL 2.0. Microsoft developed its high level shading language (HLSL) for DirectX and also worked closely with Nvidia to develop the Cg shading language at around the same time.

The unified shader model uses a consistent instruction set across all shader types. This means much of the same capabilities that exists in vertex shaders also exists in fragment shaders. This gives a much more flexible framework to develop in but also improves GPU performance.

The general use of shaders for GPGPU uses pixel or fragment shaders. The output data must be encoded in pixels of a texture. Data can be computed by rendering to this texture using shaders. For example input data is available as a texture. Then a full screen polygon is rendered to an output texture. Given the position of the output pixel and some data from the input texture each fragment process can do some computation and write the result into colour channels of the pixel. There are limitations to using shaders for general purpose computing. All data involved must be encoded in textures. There is extra overhead for texture look-ups compared to reading a variable and the bandwidth in transferring texture data between system and graphics memory is costly.

While each GPU core may not be as powerful as a CPU given their number in today's GPUs, they can outperform the CPU for some applications. Applications that do this are ones that scale well to the streaming nature of the GPU. They are the ones which require little effort to divide into smaller tasks that can be spread evenly amongst processors running in parallel. Such tasks are named embarrassingly parallel. Particle systems in general have the potential to do this.

2.9 GPU Computing

ATI has recently released ATI Stream (formerly Close To Metal) and Nvidia has released CUDA (Compute Unified Device Architecture). These interfaces allow general purpose C-like code to be run on the GPU. This enables a much more flexible form of GPGPU programming and is why it is being termed GPU computing in order to differentiate from GPGPU, and is the next step in the evolution of programmable GPUs. Standard data types can be used instead of textures. For example an array of floating point values can be copied to video memory and operated on directly. This means there is no overhead for texture look-ups. There is shared memory between GPU cores. Unlike shaders, threads can communicate with each other. There are also atomic operations that allow writes to the same block of memory.

To clarify, the term GPGPU now relates more to shader based GPU computation where as the new term, GPU computing, refers to the use of non shader development environments such as CUDA.

2.10 CUDA

GPUs supporting CUDA have a compute capability which defines the minimum level of CUDA functionality they support. The first few Nvidia G80 GPUs support compute capability 1.0. 1.1 compute capability gives access to atomic operations. 1.3 adds support for double precision floating point numbers.

CUDA is designed with the intent of perfect parallel scalability. The following example is based on the template project from the CUDA SDK. A CUDA kernel is the program that is

executed on GPU cores. It is similar in code to a C function:

```
__global__ void
testKernel(float* idata, float* odata)
{
   //identify this instance using blockIdx and threadIdx
   //operate on idata and write to odata
}
...
cudaMalloc((void**)&inputData, mem_size);
cudaMalloc((void**)&outputData, mem_size);
cudaMemcpy(inputData, hostData, mem_size, cudaMemcpyHostToDevice);

testKernel<<< grid, threads, mem_size >>>(inputData, outputData);
```

The `__global__` keyword tells the CUDA compiler that that function is a kernel. Before executing this kernel graphics memory must be allocated and populated. The data in `hostData` is the input data for computation in system memory. This is copied to `inputData` which is in graphics memory. The kernel is then called using pointers to graphics memory data.

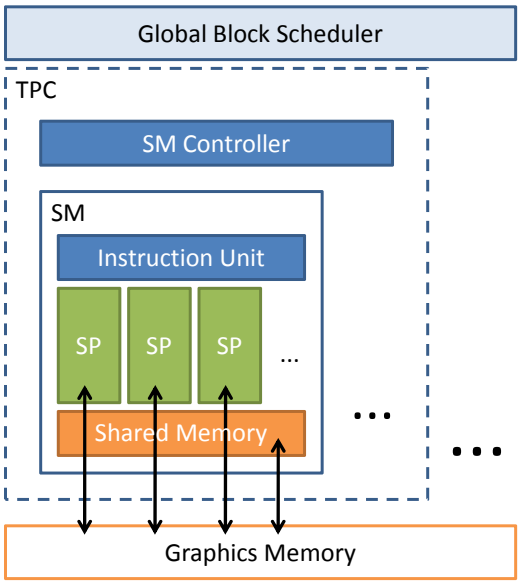


Figure 9: Nvidia GPU architecture for CUDA

When a kernel is executed it is distributed across thread processing clusters (TPCs) by the global scheduler. Each TPC has a number of streaming multiprocessors (SMs) and an SM controller. SMs each contain an instruction unit and number of stream processors (SPs), previously referred to as GPU cores. This hierarchy is shown in Figure 9. An SM creates threads, or "warps", and schedules them across SPs. Each SP is equivalent to a CPU's arithmetic logic unit (ALU) and is where the actual computation is performed. There is no overhead for thread scheduling and a GPU with more SMs can execute the same kernel in less time automatically.

The Nvidia G80 and G90 GPUs have two SMs for each TPC, where as the GT200 GPUs have three SMs per TPC. Two graphics cards we have used in our investigation are the Geforce 9600GT (with a G90 series GPU) and the GTX275 (with a GT200 series GPU). The G90 GPU

on the 9600GT graphics card has 8 SMs (4 TPCs) and 8 SPs per SM giving 64 SPs in total or 64 GPU cores. The GT200 GPU on the GTX275 graphics card has 30 SMs (10TPCs) and 240 SPs.

2.11 GPU Based Physics

In 2006 Ageia released its PPU (physics processing unit) — the so called PhysX chip. Ageia's physics engine PhysX would offload computation to the PPU just as rendering had been offloaded to the GPU. This was then retargeted to the GPU using CUDA when Nvidia bought Ageia.

An edition of the Havok game physics engine, Havok FX was created that offloaded effect physics (physics that do not affect game-play) to the GPU using shaders. Havok was bought by Intel and the Havok FX edition may have been canceled. Bullet, created by Erwin Coumans who previously worked on the Havok engine is an open source physics engine that also utilizes CUDA. AMD has recently formed a partnership with Intel owned Havok to incorporate the use of AMD's ATI GPUs for physics processing.

2.12 OpenCL

OpenCL is a framework that abstracts specific processor architecture. This allows code to be run independently of the device. For example a disadvantage of CUDA is it requires a CUDA enabled Nvidia graphics card. CUDA code will not run on ATI cards. However OpenCL, given driver support, would allow the same set of instructions to be run on an Nvidia or ATI GPU, or on the CPU. OpenCL may also allow for a high level load balancing between different processors.

Apple originally worked on OpenCL and then later the Khronos Group released the official OpenCL 1.0 specification in 2008. Both Nvidia and ATI have recently released the first drivers to support OpenCL operation on their GPUs.

2.13 Single vs Double Precision

IEEE (Institute of Electrical and Electronics Engineers) standardized floating point formats in the IEEE 754 specification [Ste81]. Two of these floating point standards are the most widely used in computer hardware languages:

- Single precision or "float" in the C language (4 byte floating point value). This stores approximately 7 decimal places.
- Double precision or "double" in the C language (8 byte floating point value) This stores approximately 15 decimal places.

CPUs generally perform single and double precision calculations in similar time. A limiting factor is the data transfer as double precision data is larger, which can make double precision computation slower.

Most recent GPUs these days are optimized for single precision operation and there has been a lack of support and performance for double precision. Only more recent GPUs, for example Nvidia's GTX260 and later support double precision. Although these GPUs are able to perform double precision operations their speed is far less than that of single precision. The next generation of GPU from Nvidia is intended to support faster double precision. Figure 10 shows preliminary results from the GT300 (Fermi) whitepaper. A 400% speed increase in double

precision computation speed can be seen when comparing Fermi with the GT200 series. This makes GPUs much more useful for GPGPU scientific applications which require high floating point accuracy.

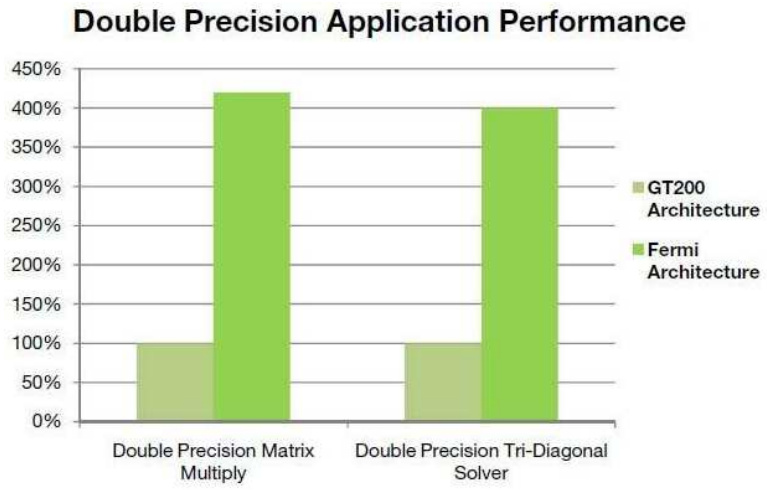


Figure 10: Nvidia performance evaluations for the GT200 and GT300 (Fermi)

2.14 Performance

In computer science, performance of data structures and algorithms are commonly analysed in terms of their computational complexity. However in computer graphics it is common to analyse performance using a performance benchmark approach where computational complexity analysis is not possible. We use both approaches in this thesis although the focus is on performance benchmarking and computational experiments.

2.15 Parallel Programming

For some problems it is possible to divide a task amongst multiple processors so that the processing time is divided by that number of processors. Some applications are better suited to this than others. For example a step in an algorithm may be dependent on a previous step, forcing processors to wait on others. This part of the algorithm is serial and a synchronization between processors must be made. In the case of parallelisation, Amdahl's law [Amd67] states that the total speedup from parallelising an algorithm is

$$\frac{1}{(1-P) + \frac{P}{N}} \tag{1}$$

Where P is the proportion of the program that can be made parallel, $(1-P)$ is then the serial portion, and N is the number of processors. The result of this law is that no matter how many processors are used to compute an algorithm, it will never be faster than the serial portion.

The portion of a program that can be parallelised has a big impact on the speedup that can be obtained as shown in Figure 11 (red is the portion of a program that must be serial). In the case where a large portion of a program cannot be parallelised the overhead from paralellising the implementation may even reduce overall performance.

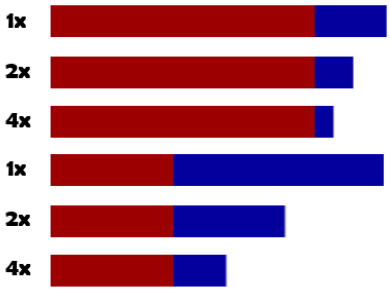


Figure 11: Parallel speedup.

2.16 Spatial Data Structures

One difficulty with short range interacting particles is the need to determine only nearby particles without checking distances to all particles. A spatial data structure (SDS) groups objects into containers (leaf nodes/buckets) based on position. To find a particle's neighbours the algorithm only needs to search containers that the particle intersects. Every time particles move the SDS needs to be updated and hence must be done every frame.

Two common SDSs are the uniform grid and the k-D tree (k-dimensional tree). The uniform grid divides space evenly along each dimension. More advanced implementations can dynamically change divisions in each dimension to compensate for uneven distribution. The uniform grid performs better when objects are evenly distributed.

The k-D tree is a binary tree that divides space where needed. If a leaf of the tree has too many objects a split is made at the center of the objects in the leaf, creating two leaves. At every branch the axis of the split rotates (the first split is in the X axis, second in Y etc.). Because the structure divides space in the center of the objects it does not matter whether the objects are evenly distributed or not.

Figure 12 shows the uniform grid and the k-D tree and their storage method. It is better to use a uniform grid when objects are evenly distributed as it is easier to implement and faster. If objects are not evenly distributed it may be necessary to use a different SDS such as the k-D tree which adapts to the distribution. Figure 13 shows implementations of these SDSs: Figure 13(a) shows a uniform grid dividing evenly distributed particles while Figure 13(b) has unevenly distributed particles and shows the use of a k-D tree with a higher concentration of subdivisions closer to the cluster of particles in the cube corner.

In this work we use the uniform grid as the particles in the SRIPS implementations are evenly distributed. Due to time constraints we were unable to investigate situations requiring the use of a k-D tree.

2.17 Numerical Integration

Particle systems involving position, velocity and acceleration require numerical integration of the equations of motion in order to animate smoothly and accurately. The most basic numerical integration technique is the Euler forward method, commonly referred to as the Euler method

$$\begin{aligned} V_{n+1} &= V_n + hA_n \\ P_{n+1} &= P_n + hV_n \end{aligned}$$

where P is position, V is velocity, A is acceleration and h is the time between steps n and $n+1$. This adds time multiplied by the current gradient at step n for both position and velocity. The

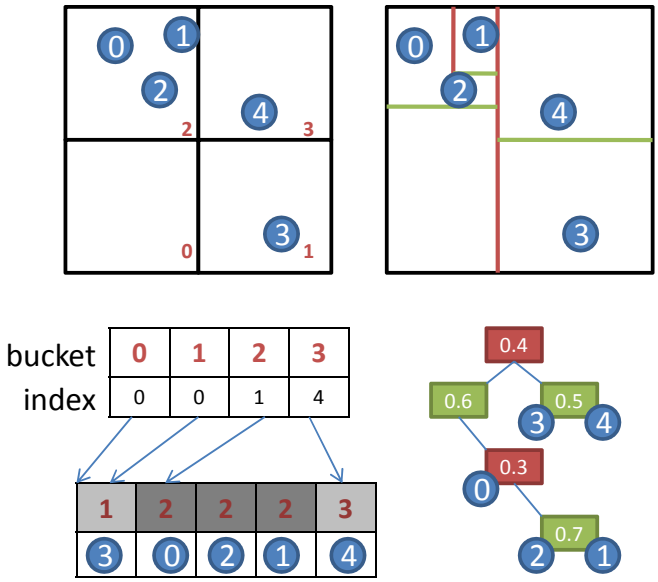
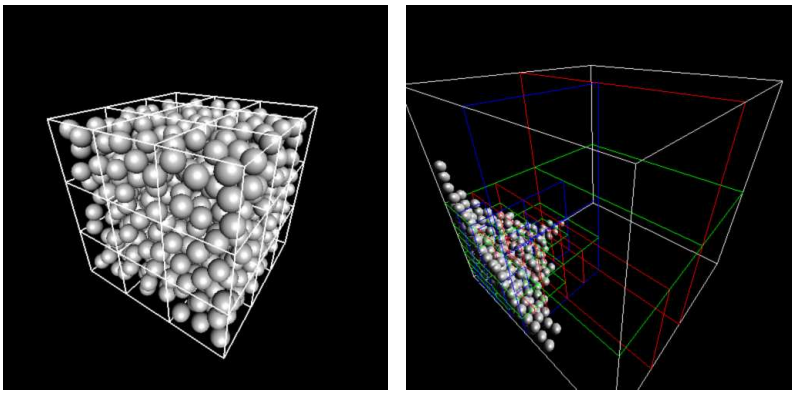


Figure 12: Uniform grid and k-D tree storage diagram



(a) Uniform grid (b) k-D Tree

Figure 13: Spatial data structures in use

euler method gives an accurate velocity for constant acceleration or an accurate position for constant velocity but becomes very inaccurate when one varies or the timestep h becomes large, as can be seen in Figure 15. The improved Euler method, also known as integrating using the mid-point, uses the average of the gradient at n and $n+1$ instead of just the gradient at n to approximate the integral and has better accuracy

$$\begin{aligned} V_{n+1} &= V_n + hA_n \\ P_{n+1} &= P_n + h\frac{1}{2}(V_n + V_{n+1}) \\ &= P_n + h\frac{1}{2}(V_n + V_n + hA_n) \\ &= P_n + h(V_n + \frac{h}{2}A_n) \end{aligned}$$

The error of the improved Euler method is far less as can be seen in Figure 15. This figure shows the vertical position of a particle moving over time with initial acceleration $A_0 = -9.8$ and velocity $V_0 = 10$.

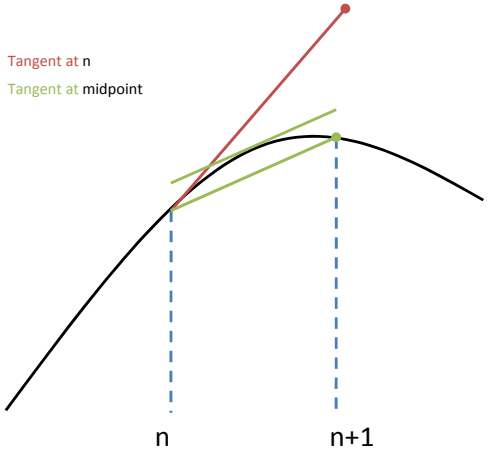


Figure 14: Tangents of a curve. Represents integration at discrete timesteps.

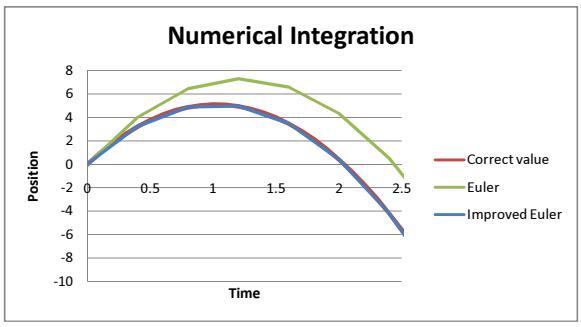


Figure 15: Numerical integration techniques

In terms of performance the time difference between these techniques is constant. This means choosing different techniques will only affect a particle system's computation by a constant factor for a given particle system size. It will affect CPU and GPU implementations in similar ways, and whilst accuracy is of course important, our primary concern is performance.

2.18 Memory Requirements

A simple particle system storing position and velocity only, using single floating point precision takes 2(position + velocity) × 3(dimensions) × 4(floating point bytes) = 24 bytes per particle. A common graphics card today has 512MB of memory. The maximum number of particles that can fit in graphics memory is then around 22 million.

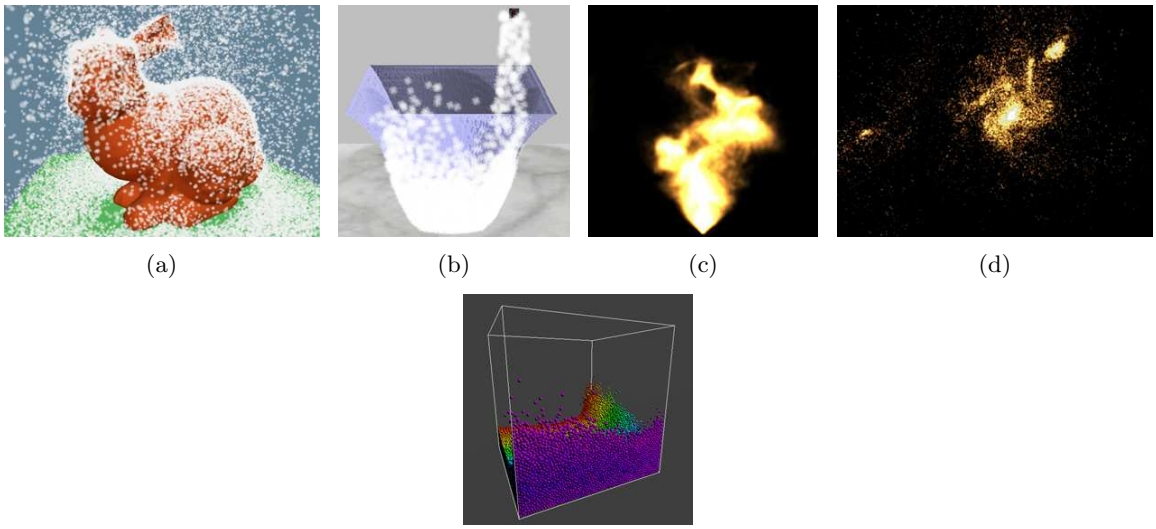
Many particle systems need to store additional information for example colour, size, temperature, time to live etc. Add to this the need for textures and geometry commonly found in a game and memory storage limits simulation of GPU based particle systems.

When a program running on the CPU runs out of physical memory the operating system uses virtual memory. Virtual memory uses the hard drive as extra memory and does so automatically. Currently graphics cards do not support virtual memory. In a GPGPU application if the data

cannot fit in video memory it must be stored elsewhere and loaded in on demand by the application. In effect, if a particle system requiring more video memory than available, the application must implement some form of virtual memory. If all particles require rendering at once, their data must be swapped in and out of video memory mid-frame, which can result in performance "cliffs".

3 Related Work

A number of papers have discussed performance of GPU based particle systems. Most of these have used a shader based approach which until recently was the only way to utilize the GPU for GPGPU applications. Each use different techniques with the aim of larger particle systems with increased performance.



(a) (b) (c) (d)

(e)

Figure 16: Other GPU based particle system implementations

UberFlow [KSW04] is a shader based particle engine that has inter-particle collision, particle and other geometry collision and also renders using transparency which requires sorting. This particle system was implemented entirely on the GPU so there was no bandwidth constriction between CPU and GPU. Uberflow found when adding collision between close particles, processing one million particles dropped from approximately 50 to 10 FPS.

Latta [Lat04] achieved one million particles running at close to real time speeds using shaders. Their particle system implements collision detection with a height field and also sorts the particles based on depth for alpha blending. Kolb, Latta and Rezk-Salama [KLRS04] extended this work and used depth maps for particles colliding with arbitrary geometry (Figure 16(a)). Kolb and Cuntz [KC05] implement fluid simulation (Figure 16(b)) on top the [KLRS04] particle system entirely on the GPU using shaders. The simulation achieves thousands of particles at real-time speeds.

Sebastian Sylvan [Syl07] implemented a particle system on Microsoft's XBox 360. It utilized the GPU and achieved 90 fps with just over two million particles (Figure 16(c)). The implementation included particle depth sorting.

Beeckler and Gross [BG08] implement particle systems on a number of field programmable gate arrays (FPGA). Using three FPGAs for computation, the particle system reach 112 million particles per second or 3.7 million particles at 30 fps. Although this is very fast, FPGAs are not widely available, and instead most computers have GPGPU compatible GPUs.

Nyland, Harris and Prins [NHP07] created an astrophysical simulation using CUDA (Figure 16(d)). This particle system groups clusters of particles together for computing gravitational pull to reduce the otherwise $O(n^2)$ computation to $O(n)$. Although this reduces accuracy it gives a huge performance increase because the mass of a cluster, not each individual particle can be used for particle movement computation.

Green [Gre07] from Nvidia wrote a particle system using CUDA that included inter-colliding particles (Figure 16(e)). This implementation uses a uniform grid spatial data structure that is constructed in parallel also using CUDA. It reached 65,536 particles at 120 fps.

The implementations for the last two mentioned particle systems are available. These have been run on our test platform and we compare results with our particle systems in Section 5.9.

3.1 Discussion

There has been a selection of work done in relation to GPU base particle systems. Many implementations utilize shaders, however as discussed earlier in Section 2.8 there is a lot of graphics specific overhead that is unnecessary when shaders are used for GPGPU. Two papers discussed particle systems implemented using CUDA, a new form of GPGPU or more specifically, GPU computing (as discussed in Section 2.9). While their end goals are similar to ours, these papers do not focus on examining and comparing performance.

This research follows on from previous work to produce in-depth analysis of the performance behavior and of different particle system types.

4 Implementation

In order to investigate our research questions some particle system implementations had to be constructed to measure performance. As discussed in Section 2.1 we categorize particle systems into three types: NIPS, SRIPS and LRIPS. For each particle system type we create a test case to compare performance gains, or losses, between CPU and GPU processing. Each case consists of a box filled with particles that bounce off the box sides. For each case we implemented a single and multi core CPU based particle system and a GPU based particle system. A further case introduces rendering geometry in addition to particle system computation. The different particle systems systems, as shown in Figure 17 are:

- Non interacting particles (NIPS). In this case a cube is filled with particles with randomized position and velocity. Particles bounce off the sides of the cube but have no inter-particle interaction (Figure 17(a)).

- Short range interacting particles (SRIPS). As in the first case a cube is filled with randomized particles that bounce off the cube sides. However particles are modeled as spheres and collide and bounce off each other as well (Figure 17(b)).

  A spatial data structure is required to limit collision detection to neighbouring particles. Since the particles in the cube stay evenly distributed a uniform grid spatial data structure is the best choice. The uniform grid implemented is constructed using radix sorting as described in Green's article [Gre07], and discussed in Section 3.

The use of the spatial data structure increases particle system performance but adds the construction time of the data structure to the frame time. Our CPU implementation of the uniform grid is not a parallel implementation and hence SRIPS has a serial component for the CPU tests. The timing specifically for uniform grid construction is analysed in the results.

- Long range interacting particles (LRIPS). This is implemented in the form of gravity (Figure 17(c)). A spatial data structure is of no benefit here as interaction is required to include all other particles. Hence computational complexity has a lower bound of $\Omega(n^2)$.
- Lastly a rendering load is created by rendering a terrain at the same time as performing particle system calculations (Figure 17(d)). This is an extension of the SRIPS implementation. Particles collide with both each other and may also collide with the terrain.

The timing results from one frame to the next may differ significantly as a result of caching and many other variables. The frame time generally settles after a short time so tests are run for 10 seconds each and timing results are then recorded. As discussed in Section 2.14 we take the average frame time. Results still varied slightly using this method. We found taking the median of the average frame time every 100 milliseconds to give much more consistent results with less jitter.

4.1 Test Environment

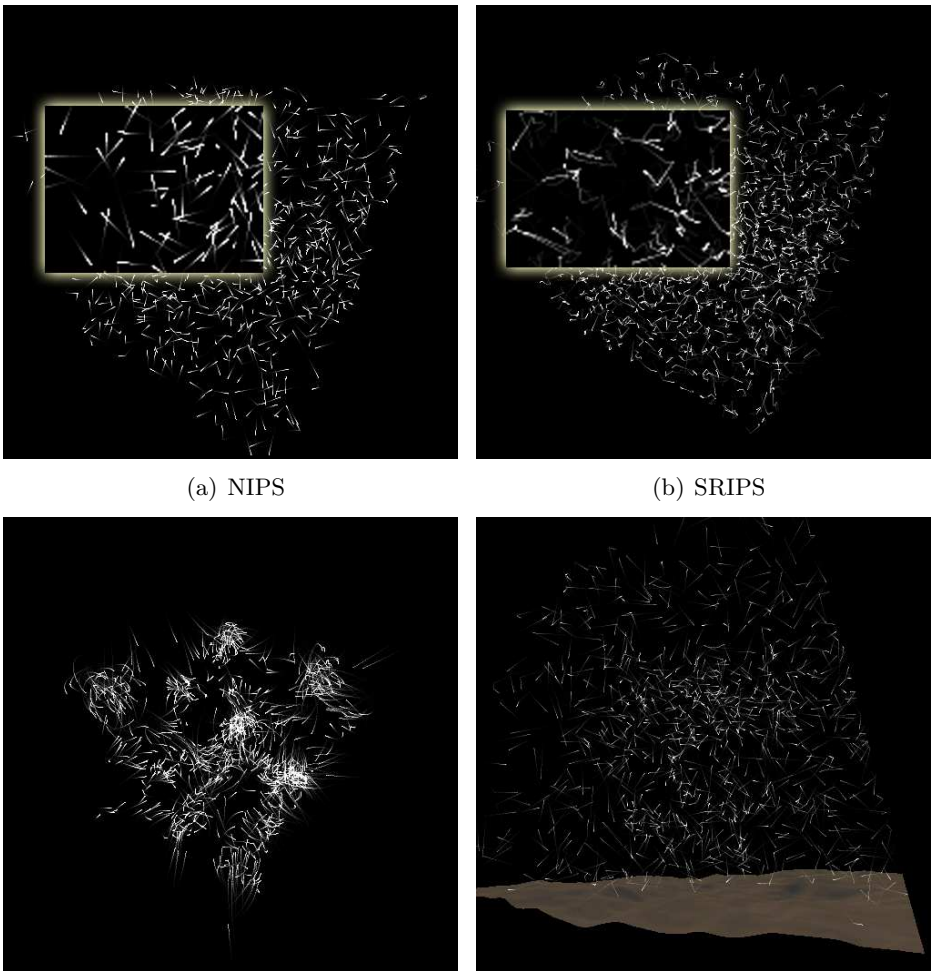
The specifications for the test platform are as follows:

- Intel Q9300, 2.5GHz quad core CPU (released March 2008)
- GPU1: Nvidia Geforce 9600GT (64 SPs, CUDA 1.1 capable, released February 2008)
- GPU2: Nvidia Geforce GTX275 (240 SPs, CUDA 1.3 capable, released April 2009)
- HP wx4600 Motherboard (Intel X38 Express chipset)
- MS Windows XP
- Implementations use CUDA 2.3, and OpenGL 3.1

For our tests we have endeavored to use hardware of a similar period. The CPU and GPUs we have used have been released within around a year of each other. There are now newer CPUs, the Intel i7s, which are of the same generation of the GT300 GPUs (Fermi), announced and to be released by year end.

5 Results

We measure the time per frame taken to compute particle systems. These results are given in milliseconds (ms). Commonly in games and graphics timing results would be given in terms of framerate measured in frames per second. Time per frame, the inverse of framerate, is more appropriate here. To relate these different scales, 30–60 frames per second is approximately 33.3–16.7 milliseconds per frame. All tests, unless stated otherwise, use the Nvidia GTX275 graphics card (which has a GT200 GPU) for particle system computation. Also computation is performed using single precision floating point numbers unless stated otherwise.



(a) NIPS (b) SRIPS

(c) LRIPS (d) SRIPS With Geometry

Figure 17: Different types of particle systems[6]

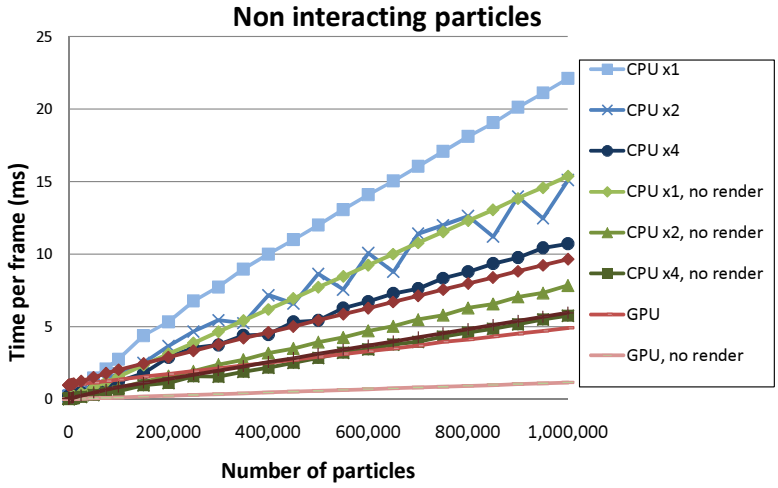
5.1 NIPS

Figure 18(a) shows the total time per frame versus particle system size up to 1 million particles. Some tests exclude rendering time for particles denoted as "no render". "x1-4" indicate the number of cores used in the CPU tests. "Copy back" indicates the data is being copied from graphics memory to system memory at the end of each calculation step. The tests which involve rendering using point sprites which is the simplest form of rendering required, consisting of just a 3D coordinate per particle.

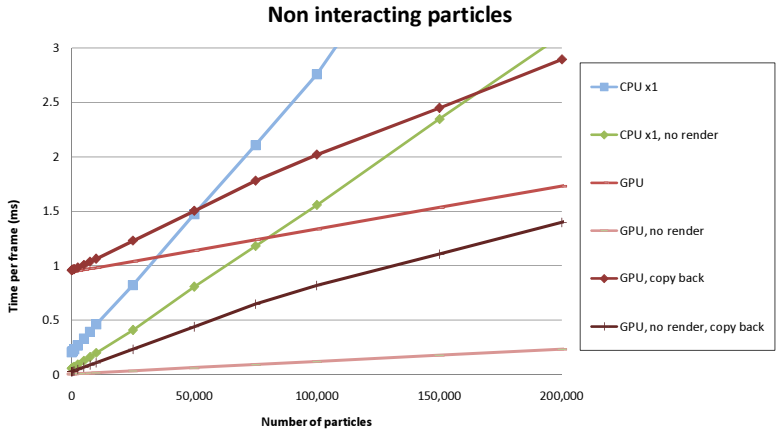
From the results in Figure 18(a) it is clear that the relationship for all CPU and GPU tests are broadly linear. That is, the time taken to process the particle system increases at a constant rate with increasing numbers of particles. This confirms the computational complexity is $O(n)$. This also shows that NIPS scale well on the highly multicore architecture of the GPU.

The GPU based NIPS processing time is 4.5 times faster than that of a single CPU core and 2.2 times faster than that of all four cores. These GPU based particle systems are faster

[6]Blurred to show motion



(a)



(b)

Figure 18: Non interacting particle system results.

by a factor of between 2–4.

Moving from the using 1 to 2 CPU cores decreases the processing time of 1 million particles to 51% without rendering, almost exactly a 2x speedup which is the best case possible. Using 4 cores decreases processing time to 73% that of dual core, or 37% that of single core (compared to the optimal 25%). These results suggest that NIPS do not continue scale well with increased CPU processors.

Another interesting finding is that when the particle data is required to be in system memory after calculation, the added time taken to copy back the data from the GPU increased the total frame to nearly double (196%) that of the original time. This shows that the transfer of a NIPS particle system position data takes almost as long as its computation and rendering. This performance loss brings the GPU based NIPS time close to that of the quad core based NIPS.

CPU based particle systems using 2 out of 4 cores and rendering has consistently given results where the time per frame jumps between values. Interestingly, the same pattern is not observed without rendering. This is seen again in Section 5.5, Figure 26. To speculate this could be a result of the scheduler only running two threads on a quad core CPU. It may also be a specific implementation detail. Further research should be done to find the cause.

Figure 18(b) displays a zoomed in section of Figure 18(a), showing in detail the results for up to 200,000 particles. It can be seen that for these small particles the CPU takes less time to compute non interacting particles than the GPU until the crossover occuring ar around 50,000–100,000 particles. CPU computation involves the overhead of transferring particle data to the graphics card to be rendered. However there is little difference in overall CPU time with and without rendering for small NIPS. This is not the case for GPU based NIPS. This would suggest there is overhead when rendering is applied as well as GPU based computation. It is possible there is a context switch between GPU processing modes causing this slowdown.

5.2 LRIPS

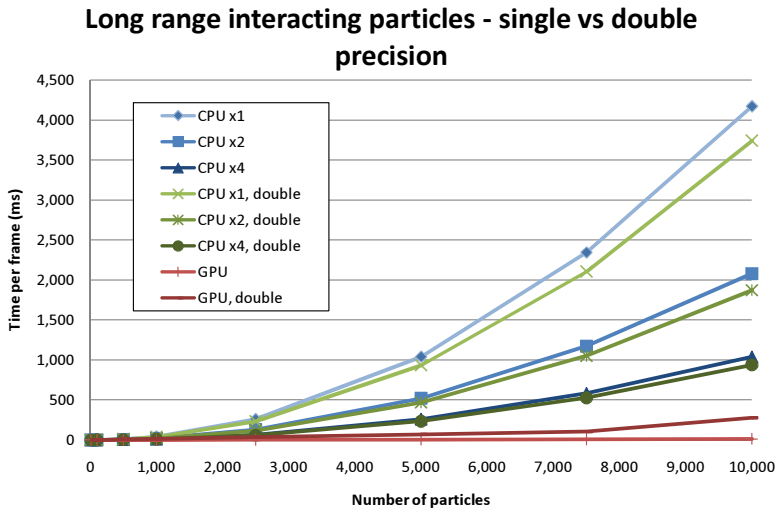


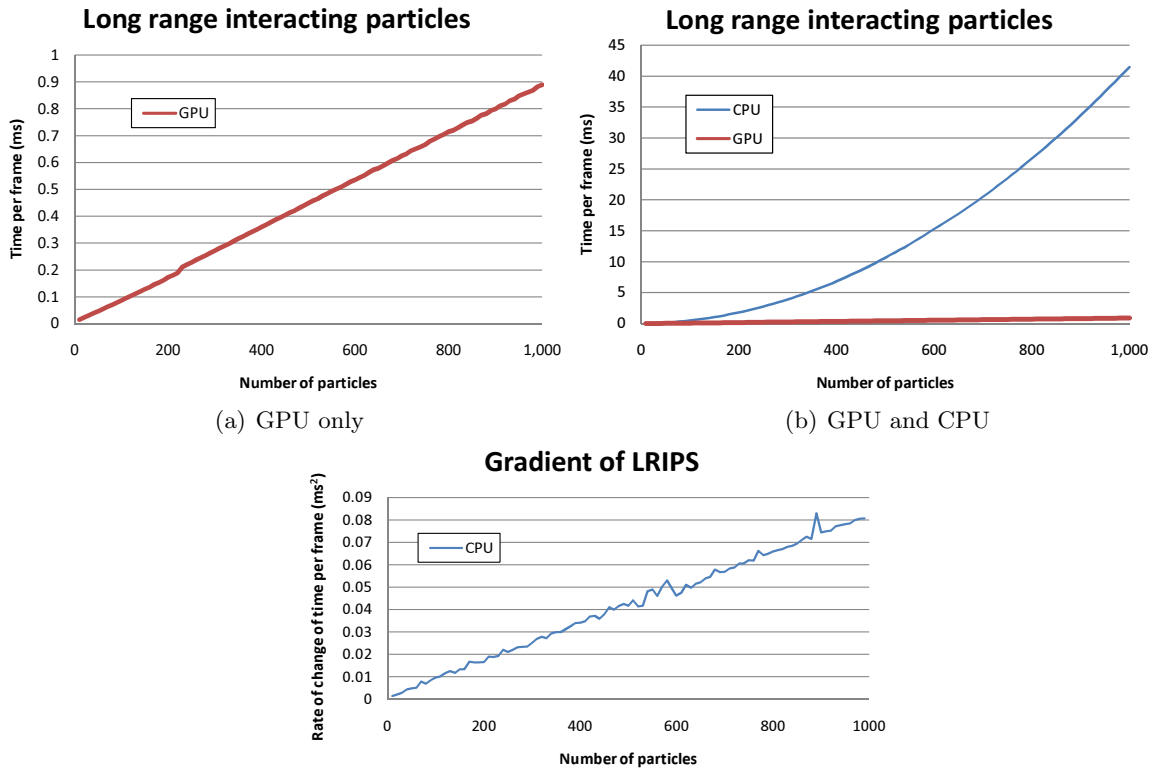
Figure 19: Long range interacting particle system results.

Figure 19 shows frame time for computing long range interacting particles. The results show GPU times to be far less than CPU times. The GPU is over 200 times faster than using one core, 158 times faster than two cores and 79 times faster than 4 cores. This clearly shows that GPUs compute LRIPS enormously faster than CPUs. Figure 19 also shows results for when double precision is used for particle computation, and is discussed in Section 5.5.

We expect LRIPS to have a quadratic shape given their computational complexity. To test this we take measurements at T_N and T_{2N} where T_N is the time per frame for N particles. If $T_{2N} = 4T_N$ for large values of N we conclude that the graph is quadratic. The results show all CPU tests are quadratic. However GPU results show $T_{1,000} = 1.98T_{500}$ and $T_{10,000} = 2.67T_{5,000}$ (without rendering). The first GPU result suggests a linear relationship while the second is not linear. However neither are close to quadratic. At this point we do not have a model which explains these results for GPUs.

Dual core CPU again reduces the time to half that of single. Quad core reduces time to half that of dual, or one quater of the single core. These results show LRIPS scale very well on multicore CPUs.

Figure 20 shows detailed LRIPS results for small systems with a high number of measurement points. Figure 20(a) displays GPU results only. It can be seen that the GPU computes small LRIPS with a linear relationship to the number of particles. Figure 20(b) adds CPU results. Figure 20(c) takes the derivative of the CPU results in Figure 20(b). While there is



(a) GPU only

(b) GPU and CPU

(c) CPU Gradient

Figure 20: Small long range interacting particle system results.

a small amount of variance in the derivatives it can be seen that gradient increases roughly linearly and hence the graph is quadratic, as expected.

5.3 SRIPS

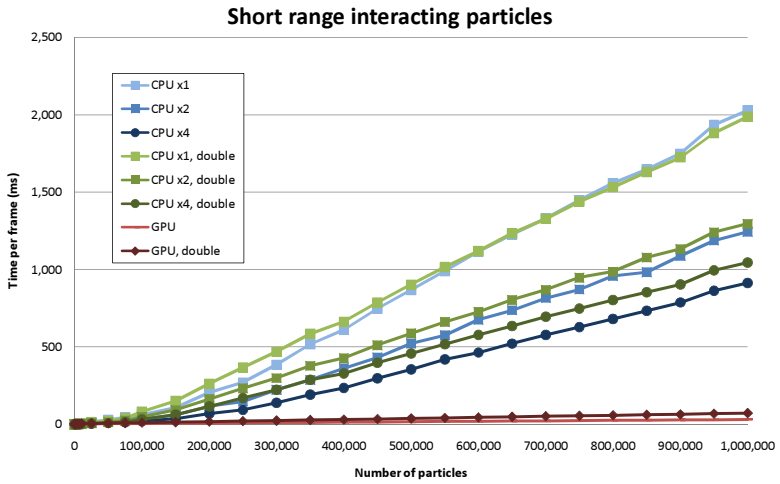


Figure 21: Short range interacting particle system results.

Figure 21 shows short range interacting particles. Similar to results for LRIPS the GPU is

much faster than all CPU results. The GPU is 74 times faster than using one CPU core, 46 times faster than two cores and 33 times faster than all four cores. The double precision results are discussed in Section 5.5.

Moving from 1 to 2 CPU cores reduces time to 62%, using all 4 CPU cores reduces time further to 72% that of 2 CPU cores which is 45% of the single core time. As stated in Section 4 the multicore versions of short range interacting particles do not parallelise the construction of the uniform grid. Following this Amdahl's law shows we cannot decrease particle system time beyond that of the time taken to construct the uniform grid. If the uniform grid construction time is factored out so we are measuring the time taken just to process the particles, the single to dual core results in 55% time and dual to quad a further 62% which is 34% that of the single core time. Even assuming zero overhead for uniform grid construction the results suggest multiple core CPUs do not scale well for SRIPS computation.

The relationship between number of particles and time per frame becomes broadly linear after the $200,000$ particles mark. As discussed in sections 2.1 and 2.16 the uniform grid spatial data structure reduces the comparisons between particles to a constant number assuming evenly distributed particles. This uniform grid implementation divides the cube up evenly so on average there are only 12 nearby particles to check.

The particle radius affects the speed of the system significantly as the bounce calculations are relatively expensive. To get consistent results the particle radius R was based on the system size N

$$R = \frac{1}{2} \cdot \frac{D}{\sqrt[3]{N}} \tag{2}$$

where D is the edge length of the bounding cube. This gives a reasonably frequent number of collisions but not so much that they become a dominating factor.

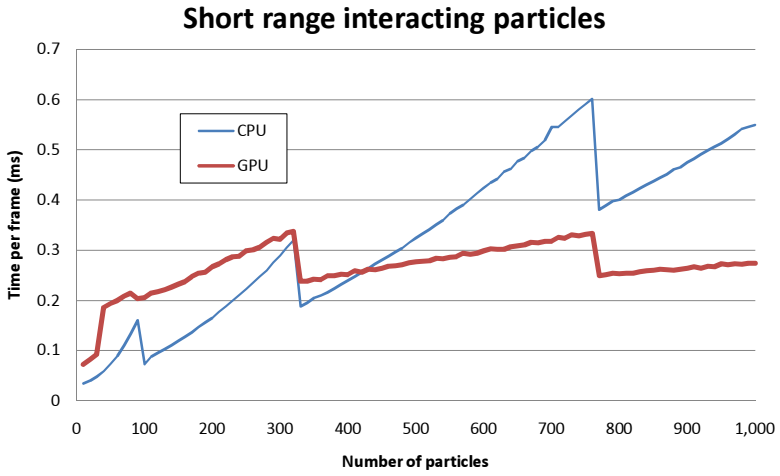


Figure 22: Small scale short range interacting particle system results.

Figure 22 shows a zoomed in section of our SRIPS results for small systems using single core CPU and GPU implementations. The time for SRIPS computation drops suddenly after every few hundred particles. This corresponds with changes in the uniform grid bucket count. The formula we used to calculate the grid size is as follows

$$B = \lceil \sqrt[3]{\frac{N}{12}} \rceil \tag{3}$$

where B is the number of buckets in each dimension of our grid, N is the number of particles and $\lceil\rceil$ is the ceiling function.

5.4 Particle Systems and Graphics

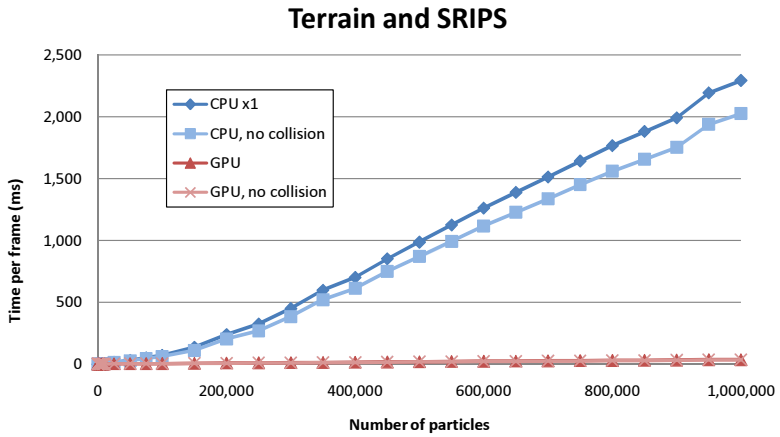


Figure 23: SRIPS with added rendering.

The addition of a rendering load changes the type of relationship between SRIPS processing time and particle count. Figure 23 displays SRIPS with and without collision between particles and the terrain. The results show SRIPS are computed faster on the GPU by a factor of 61 when particles collide with the terrain and 64 when they do not. Removing the collision between terrain and particles increases the speed of the GPU implementation by 1.05 and the speed of the CPU implementation by 1.13.

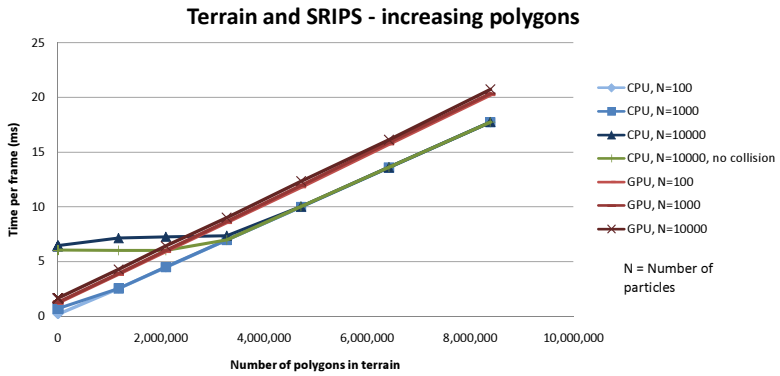


Figure 24: SRIPS with increasing geometry.

Figure 24 shows a number of tests where the particle system size remains constant and the tessellation of the terrain changes, increasing the number of polygons rendered. All GPU based particle systems run at similar speeds with differing rendering loads. The CPU is consistently faster for small particle systems of around 100–1,000 particles. For larger particle systems the CPU starts off being slower but as more geometry is added the CPU becomes faster than the GPU.

At the point where rendering takes the same amount of time as the CPU based particle system there is a sudden change in gradient for CPU based particle systems. At this point the

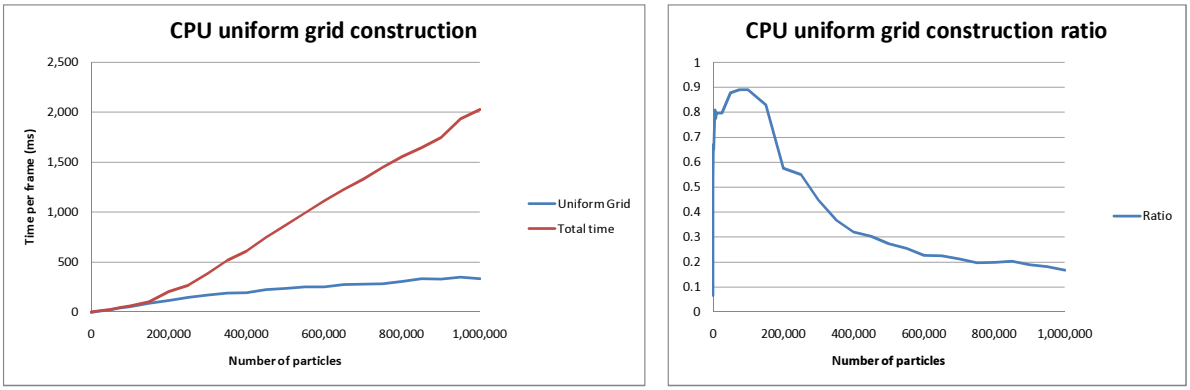
bottleneck transfers from being the CPU based particle system to the graphics rendering load.

In the case of GPU based particle systems the GPU is also used to render geometry. Offloading the processing to the CPU subtracts the GPU processing time of the particle system assuming the CPU can compute the particle system faster than the GPU can render the geometry. If the CPU can not compute the particle system at a faster rate then it becomes the bottleneck and we see a nearly straight line which represents the constant CPU computation for the particle system. When collision between particles and terrain is disabled the variance in this line is removed.

Uniform Grid Construction

The uniform grid is necessary to reduce the computational complexity for SRIPS from $O(n^2)$ to $O(n)$. It can be seen that the time for constructing a uniform grid does not compare with the speedup from its use, that is, it is well worth spending this extra time. This is shown by the results in Figures 19 and 21, where Figure 19 shows the $O(n^2)$ results for LRIPS and Figure 21 the $O(n)$ results for SRIPS using a uniform grid.

Figure 25 shows the uniform grid construction time for a single core CPU based SRIPS in relation to the total particle system time. Figure 25(b) shows the ratio of this construction time. For small SRIPS, below $200,000$ the uniform grid construction takes the majority of the total time. While the use of the spatial data structure greatly reduces processing time compared to the brute force method, the time could be further increased by optimizing and parallelising the spatial data structure construction. For larger SRIPS the uniform grid construction time drops to below 10%.



(a) (b)

Figure 25: Uniform grid construction.

For GPU based SRIPS, using the Nvidia CUDA profiler we found the uniform grid construction time to be 29% of the total GPU computation time (not including rendering time). While this is significant it is clearly not a bottleneck.

5.5 Single and Double Precision

Figure 26 shows NIPS computation with single and double floating point precision. Double precision results for LRIPS and SRIPS are given in Sections 5.2 and 5.3, Figures 19 and 21. In all cases for CPU computation there no more than a factor of 2 difference between single and double precision computation. The largest difference is the quad core computation of NIPS where

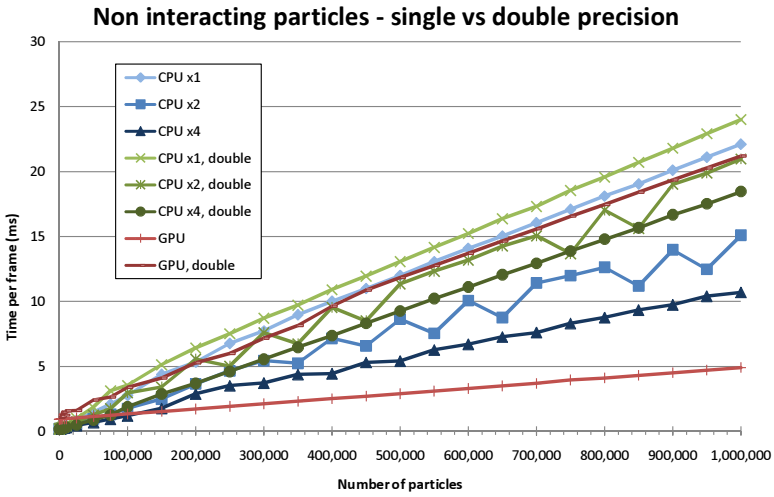


Figure 26: Single and Double precision for NIPS.

double precision takes 1.72 times longer than single precision. Interestingly double precision results are faster for CPU based LRIPS computation, although without further analysis we cannot speculate as to why.

GPU based double precision is quite different. GPU based double precision computation for SRIPS takes 2.3 times longer, NIPS takes 4.3 times longer and LRIPS takes 24.4 times longer than single precision computation. In general this shows that current generation GPUs are quite slow at performing double precision computation.

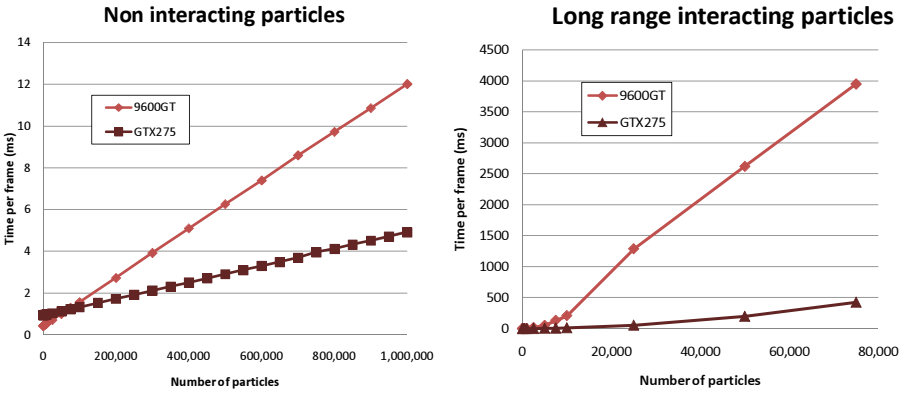
5.6 Comparing GPUs

The graphs in Figures 27(a) and 27(b) show particle system time for NIPS and LRIPS respectively, computed with different GPUs and graphics cards. Both graphs showing the newer GTX275 taking much less time to compute the particle systems. The GTX275 has exactly 2.2 times the memory bandwidth and 3.75 times the number of cores that the 9600GT has. There is also a difference in CUDA compute capability. For smaller particles the GT9600 performs better than the GTX275 but the GTX275 increases to 2.44 times the speed of the GT9600 at 1 million particles as shown in Figure 27(c). Given both performance of the 9600GT and GTX275 are linear relative to the number of particles the performance increase can be calculated as the ratio of the gradients of each graph. Using this method the GTX275 computes NIPS 2.92 times faster than the 9600GT. The time per frame for LRIPS computation of 1 million particles give an increase in speed of 9.26 for the GTX275. Thus the GTX275 performance is near 3 times better than the 9600GT as expected by the increase in number of cores.

5.7 Programming Complexity

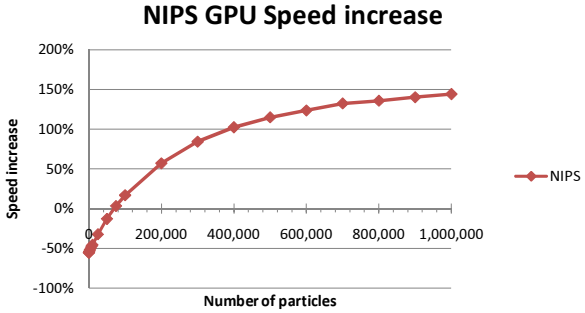
CUDA offers automatic thread creation and scheduling and also automatic scalability. This has a big advantage over multithreaded CPU applications as seen in previous results.

However to get those advantages an algorithm or application must be "embarrassingly" parallel before it will scale well with many-core GPUs. An example is the construction of a parallel uniform grid construction algorithm. The serial algorithm is fairly simple to implement compared to the parallel sorting method used in parallel uniform grid construction. Fortunately



(a) NIPS

(b) LRIPS



(c) NIPS ratio

Figure 27: Comparing Nvidia 9600GT and GTX275 GPUs.

a parallel algorithm existed in this case. There are some algorithms that can not be parallelised.

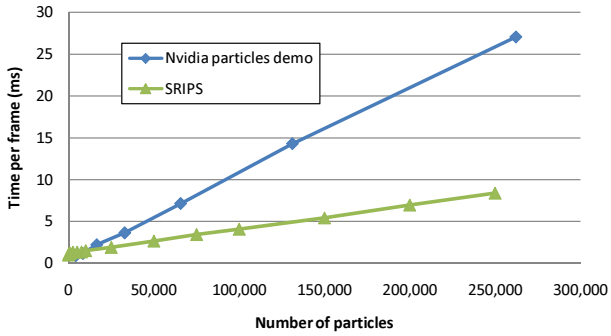
Development tools are still in the process of being built. One example is the CUDA debugging tool Nexus which Nvidia released this year during the course of this research. CUDA is a new technology and development tools are still in their early stages.

CUDA threads do not have access to dynamic memory allocation. Excess data must be allocated and memory management performed manually. Managing multiple memory address spaces is another programming concern which must be handled cleanly. Nvidia GPUs do not recover well from invalid memory access.

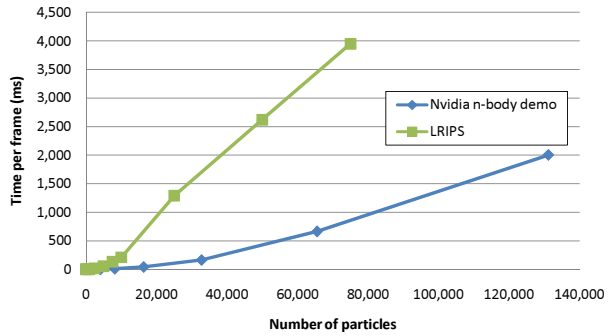
Communication between graphics and system memory is quite slow and applications seeking to take advantage of the GPUs computational power need to program around this, keeping data transfer to a minimum. It is also possible to use shared memory and GPU cache to increase data flow efficiency.

5.8 Comparing Implementations

Figure 28 displays previous particle system implementations by Nvidia, discussed in Section 3 compared against our own particle system implementations. As previously stated the GTX275 was used to run each of these tests. Figure 28(a) shows our SRIPS implementation runs faster than Green's particle system implementation (Figure 28(a)) which has the same type of interaction between particles. This is expected as Green's implementation has more interaction features. These results show our implementation is broadly similar to current ones and a good base for performance investigation. Figure 28(b) compares our LRIPS system against the n-body implementation by Nyland, Harris and Prins. These results show our system is expectedly



(a) SRIPS



(b) LRIPS

Figure 28: Results compared with Nvidia's particle systems.

slower because the n-body simulation uses a clustering technique that reduces it's computational complexity. However our LRIPS results are still broadly within the same range as the n-body simulation results.

5.9 Summary

Figure 29 summarises results from previous sections. To reiterate, the GTX275 performed faster than the Q9300 CPU in all particle system cases. For SRIPS, the GPU is 40 times faster than the quad core CPU and for LRIPS, the GPU is 79 times faster. These speed differences are extremely high. Even for NIPS the GPU can double the performance of the CPU.

Particle System	CPU x1	CPU x2	CPU x4	GPU
NIPS, 1M	22ms	14ms	11ms	5ms
SRIPS, 1M	2027ms	1244ms	913ms	31ms
LRIPS, 10K	4172ms	2079ms	1040ms	13ms

Figure 29: Particle system results in milliseconds per frame.

Double precision floating point computation does not change the time per frame for CPU based particle systems much. However double precision computation can increase the time for GPU based particle system performance from 2–24 times that of single precision.

The uniform grid construction time is significant but is not a bottleneck for SRIPS computation.

6 Conclusion

We found that GPU outperforms the CPU for large particle system computation, sometimes enormously so — the GPU can perform 79 times faster than the CPU in the case of long range interacting particles. Even the smallest GPU performance increase found in the case of non interacting particle systems was twice as fast as the CPU computation.

For small non interacting particle systems of the 100,000 particle range and below, the CPU is either faster or has similar speeds to GPU based non interacting particle systems. This may be of interest for applications with many small, different particle systems. This is a case left for future work.

Particle systems scale very well with GPU architecture. This is due to the inherent stream processing nature of GPUs matching the "embarrassingly parallel" characteristic of particle systems. Particle systems do not scale as well for multithreaded CPU applications. This is partly due to manual thread managing and complex scheduling.

Applications that require particle system data to be accessible in system memory will require the data to be copied back from GPU memory. This copy back time is costly and for large systems that are fast to compute, for example NIPS, may significantly decrease overall particle system speed. Another concern with this transfer time is for very large particle systems that take up more memory than is available. The GPU has no virtual memory so data must be manually swapped in. In order to implement large particle systems which do not fit in graphics memory it's computation and rendering would require multiple passes each frame including multiple context switches and data transfer.

In answer to the research questions in section 1:

1. We have shown huge performance improvements from CPU to GPU particle system computation. There seems to be a trend that higher computational complexity particle systems receive higher performance improvements from GPU computing.

2. Introducing a graphics rendering load to the particle systems has little effect compared to the processing time for large particle systems. There is a slight drop in performance due to a GPU context switch.

3. Designing embarrassingly parallel algorithms is the primary limiting factor for applications using GPU computing. Sometimes a parallel solution is not even possible. Algorithms must be molded around GPU design traits to achieve best results. There are a number of programming techniques that help to optimize algorithms, many of which are related to improving data flow using GPU cache.

In this work particle systems have performed well when computed on the GPU. Moreover, future generations of GPUs, for example Fermi, will bring dramatic increases in GPU computational power. This will open up a new realm of possibilities for particle systems. For example games may be able to use larger particle systems of higher computational complexities. This will increase visual realism and allow new gameplay.

7 Future work

GPU hardware is continually changing. Future work should include investigation of future GPUs. Preliminary results for Nvidia's GT300 (Fermi) GPUs show an approximate 400% increase in double precision computation. Although GPUs do not currently perform well for

double precision, Fermi may bring double precision performance closer to that of single precision performance.

OpenCL should allow particle system implementations to be switched between execution on different processor architecture, for example Nvidia and ATI GPUs as well as CPUs. OpenCL may also allow for load balanced particle system computation between CPU and GPU processors.

Particle systems too large to be computed all at once on a GPU may be of interest to scientific applications. Previously clusters of computers have been used to perform these experiments on CPUs [PSS97]. The use of GPUs and GPU computing in these clusters could potentially give a huge performance increase.

Small particle systems were were found to process faster on the CPU in some cases. Possible future work includes investigation of applications where many small particle systems are processed at once, for example games may have many particle systems of differing types such as explosions, cloth and water.

Particle systems where particles are not distributed evenly requires a spatial data structure that can adapt to this. It would be useful to investigate the performance of particle systems using different types of spatial data structures with differing particle arrangements.

References

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, New York, NY, USA, 1967. ACM.

[Ber98] Edmund Bertschinger. Simulations of structure formation in the universe. *Annual Review of Astronomy and Astrophysics*, 36:599–654, Sep 1998.

[BG08] John S. Beeckler and Warren J. Gross. Particle graphics on reconfigurable hardware. *ACM Trans. Reconfigurable Technol. Syst.*, 1(3):1–27, 2008.

[Bli77] James F. Blinn. Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, 11(2):192–198, 1977.

[Gre07] Simon Green. *CUDA Particles*. Nvidia, November 2007.

[Har08] Mark Harris. Cuda fluid simulation in nvidia physx. In *SIGGRAPH Asia 2008: Parallel Computing for Graphics: Beyond Programmable Shading*, December 2008.

[KC05] Andreas Kolb and Nicolas Cuntz. Dynamic particle coupling for gpu-based fluid simulation. *Proc. 18th Symposium on Simulation Technique*, pages 722–727, 2005.

[KLRS04] A. Kolb, L. Latta, and C. Rezk-Salama. Hardware-based simulation and collision detection for large particle systems. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 123–131, New York, NY, USA, 2004. ACM.

[KSW04] Peter Kipfer, Mark Segal, and Rüdiger Westermann. Uberflow: a gpu-based particle engine. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 115–122, New York, NY, USA, 2004. ACM.

[Lat04] Lutz Latta. Building a million particle system. In *Game Developers Conference 2004*, 2004.

[LHvdS01] Erik Lindahl, Berk Hess, and David van der Spoel. Gromacs 3.0: a package for molecular simulation and trajectory analysis. *Journal of Molecular Modeling*, 7(8):306–317, 2001.

[Moo65] Gordon Moore. *Moore's Law.* 1965. URL: `http://www.intel.com/technology/mooreslaw/`.

[MST04] Matthias Müller, Simon Schirm, and Matthias Teschner. Interactive blood simulation for virtual surgery based on smoothed particle hydrodynamics. *Technol. Health Care*, 12(1):25–31, 2004.

[NHP07] Lars Nyland, Mark Harris, and Jan Prins. Fast n-body simulation with cuda. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 31. Addison Wesley Professional, August 2007.

[PF05] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems).* Addison-Wesley Professional, 2005.

[Pho75] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, 1975.

[PSS97] David W. Pfitzner, John K. Salmon, and Thomas Sterling. Halo world: Tools for parallel cluster finding inastrophysical n-body simulations. *Data Min. Knowl. Discov.*, 1(4):419–438, 1997.

[Ree83] W. T. Reeves. Particle systems—a technique for modeling a class of fuzzy objects. *ACM Trans. Graph.*, 2(2):91–108, 1983.

[Ste81] David Stevenson. *A proposed standard for binary floating-point arithmetic: draft 8.0 of IEEE Task P754.* IEEE Computer Society Press, 1981.

[Syl07] Sebastian Sylvan. *Particle System Simulation and Rendering on the Xbox 360 GPU*, 2007.

[TZ99] M. S. Tillack and J. D. Zhang. Particle dynamic simulation of free surface granular flows. June 1999.

[VMT04] Pascal Volino and Nadia Magnenat-Thalmann. Animating complex hairstyles in real-time. In *VRST '04: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 41–48, New York, NY, USA, 2004. ACM.