Testing a Complex Data-structure on General Purpose Graphics Processing Units.

Daniel Persson

Department of Computer Science General Purpose Computing
Blekinge Institute of Technology

Thesis submitted for the Degree of Candidate of Computer Science at the
Blekinge Institute of Technology

· 2007 ·

Abstract

English
This thesis is about general purpose computing on the graphics processor. The reason why this is important is because of the performance advantages that can be achieved in ordinary applications by using the GPUs programmability and performance. The problem investigated is the use of a complex data-structure, namely linked lists, and what their possible benefits are when run on the GPU. I also wanted to investigate if it was viable to implement a complex data-structure on a GPU. Implementations was made of the linked list both on the GPU and on the CPU and then measurements of the performance of doing different linked list operations was conducted. Also tests was made to measure the quality of the output. I was surprised to see that the GPU performed bad when compared to the CPU on all of the linked-list operations but the quality testing showed that the GPUs and CPUs output were the same. Testing of what parts of the GPU application that caused the bad performance showed that it was the initiation of the application. I also found out that it was not that hard to learn how to program applications for the GPU except for when learning the new programming model. To conclude it can be said that my first investigation showed that linked lists does not run faster on the GPU than on the CPU but the quality is sufficient. My second investigation about the viability to use data-structures on the GPU showed that it was much easier than I expected and therefore viable if you can tolerate the bad performance.

Svenska
Den här avhandlingen handlar om generella beräkningar på grafikkortet. Anledningen till varför detta är viktigt är för att hastighetsfördelarna som kan uppnås i vanliga applikationer genom att använda GPU'ns programmerbarhet och hastighet är stora. Problemet som undersöktes var användningen av en komplex datastruktur, nämligen en länkad lista, och vad dess fördelar är när man jämför med en CPU. Jag ville också undersöka om det är värt att implementera en komplex datastruktur på en GPU. Implementationer gjordes av den länkade listan på både en GPU och en CPU och sen mättes prestandan när olika operationer utfördes på listan och kvaliten mättes på utdatan. Jag var överraskad av att se att GPUn presterade dåligt jämfört

med CPUn på alla listoperationerna men kvaliten visade att mina utdata var samma på GPUn och CPUn. Test gjordes om vilken del av GPU applikationen som gjorde att prestandan blev så dålig och jag kom fram till att det var initiering av applikationen. Jag kom också fram till att det inte var så svårt att lära sig hur man programmerar applikationer för en GPU förutom när man skulle lära sig den nya programmerings modellen. Som slutsats så kan man säga att min första undersökning visade att den länkade listan inte är snabbare på GPUn jämfört med CPUn men kvaliten är tillräcklig. Min andra undersökning om det var lönt att använda datastrukturer på GPUn visade att det var mycket lättare än vad jag trodde och att de därför är lönt om man kan tolerera den dåliga prestandan.

Contents

Chapter 1

Introduction

With the recent years advancement in computer graphics the graphics processor has not only been getting faster but it has also become very programmable. If it is possible to fully utilize the power of the GPU (Graphics Processing Unit) in areas where the CPU (Central Processing Unit) has had its stronghold before, you can get innovation and differentiation in general applications. When not using your GPU, i.e. when you are not playing a game, there is very much computing power that could be utilized in different areas of your computer system.

1.1 Task

The task of this thesis is to conform to the recent development and examine data-structures when applied to GPUs. The data-structures which is a "complex" variation, e.g. linked list or a binary tree, is implemented on a GPU and on a CPU in different programming languages. To measure the execution and to make sure that the quality is sufficient a test-system with a GPU conforming to recent development in programmability is used.

When implementing the data-structures I measured the viability of the implementation in a real-world project. E.g. if it is hard to implement and if it takes a long time to do so. This is only an approximation of todays tools and my experience. In the future this will probably be made easier by better frameworks and further programmability on the GPU.

1.2 Goal and Purpose

My first hypothesis is:

* The comparison of data-structures on the GPU and on the CPU will show that the GPU has better performance.

When completed the comparison test, of data-structures on different architectures, I was able to conclude if my first hypothesis was right or wrong, whether the GPU or the CPU faster and generally better on computing data-structures. I was also able to determine the limitations and strengths of each architecture. The performance measured is the speed of the different implementations.

My second hypothesis is:

* Data-structures on the GPU are hard to implement because of the stream programming model.

When doing the implementations I assessed if it was viable to use data-structures on a GPU. I found out what my problems were and where I think simplifications can be made. It is a loose hypothesis that only I will answer and ponder upon.

1.3 Motivation

The motivation is that the GPU should be able to work as a general purpose computing unit and not only do graphical computations. The GPU has enormous powers compared to the CPU and not to use them at all times is unnecessary. [2] The GPU could be as transparent as the CPU is i.e. the kernel of the operating system should be responsible for sending work to the GPU. This thesis is to give more insight on the possibilities and the difficulties to reach general purpose computing on the GPU.

1.4 Method and Materials

Firstly a literature study concerning GPGPU (General-Purpose Computation Using Graphics Hardware) was conducted, to find out what other people has accomplished in the area. I read about the stream programming model and general backgrounds information concerning the programmability of the GPU to increase my awareness.

I implemented a linked list to do my testing because of its simplicity but still being a very general data-structure. [10] I implemented one version in C because of its speed, one in Java because of its high-level programming and one in Brook [6] because of it being a language to program on the GPU. The linked list is a linear one-way linked list and I made the different implementations a similar as possible. The C, Java and Brook implementations is very much alike with the same functions with the same parameters, the same amount of iterations, generating the same output and the same amount of function calls. In the Java implementation there was some differences though because of the way the language is built. I had to use implicit pointers in Java but in C and Brook explicit pointers could be used.

The java implementation is only included in the report as a reference comparison between the different CPU implementations. It was not necessary to include it to be able to conclude my hypothesises but it is interesting to see the major differences in CPU languages.

Because of recent development in the version controlled repository of Brook a snapshot from 2007-04-20 was used with various patches applied to get the latest version with the most features. The Brook system provides four backends to run the Brook code on and I was able to test two of them on my GNU/Linux system, OpenGL and CPU. [6]. I wanted to try different versions of Brook but there was problems when I wanted to compile some versions for Linux. There was incompatible parts between the nVidia drivers and the Brook code.

The features implemented and tested in my linked list is insertion and deletion where I can; insert and delete at the first, the last and at a specified position. I also implemented a search-function and a list-function. The search-function is to search for an element in the list and return true if it finds it. The list-function is a way to print the entire list to the console. The reason why I choose to implement insertion and deletion at the first, the last and at a specified position is because the different operations take different amount of computing to be performed. Inserting and deleting at the first position the application only have to traverse one position in the list and then perform its operation. Doing the same operation but at the last position requires the application to traverse the entire list until it reaches the end and doing the operations at a random position is to receive a general result from ordinary linked list usage. The search-function was implemented because it being a good way to traverse the whole list without inserting or deleting a node and the print-function was implemented to control the linked list when executing the other operations.

To test the performance I used the UNIX ”time” application which is a time-measuring facility invoked by running `"time <binary>"`. Time then returns three results; real, user and sys. ”real” is the actual time from invocation to the termination of the application i.e. the time from start to finish. ”user” is the amount of CPU time that the application spends in user-space and ”sys” is the amount of CPU time that the application spends in kernel-space. The most important of these are the real time because of it being the actual time the application spends computing.

I also tested the quality of the output with the concern that the different floating-point standards would make a significant difference between the implementations. I did this test by using floating points in all my implementations and printed my calculated values after an operation. Then I could learn what standards my GPU was using.

To see that my implementations on the GPU were not flawed in some way I also did a test were it was known that the GPU would be superior in comparison to the CPU. To confirm this I made a kernel in Brook with very heavy floating point calculations.

When retrieving the results two iterations of each test was made and if they were practically the same the first results was used. The reason for doing this is to exclude any other parts of my operating system using an significant part of the computers resources which could results in invalid results.

Here follows the test-system that I used. The first list is the most influential parameters concerning the GPU because it is the actual software dealing with graphical output on a GNU/Linux operating system.

- Nvidia Geforce 7950GT(G71) 512MB 1.4ns DDR3 PCI-Express

- nvidia-drivers - 1.0.9755
- nvidia-drivers - 1.0.8766
- Xorg [17] - 7.3

The first item is the graphics card I used to test the linked list on. To make sure that there was not any bug in the drivers I tested with different version of the graphics drivers to see that the results did not differ. And finally I tested with Xorg, the currently stable Xorg version.

The next list is what hardware I otherwise have in my test-system. The items in the list is influential in both the GPU- and CPU-tests because they are the underlying main parts of a computer system were all applications are initiated from.

- Intel Core 2 Duo E6600 @ 2.4GHz
- Corsair DDR2 2048MB

The first item is the CPU and it is a dual-core CPU but for the sake of homogeneousity I only used on core when I ran my tests. The second item is the amount of memory I have in my system.

The third and final list is the software in the base-system that the test-system was using.

- Linux - 2.6.20
- Linux - 2.6.21-rc5
- GCC [18] - 4.1.2
- glibc [19] - 2.5 (NPTL)
- GNU time [20] - 1.7
- Bash [21] - 3.1

I tested with two different kernels to try to exclude any bug in the kernels. It is possible that the bug is in both kernel but the differentiation delimits the possiblity a little bit. I used the currently stable versions of GCC and glibc where GCC is the compiler of the system and glibc is the standard C library used in GNU/Linux. To measure the time an operation took I used two different software. The first is the one provided by the GNU foundation and is called GNU Time. To verify my results I also tested with the built-in implementation available in Bash with the same functionality but with another implementation.

The rest of this thesis is structured as follows. I will start with background-information concerning CPU, GPU and the difference between them. I will also try to inform on how the programming model used on GPUs work and the framework to do the actual programming on work. Following that is the results of my testing and then a discussion about the results. In the end some conclusions are stated and also some guidance to what future work should focus on.

Chapter 2

Background

Processors have since 1965 doubled its number of transistors every year as Gordon E. Moore predicted with Moore's law [1]. In 1965 the CPUs had about fifty transistors per die and today they have hundreds of millions of transistors. Not only is the number of transistors increasing per die but the size of the transistors is decreasing by 15% per year which directly reflects the core clock speed. Combining the increase in transistors and decrease in transistor size results in 21% performance increase.

As chip size increases the amount of time, measured in clock cycles, it takes for a signal to travel from one end of the chip to the other end also increases. This results in that multiple cycles is required for a signal to travel across an entire chip. This is an increase of time for the communication that also increases every processor generation. In the future chip designers will use simple multiple cores instead of a very advanced and expensive single chip design. Another impact of this is that there will be an increase in the amount of computation available per word of memory bandwidth. In this area the GPUs have been very successful. [2]

Latency and power-requirements is also a concern for future processors. Latency decreases in a not enough rate to keep up with the bandwidth. Therefore the processors have to be able to tolerate larger amounts of latency by doing more work by themselves while waiting for data to return from other operations. The processors also requires more power every generation and this will probably be a very limiting factor in the future and that is why a more parallelized approach is required. [2]

All of these shortcomings have to be solved when you want to achieve efficient computation and efficient communication and the CPU is not a good choice to solve these problems. [2]

A transistor can be roughly categorized into three sections, control, data path and storage. The control section is a number of transistors that the hardware use to direct computation to different sections on the processor. The data path is the actual hardware that performs the computations and the storage section is the hardware that stores data. [2]

To achieve efficient computation the processors have to maximize the hardware devoted to the data path and allow multiple computations operate at the same time. With graphics processing the typical tasks is to compute sequentially which makes it possible to run several of these tasks simultaneously. The arithmetic units are not fully programmable yet but we can achieve better efficiency by taking advantage of specialization [2]. For example when a unit performs one kind of computation it can gain considerably more efficiency.

To achieve efficient communication you have to reduce the amount off off-chip communication and the best way to do that is to keep as much required data communication on-chip as possible. The only communication should be truly global data. Another way is to use caching. By using caching you have a recent copy of data in memory to remove the need to fetch off-chip data that is needed. Another very powerful way is to use compression when transferring data between chips but the compression trades transistors and computation for off-chip bandwidth. [2]

Traditional microprocessors target general purpose computing with goals unlike the graphics microprocessors. The CPU has less parallelism, advanced control requirements and performance goals that is not adequate enough to perform advanced graphics operations and many other similar applications. The programming model of the CPU is in general not parallel and the CPU hardware reflects this when it only computes one piece of data at the time. There have been some recent improvements in this area with instruction sets like SSE (Streaming SIMD Extensions) and AltiVec but the degree of parallelism is far less then on the GPU. The reason why the CPU is less efficient then the GPU is because more transistors have been dedicated to control requirements on the CPU and only a fraction of the CPUs die is spent on actual computation. The CPU also has no specialized hardware for particular functions that the GPU have. CPUs is optimized for low memory latency [3] whereas the GPU is designed for maximum throughput. Because of the CPUs inferior parallelism it needs to return memory references as fast as possible (latency). The GPU tries to achieve maximum throughput for all elements rather than focusing on latency which in the end results in better utilization of the memory and higher performance overall. [2]

2.1 GPU

A GPU [23] is a graphics processing unit that is a second processor in your computer that is dedicated to graphic calculations. The reason why you need an extra processor is because games, photos and videos require much more processing power that the CPU alone cannot handle. The performance of the GPU has increased alot in the last five years. The rendering rate [24] has doubled every six months over five years. Also five years ago our graphics output was not made by GPUs but by graphics accelerators. The accelerators didn't do anything other than to accelerate what the CPU gave to the GPU. With GPUs you try to do the actual graphical algorithms on the graphics processor and then output it. With the change from graphics accelerators to GPUs the pipeline was broken down into steps that could be programmable and parallelized. The demand of GPUs has risen because of the rise of video games and that has made the GPUs very cheap. The low price and the high performance of the GPU have driven the market where you can buy a graphics processor that is capable of almost one teraflop of floating-point operations per second for a couple of hundred dollars. [2]

As mentioned before the GPU has become more and more programmable over time which has led to people wanting to convert their programs to utilize the power of the GPU. The different stages [25], fig: [2.1] that is programmable is the vertex stage, fragment stage and on new graphics processors also the geometry stage.

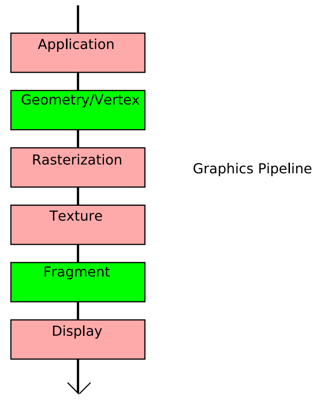


Figure 2.1: Programmable Graphics Pipeline. Green stages programmable.

The vertex processor is the stage where the GPU applies the vertex shader and modern GPUs have multiples of these processors. The vertex shader processes streams of elements contained in vertex elements. A vertex is any point in a polygon and a polygon is a shape in computer graphics. A vertex consists of positions, colours, normals, vectors and possibly other attributes. The vertex shader processes this information in chunks (streams [4]) on each vertex to make the positions relative to the camera and then a set of three vertices to compute a triangle. Triangles and nurbs is the building blocks in computer graphics but triangles is the ones that are used in realtime graphics. Then a stream of fragments is generated. In terms of programmability vertex shaders is not used as much as the fragment processors because of limitations on older graphics cards that is not using Shader Model 3.0. Earlier versions has the inability to read information from input elements other than the current stream that it is operating on. Also there is more fragment processors than vertex processors on a graphic processor which makes the fragment processor more favourable. [25]

The fragment stage operates on streams of fragments that contains the information needed to generate shadows and lightning on the final image. The fragment processor applies a fragment shader, also called pixel shader, to each fragment to compute the final colour for each pixel. The fragment processor also has the ability to fetch data from textures which makes it more versatile than the vertex processor. Another reason why the fragment processor is more favourable is because the data generated is being fed directly into the memory where the vertex processors data goes through the rasterization stage and the fragment stage before reaching memory. [25]

2.2 Stream-processing

As described earlier the CPU is designed after their serial programming model which does not expose parallelism and communication. The GPU on the other hand uses the stream programming model [4] which is designed specifically to accomplish this higher efficiency in computation and communication. When programming with the Stream model everything is represented as a stream. A stream is a set of data of the same type for example integer, floating points, triangles or matrices. With greater length of the stream comes greater efficiency and it is possible to perform a multitude of operations on them. One of the operations is called a kernel(not to be confused by the kernel of an operating system). A kernel is an operation that operates on a complete stream with one or more input streams and one or more output streams as a result. [4] A kernel could for example be used to:

- Evaluate a function on each element of an input stream.(Transformations)
- Outputting multiple elements for every single input element.(Expansions)
- Outputting one element from multiple input elements.(Reductions)
- Where a subset of input elements becomes the output.(Filters)

Kernel outputs are functions of their kernel input and the internal kernel computations on one stream element never depends on computations on another element. This has the advantage that it is easy to do parallel computations and that the data required for the kernel is known. The stream model is then completed when you group several kernels together to complete a task. That is why the stream programming model is good for the graphical pipeline which consists of stages connected by data flows. [2]

The stream model makes the computation demands of future processors very feasible. The biggest difference is its inclusion of parallelism in the foundation of the model. Because the kernels operate on entire streams, stream elements can be processed in parallel using data-parallel hardware. Long streams with lots of elements allow the parallelism to be highly efficient. Because that tasks is constructed from multiple kernels they can be deeply pipelined. Also hardware implementations for special kernel operation can be made very efficient compared to programmable hardware. [2]

In regard to efficient communication the stream programming model also achieves great performance. When transferring whole streams instead of separate elements large initiating costs is avoided. Also the structuring of the kernel as a pipeline results in a kernel that can stay on chip and be transferred directly to next stage in the pipeline. Therefore eliminating the need to go over into the memory when it is waiting to be processed [2]. And finally the deep pipeline also allows for continuation of the execution while waiting for data to return from global memories. This allows for a high tolerance in regards for latency and allows very high throughput. The stream programming model has been gradually introduced into the graphics pipeline [5] but in 2000 the first GPU was released that allowed the user to write individual kernels in the graphics pipeline. In current GPUs the programmable stream processors is called the vertex program, fragment program and the more recently introduced geometry shader.

2.3 Architectural comparison

Because the purpose of the GPU and CPU is different there is some terminology differences [25] between the two. For example when you have an array on a CPU you have a texture on the GPU. When you program for the fragment processor the behaviour of the data is almost exactly the same as the CPU and that is one of the reasons why fragment processors tend to be more useful when doing general purpose computing on the GPU.

To represent inner loops on the GPU you have to think about the stream programming model and the parallelization on the GPU. Because of the GPUs multiple processors you do a non-sequential and parallel loop over a stream when inside a kernel. The instructions inside a kernel are applied to all elements of a stream and on a CPU the instructions inside the loop are the kernel. [25]

Because of the CPUs unified memory model it is easy to read and write to memory anywhere in the program but in GPUs it is not so easy. On the GPU you have the constraints of the pipeline to consider. A stream must be entirely completed by a kernel before it can continue to its next stage so when you want to have feedback from your fragment program you have to render to a texture when it is completed and use that as input to future programs. [25]

To invoke computation on a GPU it is needed to generate streams of fragments and that is typically done on GPUs by processing every element of a rectangular stream representing a grid. Computation invokation on GPUs is basically just drawing geometry. [25]

Another architectural difference is that GPUs normally support several floating point standards [27] while CPUs generally only supports one standard. A floating point number is represented by `sign x 1.mantissa x 2^(exponent - bias)` . That means that with a 23 bit mantissa we can only represent the number 143.98375329 to 143.98375. When using NVIDIA fp32 you get the same result as to when using the CPUs representation. The most commonly used is:

- NVIDIA fp32: 23-bit mantissa, 8-bit exponent
- NVIDIA fp16: 10-bit mantissa, 5-bit exponent
- ATI fp24: 16-bit mantissa, 7-bit exponent
- CPU [26] fp32: 23-bit mantissa, 8-bit exponent

Here is a table of the different floating point standards and their respective representation of the floating point number 143.98375329:

Format	*Result*	*Error*
NVIDIA fp32	143.98375	0.00000329 (2 x 10 $\hat{(}$-6)%
NVIDIA fp16	143.875	0.10875329 (7 x 10 $\hat{(}$-2)%
ATI fp24	143.98242	0.00133329 (9 x 19 $\hat{(}$-4)%
CPU fp32	143.98375	0.00000329 (2 x 10 $\hat{(}$-6)%

2.4 Data-structures

The definition of a data structure is [7] ”An organization of information, usually in memory, for better algorithm efficiency, such as queue, stack, linked list, heap, dictionary, and tree, or conceptual unity, such as the name and address of a person, It may include redundant information, such as length of the list or number of nodes in a subtree”

The data structure that I am going to use is a linear linked list. A linked list [8] is a list where each item has a link to the next item in the list. The advantage of linked lists over other data-representations is because we do not have to define the maximum number of elements to insert into the list when we create it. The linked list resolves this problem by being very flexible and dynamic. Elements can be inserted or deleted from the list whenever. A linked list allocates space for a new item every time it is inserted into the list and also populates a next item which is a pointer to the next element in the list. A basic linked list looks like fig: [2.2].

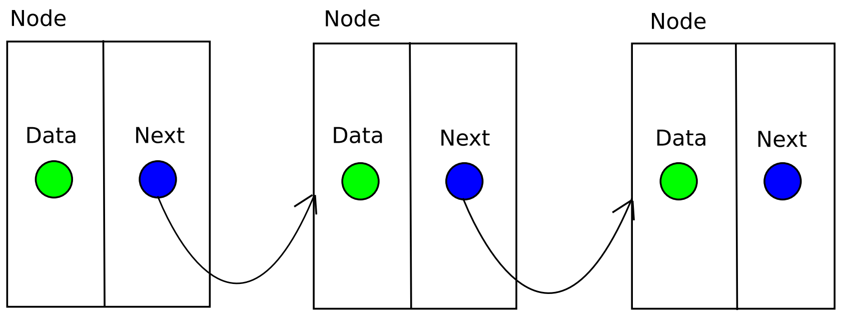


Figure 2.2: Basic linked list. [9]

As explained in the figure each node has two elements. One element that contains the actual data and one pointer to the next item in the list. The last nodes next pointer points to NULL to indicate that it is the end of the list. [10]

To handle the entire list you simply have a pointer to the first node in the list and it is usually called the head of the list. If you for example want to add an item to the first position in the list you simply: allocate space for the node, populate the data into the new node, point the new nodes next-pointer to the current head of the list and finally point ”head” to the new node. [10]

There are variations of linked list implementations each differing in a small way. Circularly linked lists is when the end of the list, often called tail, points to the head of the list and thereby finding the beginning of the list at tails next-pointer. There is also a linked list type which is double linked. Which means that each node not only contains a next-pointer but also a previous pointer. This kind of list is useful for example when you want to be able to search in a list forwards and backwards. If you want to go backwards in a simple linked list you had to go forwards through the entire list. By using double linked list you therefore save lots of time. [10]

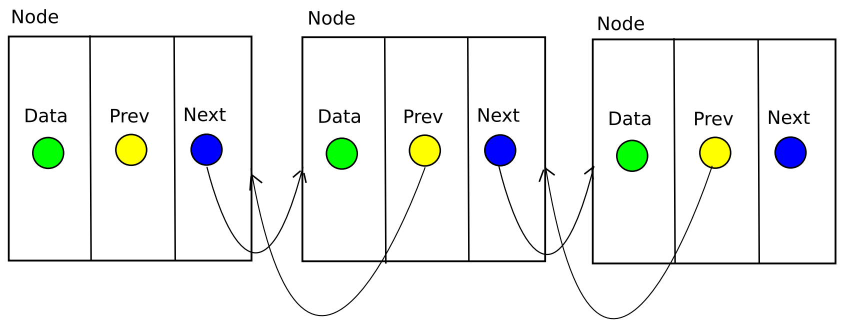


Figure 2.3: Double linked list. [9]

With linked lists you can thereafter construct more complex data-structures like queue or a stack. [10]

There is some differences however in the implementation depending on wich language you are using. In C/C++ there is explicit pointers defined but in Java there isn't. In java you have to use implicit pointers where you take advantage of the object-oriented approach that java has. The structure of nodes in java could look like this: [11]

```
public class ListNode
{
        float val;
        ListNode link;
}
```

Where ListNode link is the next implicit pointer to the next node in the list. In C/C++ the structure of a nodes could look like this:

```
struct list_element
{
        float val;
        struct list *next;
};

typedef struct list_element item;
```

In this example you use explicit pointers for the next variable to point to the next node in the list.

2.5 Brook

Brook [6] is a framework for general purpose computing on the GPU. It consists of a compiler and a runtime implementation of their own language called the Brook stream programming language. Its main purpose is to demonstrate general purpose computing on the GPU, provide a useful and simple tool for developers who want to run applications on GPUs and provide researchers a stream programming model to test the GPUs programmability and performance. The Brook stream programming language is similar to standard ANSI C [12] and it is designed specifically to incorporate parallel computing and intensity of arithmetics to conform to the GPUs specialities.

The framework consists of the BrookGPU compiler and the **B**rook **R**un**T**ime Library. The compiler translates the Brook stream programming language files (*.br) to C++ syntax (*.cpp). Its input is a source file and its output is also a source file therefore it is a metacompiler. The Brook Runtime Library is an architecture independent layer which implements the backend for the primitives defined in the Brook language, presents a generic interface for the compiler. The backends is chosen at runtime or can be specified by the user and DirectX9 [13], OpenGL ARB [14], NVIDIA NV3X and C++ is the possible backends. There is also work being done concerning ATI's CTM [15] technology. [6] i The OpenGL backend is supported under both Windows and Linux. When using a NVIDIA specific board profiles called fp30 or fp40 will be used to the assembly specific parts. They are the profiles that state what the GPU is capable of. When using ATI only fp30 will be used and therefore NVIDIA is currently the best supported board under Linux. When using Windows the DirectX backend is also supported and both NVIDIA and ATI work and if they support Pixel Shader 3.0 [16] they will also support looping in the kernels. [6] In OpenGL the Shader Model 3.0 will be used but under linux there are some limitations due to Brook using pbuffers instead of FBO. Pbuffers is a technique to allow you to do off-screen rendering with OpenGL but it comes at a cost of complexity. FBO(Framebuffer Object Extension) is a similar technique for off-screen rendering that was reworked to be easy to use. The one notable limitation I was missing due to pbuffers was the ability to do loops inside kernels. Brook is a recent development in concern to gpgpu programming but it does not always use the latest technology used for example in mainstream computer games.

The reason why they use a language similar to standard ANSI C is because it is a well known language with straightforward syntax for ease of use for programmers already familiar to C. The extensions made to the language are able to use the stream programming model. [6] Some examples of this are:

- `float foo<10>;`
- `float bar<10, 10>;`
- `float2, float3, and float4 datatypes`
- `kernel void k(float s<>, float3 f, float a[10][10], out float o<>);`

- `iter float s<100> = iter(0.0f, 100.0f);`

When using `<10>` after a variable means that the variable is a *stream* on which can be operated on in parallel. With `<10, 10>` you declare a variable with 2 dimensions similar to arrays in ANSI C. Stream variables has limited read and write possibilities except for when inside kernels which is a function that operates on every element on a stream in parallel. Kernels looks like functions but a keyword called kernel precedes the declaration. The return type of a kernel is always void but you can use the out keyword to declare a variable to use as a return variable. Another extension is iterator streams that are invoked by a keyword called iter that precedes the variable declaration. Also there has been additional data types added to represent float2, float3 and float4. [6]

There also exists other ways to operate on streams outside of kernels and those functions are called streamRead and streamWrite. StreamRead is used to read from a variable and fill a stream and streamWrite is used to read from a stream and fill a variable. [6]

By using these simple extensions with standard ANSI C you can write simple programs but still take advantage of GPUs parallelism and speed. [6]

Chapter 3

Data-structures on the GPU

The results of the testing is presented here. The charts consists of BRT_OGL, BRT_CPU, C and Java and those are the different implementations. Both BRT_OGL and BRT_CPU are the Brook implementation but with different backends. BRT_OGL is the OpenGL backend and BRT_CPU is the CPU backend.

The first test, fig: [3.1], performed was to test the time it took to insert 100 elements last into a linked list. The real time of the operation showed that the C implementation was very fast and the Brook code running on the CPU finished second. The Brook code that ran on the GPU performed 10 times as slow as the Brook code on the CPU. Java finished last in the test.

The second test, fig: [**??**], was the same as the first one but this time 1000 elements was inserted into the list. The similarity to the results in the first test is very alike.

The last insertion test, fig: [**??**], was to try to insert 10000 elements into the list but then an obstacle was discovered. This error message from Brook was introduced:

```
Brook Runtime (gpu) - gpukernel.cpp(312):
No appropriate map technique found
```

For this reason it was difficult to run with lists with more than 1000 elements. I found out that there were limitations in the hardware and in Brook which resulted in that I couldn't run my initial planned tests with 10000 and 100000 elements. I did some further testing to see where the exact limitations of my hardware were and I found out that it was at 1024 elements. I wanted to test if there were some boundary where the GPU would be faster than the CPU but because of this limitation I couldn't.

After the insertion tests a traversion-test was performed with different amounts of elements. 100, fig: [3.2], and 1000, fig: [3.3], elements was tested in the list.

Then testing deletion of elements from random positions and with different amounts of elements. In this test also 100, fig: 3.4] and 1000, fig: [3.5] elements was tested.

The standard operations consisting of the insertion-, traversion- and deletion-tests all show very similar results. The C implementation always wins with almost no execution time. The Brook implementation running on the CPU get the second place but it still have 8 times longer execution time compared to the C implementation. In third place the Brook implementation running on OpenGL completes its tests with almost 10 times longer execution-time compared to the Brook CPU implementation. The Java implementation performed the worst by being last in all the tests. In addition to these standard operations I also tested insertion on the first position, insertion on a random position and deletion on the first position and at the last position but with similar results.

The kernel to confirm the GPUs capabilities used three float4 to do its multiplications. It was not possible to use a for-loop to represent multiples of multiplications because of limitations in Brook. The GPU performed very well, as shown in fig: [3.6]. I was surprised to see that the GPU was that good. The GPU finished in 0.1sec and the CPU finished the same calculations in 34.0sec. I only used floating-point numbers because I knew that the GPU operates best with that data-type. It was very straightforward to complete the kernel except the use of the for-loop limitation.

To determine what part of the Brook program that took the most time to complete an internal timer in the application was used. The timer that I used was derived from the examples in Brooks test-code. Starting with an insertion-test of 100 elements where timing of the actual list operations and variable declarations was performed. Those times told that the main parts of the Brook applications didn't take any significant time to complete. When I had excluded everything in the program, fig: [3.7], and only measured the actual initiation of the main-function I found out that the most consumed time in the application was the actual start-up.

Tests to see if the quality from the GPU was sufficient compared to the CPU in fig: [3.8] was also performed. The testing showed that the GPU conformed to the ANSI754 [26] standard. That means that the CPU and GPUs floating point precision is equal and thus the quality is sufficient.

The Brook framework makes it very general to program applications because it is possible to use almost identical implementations as in ordinary C programming. The only difference is that you have to be aware of the programmability and hardware limitations of current GPUs. For example you cannot do loops in kernels and you have to know how kernels work. When I implemented the linked list it was almost possible to use the C implementation without any changes. I only had to change my loops to use kernels and change my variables from ordinary data-types to streaming data types.

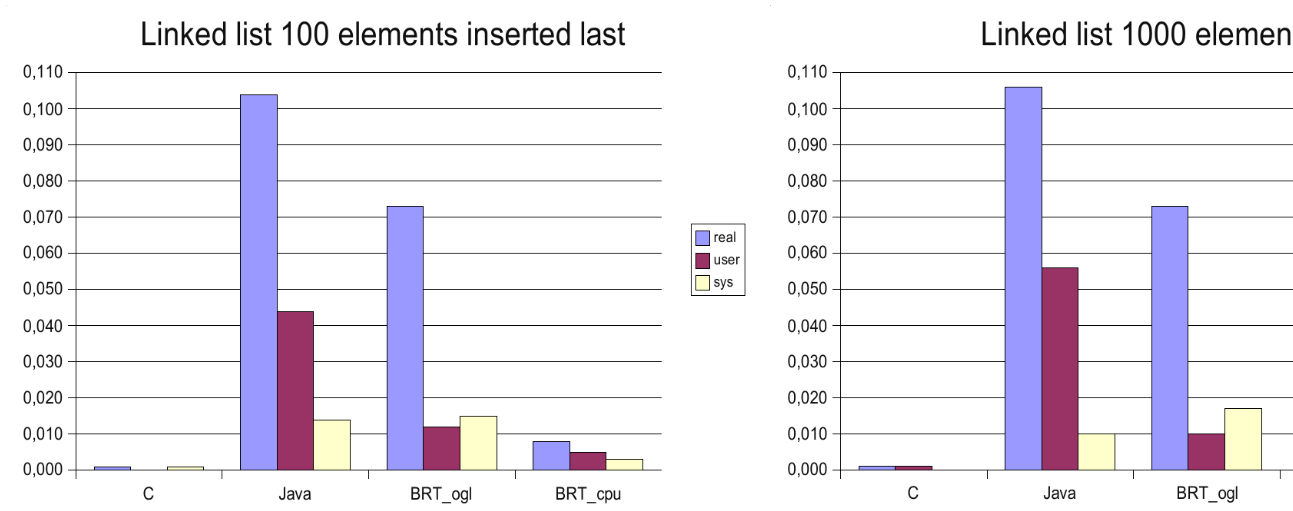


Figure 3.1: Results from 100 elements inserted into linked lists.

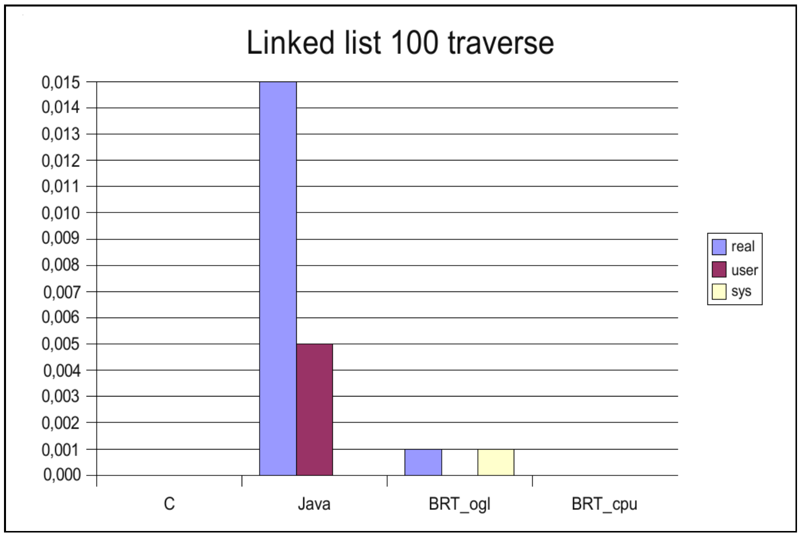


Figure 3.2: Traversing 100 elements in the list.

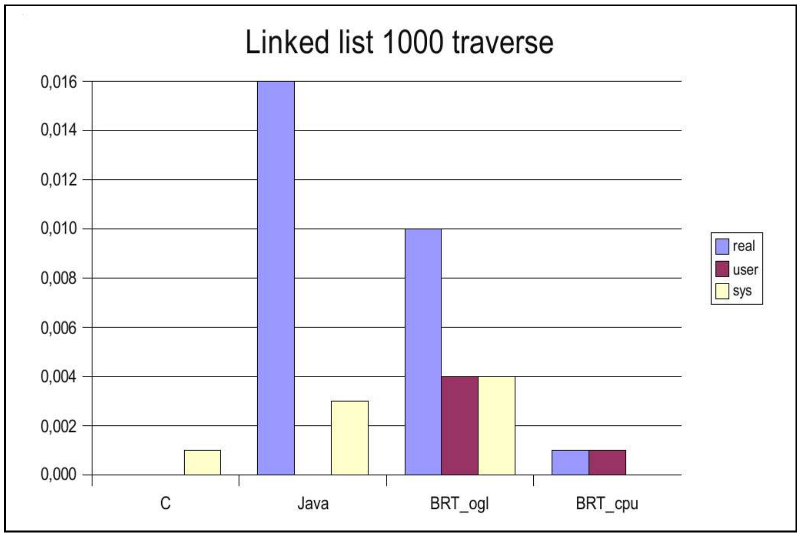


Figure 3.3: Traversing 1000 elements in the list.

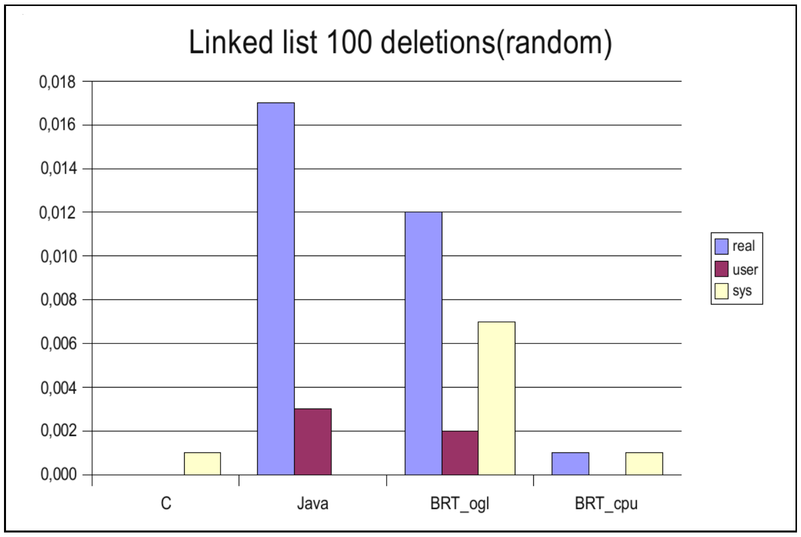


Figure 3.4: Deleting 100 elements at random positions in the list.

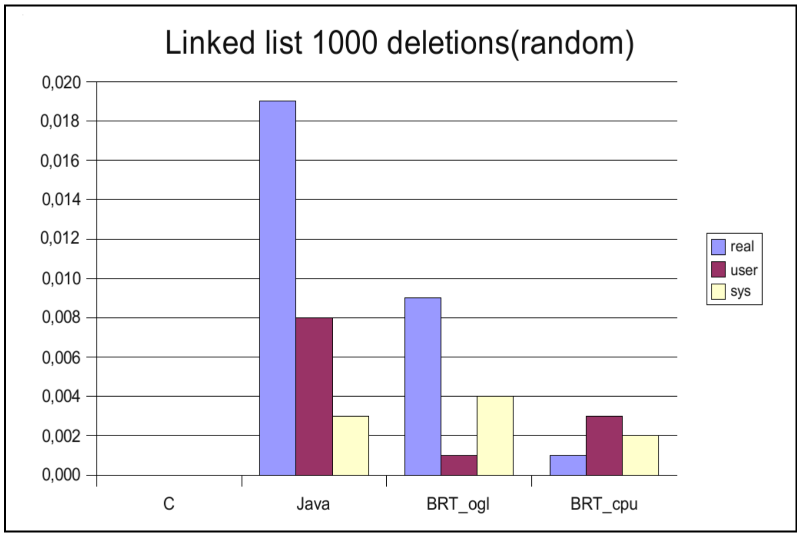


Figure 3.5: Deleting 1000 elements at random positions in the list.

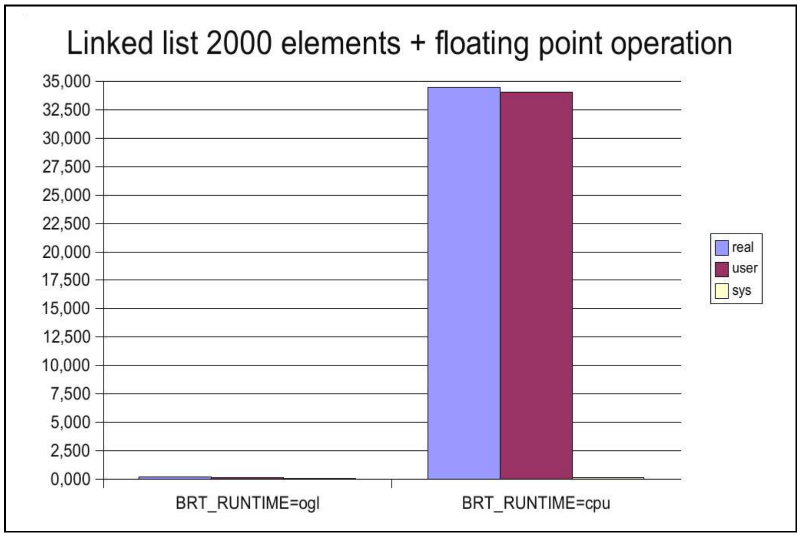


Figure 3.6: Results from the use of a heavy kernel consisting of many floating point calculations.

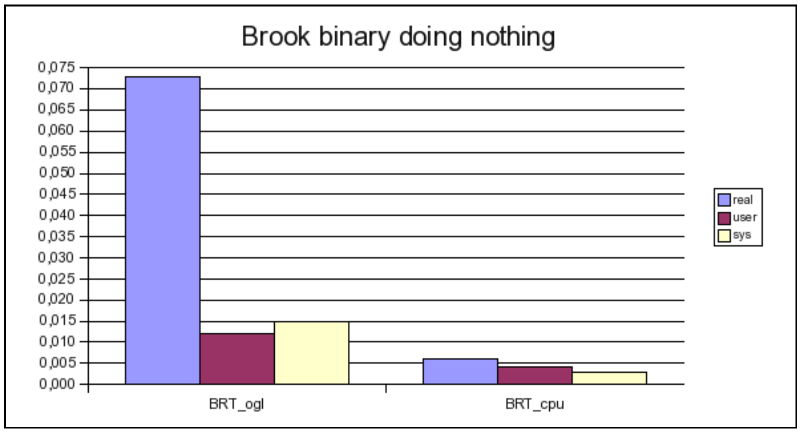
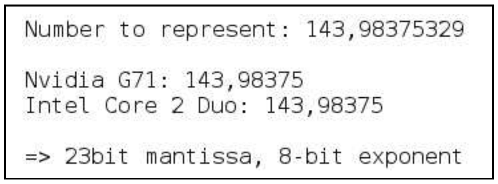


Figure 3.7: Brook program doing nothing.



```
Number to represent: 143,98375329

Nvidia G71: 143,98375
Intel Core 2 Duo: 143,98375

=> 23bit mantissa, 8-bit exponent
```

Figure 3.8: Quality tests.

Chapter 4

Discussion

The reason why standard operations performed so poorly could be due to the low bandwidth between the CPU and GPU. It was indicated from the internal timing tests that the bandwidth is not fast enough to transport the data to the GPU and the amount of calculations in a linked list is too low. You need high arithmetic intensity to really run GPUs [6] well i.e. for a simple kernel like the ones used in a linked list implementation you cannot do enough math to make up for the cost of the bandwidth required. The programs that perform well on the GPU is the one that do lots of math per byte read per kernel invocation and the ones that cannot be blocked into the cache by the CPU. Also because of the error that I received when I wanted to use more elements in my list I could not test if there was a limit when the number of elements would make the GPU more efficient. The limit in Brook is probably due to the maximal supported texture-sizes of the graphics processor. I examined this hardware limitation and found out from "tfoley" from BrookGPU forums that if "a single dimension is larger than the board supports (typically 2048), it tries to use the "address translation" code path". [28] The address translation code path is something that is only partially implemented in Brook and comes at cost of ease of use, speed and correctness.

My first hypothesises was that the data-structures would be very fast on the GPU and the tests show that this is not true. In the case of data-structures where the arithmetic intensity could be neglected it is not worth the trouble to run them on the GPU. The GPU has currently a very narrow usability area with its main focus still being graphical computations but with future enhancements the GPU will probably become more and more able to compute general purpose applications.

The really obvious choice to do with the current lines of graphics processors is to use alot of floating-point operations. According to my test of using a heavy kernel with lots of math and low bandwidth there is a significant advantage over the CPU. In the high-end graphics processors there is also integer support on the GPU which would possibly make the GPU fast on those kinds of operations too. Recent work in applications that take advantage of the GPUs power is for example Folding@home [22] which partially uses the Brook framework to do heavy calculations.

The second hypothesis that I have was that it would be difficult to program for the GPU, and that is a very hard statement to answer when only having my own experience as a reference but I still tried to anwer it. When I started using the framework it was hard to think according to the stream programming model but when using it for some time I thought it was pretty easy to use. It is just a new way of thinking. I first thought of the possibility to not use any framework and program directly to shaders through OpenGL and I now believe that it was wise of me no to do that. Because of the way that Brook simplifies the whole process by only extending the C programming language with small and well documented parts it was very easy to learn and implement using it. If you have a big project where you have a very specific problem with high mathematical intensity I think it is worth trying to do the calculations on the GPU because the threshold between not knowing stream programming and knowing it is not that high and the performance benefits could be huge.

Chapter 5

Conclusion

In conclusion it can be said that my first hypothesises was wrong that using a complex data-structure on the GPU would be faster than on the CPU. I found out that not all applications will drop onto current GPUs and dominate the CPU implementations. The applications which can take the most advantage off the GPUs performance is the ones that do much floating-point operations. It can also be said that the quality of the linked list was sufficient because the GPU and the CPU uses the same floating-point standard.

My second hypothesis is that it would not be viable to implement data-structures on the GPU because of it being to difficult to implement. After my extensive testing I can now assess that I was also partially wrong with this hypothesis. In the beginning it was hard to accustom to the stream programming model but when you got started it got easier. Other than that it was very easy probably because of the generality of the Brook framework.

Chapter 6

Future Work

Future work concerning the area of general purpose computing is to expand the testing when the programmability increases on the GPUs. The reason to do this is to be aware of the potential the general purpose applications can bring to computing. There are other frameworks in addition to Brook that could be tested but they are often commercial variants that costs to use. There is another free framework released recently called Nvidia CUDA [29] and it is supposed to simplify the process of general purpose computing even more and it would be interesting to test that but you would have to use the new G80 architecture. Maybe it would be possible to use a C implementation without any changes to run it on the GPU using CUDA.

There is backends in Brook that are not intended for GNU/Linux but with a Windows environment you could test them. Those backends include the popular DirectX API and the new CTM [15] device from ATI. It would also be interesting to test Brook with different CPU architectures like sparc [30], pa-risc [31] and powerpc [32].

Also an initial case-study of what is necessary to make general purpose computing work at the operative systems kernel. Making it transparent to every application and making it a supplement to the CPU. By doing this it could be possible to understand what is needed to write a driver for the Linux kernel.

Another interesting thing to investigate is the new cell architecture that the Playstation 3 is using. It uses a similar pipeline as the GPU and is to be used solely as a general purpose computing processor. Maybe this is a step for the GPU architecture to merge with the CPU architecture?

Bibliography

[1] Moore, Gordon E. 1965. *Cramming More Components onto Integrated Circuits.* [Online].
http://download.intel.com/research/silicon/moorespaper.pdf Visited: 2007-04-12

[2] Owens, John. 2005. *Streaming Architectures and Technology Trends.* GPU Gems 2

[3] Patersson, David A. 2004 *Latency Lags Bandwidth.* Communications of the ACM 71-75

[4] Kapasi, Ujval J.; Rixner, Scott.; Dally, William J. 2003 *Programmable Stream Processors.* IEEE 54-62

[5] Owens, John Douglas.; Dally, William J. 2002 *Computer Graphics On A Stream Architecture.* Ph.D. Thesis Stanford University.

[6] *BrookGPU framwork.* [Online].
http://graphics.stanford.edu/projects/brookgpu/ Visited: 2007-04-13

[7] Black, Paul E. "data structure" *Dictionary of Algorithms and Data Structures* [Online].
http://www.nist.gov/dads/ Visited: 2007-04-25

[8] Black, Paul E. "linked list" *Dictionary of Algorithms and Data Structures* [Online].
http://www.nist.gov/dads/ Visited: 2007-04-25

[9] *Data Structures and Algorithms - Lists* [Online].
http://www.cs.auckland.ac.nz/software/AlgAnim/lists.html Visited: 2007-05-14

[10] Parlante, Nick. 1999 *Linked List Basics* Stanford CS Education Library

[11] *Linked Lists in Java(tm)* [Online].
http://www.glenmccl.com/tip_005.htm Visited: 2007-05-14

[12] *American National Standards Institute* [Online].
http://www.ansi.org/

[13] *DirectX Developer Center* [Online].
http://msdn.microsoft.com/directx Visited: 2007-05-09

[14] *OpenGL - The Industry's Foundation for High Performance Graphics* [Oneline].
http://www.opengl.org Visited: 2007-05-09

[15] *ATI CTM Guide* [Online].
http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf Visited: 2007-04-26

[16] D. Sim Dietrich Jr. 2004 *Shader Model 3.0 - No limits* [Online].
http://www.csee.umbc.edu/ olano/s2004c01/ch15.pdf Visited: 2007-04-26

[17] *X.Org Foundation* [Online].
http://www.x.org Visited: 2007-05-10

[18] *GCC, the GNU Compiler Collection* [Online].
http://gcc.gnu.org Visited: 2007-05-10

[19] *GNU C Library* [Online].
http://www.gnu.org/software/libc/ Visited: 2007-05-10

[20] *Time* [Online].
http://www.gnu.org/software/time/ Visited: 2007-05-10

[21] *The standard GNU Bourne again shell* [Online].
http://www.gnu.org/software/bash/ Visited: 2007-05-10

[22] *Folding@Home Distributed Computing* [Online].
http://folding.stanford.edu/ Visited: 2007-05-10

[23] *Nvidia NSIST* [Online].
http://www.nvidia.com/content/nsist/module/what_gpu.asp Visited: 2007-05-01

[24] Fernando, Randima. 2004 *Foreword by David Kirk* GPU Gems

[25] Harris, Mark. 2005 *Mapping Computational Concepts to GPUs* Electronic Engineering Times

[26] Goldberg, David. 1991 *What Every Computer Scientist Should Know About Floating-Point Arithmetic* ACM Computing Surveys

[27] Buck, Ian. 2005 *Taking the Plunge into GPU Computing* GPU Gems 2

[28] *Citation from tfoley at BrookGPU forums.* [Online].
http://www.gpgpu.org/forums/viewtopic.php?t=866&highlight=appropriate+map+technique
Visited: 2007-05-14

[29] *NVIDIA CUDA - Revolutionary GPU Computing* [Online].
http://developer.nvidia.com/object/cuda.html Visited: 2007-05-10

[30] *UltraSPARC Processors* [Online].
http://www.sun.com/processors/ Visited: 2007-05-17

[31] *OpenPA: PA-RISC Information Resource* [Online].
http://openpa.net Visited: 2007-05-17

[32] *IBM Power Architecture* [Online].
http://www-03.ibm.com/chips/power/powerpc/ Visited: 2007-05-17

Chapter 7

Appendix

Data-structure of list in C and Brook

```
// list.c
struct Node
{
elementType element;
        position    next; // float
};

//list.h
struct Node;
typedef struct Node *ptrToNode;
typedef ptrToNode list;
typedef ptrToNode position;
```

Insert at a specified position function

```
void
insertSpecific( elementType X, list L, position P )
{
position tmpCell;

tmpCell = malloc( sizeof( struct Node ) );
if( tmpCell == NULL )
{
         printf("Could not allocate more memory\n");
}

tmpCell->element = X;
        tmpCell->next = P->next;
P->next = tmpCell;
}
```

Delete at a specified position

```
void
deleteSpecific( elementType X, list L )
{
        position P, tmpCell;

        P = findPrevious( X, L );

        if( !isLast( P, L ) )
        {
                tmpCell = P->next;
                P->next = tmpCell->next;
         free( tmpCell );
}
}
```

Is last in list?

```
int
isLast( position P, list L )
{
return P->next == NULL;
}
```

Delete entire list

```
void
deleteList( List L )
{
position P, Tmp;

P = L->next;  /* Header assumed */
L->next = NULL;

while( P != NULL )
        {
         Tmp = P->next;
           free( P );
           P = Tmp;
}
}
```

Find, return position in list if found. Otherwise NULL.

```
position
find( elementType X, list L )
{
position P;

P = L->next;
while( P != NULL && P->element != X )
P = P->next;

return P;
}
```

Free list

```
list
makeEmpty( List L )
{
if( L != NULL )
deleteList( L );

        L = malloc( sizeof( struct Node ) );

if( L == NULL )
printf("Cannot alloc more memory\n");

L->next = NULL;

return L;
}
```

Example of Brook code were the loops was replaced by kernels.

```
position
find( elmentType X, list L )
{
        position P;
position result;

        P = L->next;
kernelFind(P, X, &result);

        return P;
}
```

```
kernel void find(position p<>, elementtype X, out position result)
{
if(P != NULL && P->element != X)
{
P = P.next;
}
}
```

Kernel code when testing known performance advantages on the GPU.

```
kernel void
RunKernelHardWork(Element e<>, out float result<>)
{
        float4 o = {0.32002f, 0.32004f, 0.032002f, 1.23320f};
        float4 c = {0.002f, 0.004f, 0.0002f, 1.230f};
        float4 s = {0.213002f, 0.564004f, 6540.0002f, 4231.230f};

        o = c*s + s; o = c*o + s; o = c*o + s; o = c*o + s;
        o = c*o + s; o = c*o + s; o = c*o + s; o = c*o + s;
        o = c*o + s; o = c*o + s; o = c*o + s; o = c*o + s;
        o = c*o + s; o = c*o + s; o = c*o + s; o = c*o + s;
        o = c*o + s; o = c*o + s; o = c*o + s; o = c*o + s;
        o = c*o + s; o = c*o + s; o = c*o + s; o = c*o + s;
        o = c*o + s; o = c*o + s; o = c*o + s; o = c*o + s;
        o = c*o + s; o = c*o + s; o = c*o + s; o = c*o + s;
        o = c*o + s; o = c*o + s; o = c*o + s; o = c*o + s;
        o = c*o + s; o = c*o + s; o = c*o + s; o = c*o + s;
        o = c*o + s; o = c*o + s; o = c*o + s; o = c*o + s;
        o = c*o + s; o = c*o + s; o = c*o + s; o = c*o + s;
        o = c*o + s; o = c*o + s; o = c*o + s; o = c*o + s;
        o = c*o + s; o = c*o + s; o = c*o + s; o = c*o + s;
        o = c*o + s; o = c*o + s; o = c*o + s; o = c*o + s;
        o = c*o + s; o = c*o + s; o = c*o + s; o = c*o + s;
        o = c*s + s; o = c*o + s; o = c*o + s; o = c*o + s;
        o = c*o + s; o = c*o + s; o = c*o + s; o = c*o + s;
        o = c*o + s; o = c*o + s; o = c*o + s; o = c*o + s;
        o = c*o + s; o = c*o + s; o = c*o + s; o = c*o + s;
        o = c*o + s; o = c*o + s; o = c*o + s; o = c*o + s;
        o = c*o + s; o = c*o + s; o = c*o + s; o = c*o + s;
        o = c*o + s; o = c*o + s; o = c*o + s; o = c*o + s;
        o = c*o + s; o = c*o + s; o = c*o + s; o = c*o + s;
        o = c*o + s; o = c*o + s; o = c*o + s; o = c*o + s;
        o = c*o + s; o = c*o + s; o = c*o + s; o = c*o + s;
        o = c*o + s; o = c*o + s; o = c*o + s; o = c*o + s;
        o = c*o + s; o = c*o + s; o = c*o + s; o = c*o + s;
        o = c*o + s; o = c*o + s; o = c*o + s; o = c*o + s;
        o = c*o + s; o = c*o + s; o = c*o + s; o = c*o + s;
        o = c*o + s; o = c*o + s; o = c*o + s; o = c*o + s;
        o = c*o + s; o = c*o + s; o = c*o + s; o = c*o + s;

        result = e.val * o;
}
```