Exploring Graphics Processor Performance for General Purpose Applications

Pedro Trancoso and Maria Charalambous
Department of Computer Science, University of Cyprus
75 Kallipoleos Str., P.O.Box. 20537, CY-1678 Nicosia, Cyprus
{pedro,cs00cm}@cs.ucy.ac.cy

Abstract

Graphics processors are designed to perform many floating-point operations per second. Consequently, they are an attractive architecture for high-performance computing at a low cost. Nevertheless, it is still not very clear how to exploit all their potential for general-purpose applications.

In this work we present a comprehensive study of the performance of an application executing on the GPU. In addition, we analyze the possibility of using the graphics card to extend the life-time of a computer system.

In our experiments we compare the execution on a mid-class GPU (NVIDIA GeForce FX 5700LE) with a high-end CPU (Pentium 4 3.2GHz). The results show that to achieve high speedup with the GPU you need to: (1) format the vectors into two-dimensional arrays; (2) process large data arrays; and (3) perform a considerable amount of operations per data element. Finally, we study the performance when upgrading a low-end system by simply adding a GPU. This solution is cheaper, results in smaller power consumption and achieves higher speedup (8.1x versus 1.3x) than a full upgrade to a new high-end system.

1. Introduction

The demands from the game applications market have been driving the development of better and faster architectures. The addition of SIMD instruction extensions to the traditional Instruction Set Architecture (ISA) was done to support the demands of the multimedia and gaming applications. The obvious development though, can be observed in the Graphics Cards and more specifically in the Graphics Processing Units (GPU). The GPUs are the responsible entities for drawing the fast moving images that we observe on the computer screens. To achieve those real-time realistic animations, the GPUs must perform many floating-point operations per second. As such, and given that the work performed by the GPUs is dedicated to these applications, the GPUs are forced to offer many more computational resources than the general purpose processors (CPU). Given the characteristics of these applications, performance can easily be achieved from the use of vector units, *i.e.* using the SIMD programming model. In some way, these GPUs have similar characteristics with many original supercomputers (*e.g.* Cray supercomputers). However, the first GPU models were only capable of handling a restricted number of hard-wired graphics operations. A determinant factor in the development of the latest GPU models is that now they are programmable, offering the capability of executing user's code. As such, users, and game writers in particular, may write their own graphics operations. In addition, the programmability has opened the power of the GPU for other non-graphics applications. This has lead to the rising interest in a new research field known as General Purpose computation on Graphics Processing Units or GPGPU [6].

Several general-purpose applications have been mapped to the GPU. Examples of such applications include: dense matrix multiply [9], linear algebra operations [8], sparse matrix solvers for conjugate gradient and multigrid [1], and database operations [5].

Although different manufacturers offer different models of GPUs, the interface exported for programming them is standard and supported by the graphics card drivers. Currently the two major standards for the GPU interface are OpenGL [14] and DirectX [12]. Because these interfaces were written for programming graphics operations, they are not trivial to be used by general-purpose applications. Consequently there are some efforts into making environments and tools that make the GPU programming easier for general-purpose applications. One such environment is BrookGPU [2]. BrookGPU makes some extensions to ANSI C in order to support the execution of general-purpose applications on the GPU, making it relatively easy to port an application. The main issue, though, is not the porting but exploiting the GPU's features.

The contributions of the work presented in this paper are twofold. First, this work presents a comprehensive study of the performance of a simple application executing on the

GPU. Several parameters are analyzed such as the “shape” and type of the data passed to the functions executing on the GPU, the input data size, and the number of operations per data element. Second, this work analyzes the extension of the life-time of a computer system by adding graphics cards as opposed to upgrading the whole system.

We executed several experiments using a simple application in order to understand the correlation between the changes made and the results obtained. We compared the execution on a mid-class GPU (NVIDIA FX 5700LE) with a high-end CPU (Pentium 4 3.2GHz). The results showed that to achieve high speedup with the GPU it is necessary to: (1) format the vectors into two-dimensional arrays; (2) process large data arrays; and (3) perform a considerable amount of simple operations per data element. The positive result was that excluding the issue with the dimension of the input data, the GPU execution always shows a speedup larger than one. The most interesting result is that when considering to upgrade a system (Pentium III 733MHz), simply adding a GPU (5700LE) to the existing system is a solution that is less costly, results in smaller power consumption and achieves higher speedup (8.1x versus 1.3x) than a full upgrade to a new high-end system (Pentium 4 3.2GHz).

This paper is organized as follows: Section 2 presents the GPU architecture along with its programming environment. Section 3 shows the experimental setup and Section 4 presents the experiments and results obtained. Section 5 discusses the limitations of the use of the GPU for general purpose applications. Finally the conclusions are presented in Section 6.

2. Graphics Processing Unit

2.1. Architecture

The GPU is an interesting architecture as it offers a large degree of parallelism at a relatively low cost. Its operations are similar to the well known *vector processing model*. This model is also known from Flynn’s taxonomy [4] as *Single Instruction, Multiple Data* or *SIMD*. As such, it is natural that the GPU will be able to perform well on many of the applications that in the past were forced to be executed on vector supercomputers.

Another characteristic of the simple parallel architecture of the GPU is that it allows for its performance to grow at a rate faster than the well known Moore’s law. In fact, the GPU’s performance has been increasing at a rate of 2.5 to 3.0x a year as opposed to 1.4x for the CPU.

The general architecture of the GPU is depicted in Figure 1. Notice that the GPU includes two different types of processing units: vertex and pixel (or fragment) processors. This terminology comes from the graphics operations that each one is responsible for. For example, the vertex processor performs mathematical operations that transform a vertex into a screen position. This result is then pipelined to the pixel or fragment processor, which performs the texturing operations.

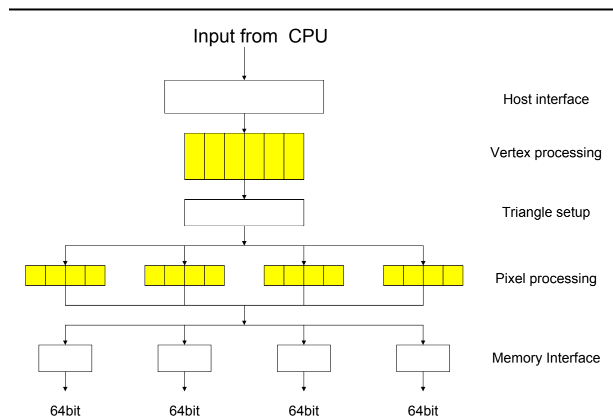

Figure 1. GPU architecture.

2.2. Programming Environment

As mentioned before, programming the GPU in a high-level language is a recent development. The first high-level language programming environments were developed for graphics applications in mind. Such examples are Cg from NVIDIA [10] and OpenGL Shading Language [7]. Although helpful, they make the job of mapping a general purpose application a considerable task. Therefore, some research teams are working on developing high-level language programming environments for general purpose programming. One such environment is BrookGPU [2] from Stanford University.

BrookGPU is an extension to the standard ANSI C and is designed to facilitate the porting of general purpose applications to the GPU. The main differences from the standard C language are the introduction of the concept of *stream* variables, and *kernel* and *reduction* functions. The programming model offered by BrookGPU for the functions to be executed on the GPU is a *streaming* model. In this model a function processes streams, *i.e.* sequences of data, but operates on a single element at a time.

Using BrookGPU, a function that is executed on the GPU can be of two types: *kernel* and *reduction*. The former is a general function that accepts multiple input and output parameters which may be of type stream or not. The latter is a function that takes multiple stream parameters but returns a single value. This is used to execute the known reduction operations such as a sum of all the values in a stream.

Finally, BrookGPU has the advantage that it is only necessary to write the code once and its runtime takes care of selecting the correct implementation. For example, the same code can execute either on the CPU, or the GPU, using the OpenGL or the DirectX interface. In addition, it also provides an indirection layer such that the user does not need to be aware if the card has an NVIDIA or an ATI chip, for example.

3. Experimental Setup

For the experiments presented in this paper we used one graphics card and two computer setups. The graphics card used was an NVIDIA GeForce FX 5700 LE [11]. This card has an NV36 graphics processor clocked at 250MHz, 128MB DDR video memory clocked at 200MHz and the data transfers with the PC are done through the AGP interface. The NV36 processor includes 3 vertex and 4 pixel pipelines [15].

As for the computer systems we used two different setups: *low-end*, an Intel Pentium III 733MHz based system with 512MB RAM; and *high-end*, an Intel Pentium 4 3.2GHz HT based system with 1GB RAM. Unless mentioned, the experimental results presented were collected on the *high-end* system.

The application used to test the potential of the graphics processor is a simple function that takes two input arrays of data or streams and operates on them with a simple addition operation. Although simple, the operations in this application are common in most scientific workloads. Its characteristics make it easier to reveal the impact on the performance produced by certain code or data optimizations. This application is provided with the BrookGPU [2] environment and is called *accumulate*. The main routine of *accumulate* that is executed on the GPU is presented in Figure 2. The complete *accumulate* code can be found together with the BrookGPU distribution [3].

```
kernel void
sum(float a<>, float b<>, out float c<>)
{
   c = a + b;
}
```

Figure 2. Main accumulate routine.

The environment used was BrookGPU version 0.3 [2] as described in Section 2. For the experiments, BrookGPU was compiled with the Microsoft Visual Studio C++ compiler using BrookGPU's *make release* command, *i.e.* with full code optimizations for both the *low-end* and *high-end* setups.

The results reported are based on the system time measurements, collected from the program execution, using the processor's hardware performance counters. The access to the performance counters is achieved with the *QueryPerformanceCounter* library function. The time is measured in three portions: the time to read the data into the card, the time to execute the function, and the time to write the data back to the system's memory. Unless mentioned otherwise the time reported accounts for the total time that it takes to execute the function, excluding the variable initialization phase.

Also, for every experiment we compare the execution time of running the code on the regular CPU of the system and on the GPU of the graphics card on the system. We measure these two situations using the same code as the BrookGPU runtime allows the user to decide where the code should be executed depending on the value of an environment variable (*BRT_RUNTIME*). If the variable *BRT_RUNTIME* is set to *cpu*, the code will be executed on the system's CPU while if it is set to *nv30gl* it will be executed on the card's GPU.

4. Experimental Results

4.1. Formatting the data for the GPU

In a general-purpose vector function the data is usually one-dimensional. In this experiment we test what is the impact on performance of passing the data as-is, *i.e.* one-dimensional, and what happens if we format the data to be passed as a two-dimensional array.

The chart in Figure 3 shows the speedup of the GPU execution comparing to the baseline CPU execution for two baseline one-dimensional cases: stream size of 100 and 2000 floating point (fp) elements.

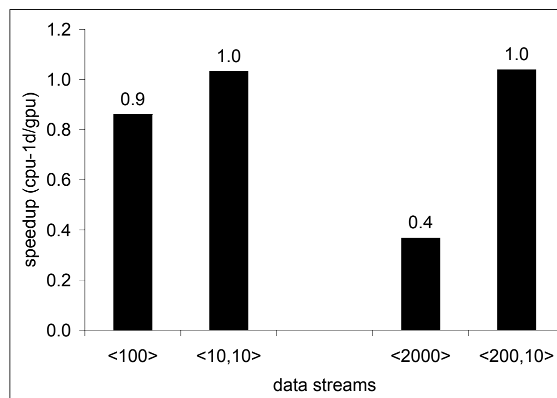

Figure 3. GPU speedup for 1D and 2D input data streams.

In Figure 3 it is possible to observe that the speedup is largely affected by the “shape” of the data. As shown, the speedup for the one-dimensional cases is always smaller than one. In addition, this problem increases as the input data increases (speedup is 0.9 for 100, while only 0.4 for 2000 fp elements). This is more obvious when we observe that for the 100 fp elements the increase from 1D to 2D is 11%, while for the 2000 fp element case this increase is 150%. As a consequence, we are lead to conclude that in order to exploit the performance of the GPU we need to reformat the input data to be two-dimensional.

The next question is whether for a certain fixed data size the size of the second dimension plays any role in the performance. To answer this question we executed an experiment with a large input data size of 20000 fp elements. The results of the GPU speedup are presented in Figure 4.

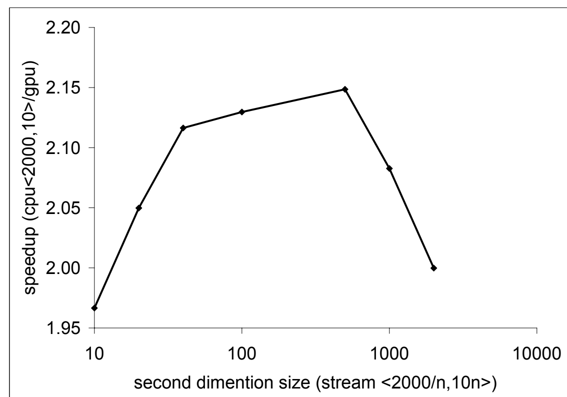

Figure 4. GPU speedup for varying second dimension size of 2D input data streams.

The results from Figure 4 show that up to a certain point, increasing the second dimension results in higher speedup. Beyond that point, increasing the second dimension results in decrease of the speedup. Notice that although these changes are quite small (9%) they are an indication that the highest speedup is achieved when the data is closer to a “square shape”. This is mostly due to the fact that the GPUs were designed to operate on images (two-dimensional) and not on vectors. Notice that, in most of the other experiments, the upper limit data set size used is <2048,40> which is not of “square” shape but uses fully the first dimension and according to the results in Figure 4 is within 2% of the highest performance.

4.2. Input Data Size

The next set of experiments addresses the question of how the speedup depends on the input data size. Given that there are data transfer overheads involved in the execution of a piece of code on the GPU, GPU execution will not be beneficial for all cases. As such, we are interested in learning what is the data size, for a simple application like the *accumulate*, when it becomes beneficial to execute the code on the GPU. Figure 5 presents the speedup of the GPU compared to the CPU for different input data sizes and one- and two-dimensional input data.

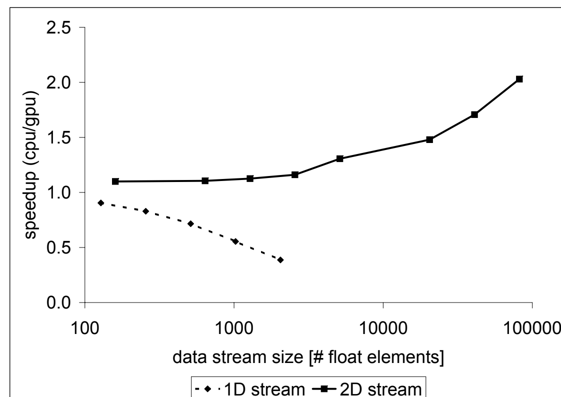

Figure 5. GPU speedup for varying input data size for 1D and 2D input data streams.

From Figure 5 it is possible to observe that, on the one hand, when the data is one-dimensional the speedup is always lower than one and it actually decreases as the data size increases. On the other hand, when the data is two-dimensional the speedup is always larger than one and it increases linearly as the data size increases (notice that the scale of the horizontal axis is logarithmic). Consequently, for the *accumulate* example, in order to achieve speedup on the GPU we need the data to be two-dimensional and the larger the data set, the larger the speedup that will be observed. For data streams of 81920 (<2048,40>) fp elements the GPU achieves a speedup of 2.0.

4.3. Number of Operations

While the previous section discussed the speedup depending on the data intensity of the application, this section analyzes the speedup for different degrees of computation intensity of the application.

In order to perform this experiment we have changed the *sum* function shown in Figure 2 to have only a single input vector. The operation performed is a multiple addition of the single input. The number of times the element is added depends on the number of operations we want to test. Therefore, the function code for testing 3 operations is the one in Figure 6.

```
kernel void
sum(float a<>, out float b<>)
{
   b = a + a + a + a;
}
```

Figure 6. **Main accumulate routine.**

In Figure 7 we present the speedup for the GPU compared to the CPU execution as we change the number of operations, for three different fixed size input data streams: 100 fp elements ($<$10,10$>$), 2000 fp elements ($<$200,10$>$), and 20000 fp elements ($<$2000,10$>$).

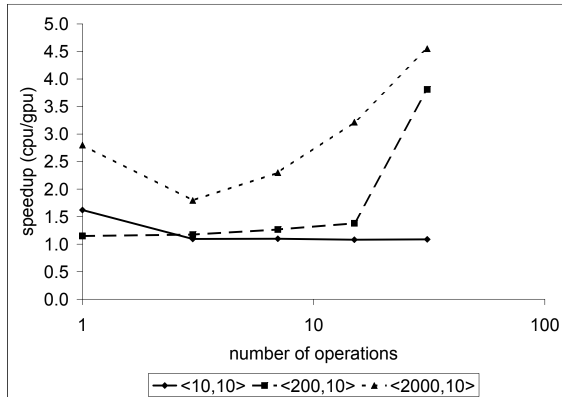

Figure 7. **GPU speedup for varying number of operations with constant input data streams.**

Figure 7 shows that, similarly to the previous case of the data set size, as the number of operations increases, the speedup increases. In this case, for the largest input data set of 20000 fp elements, for 31 operations we can observe a speedup of 4.6.

Although the previous results are an indication of what happens when the number of operations increases for a fixed number of input parameters, in real applications as the number of operations increases so do the number of parameters in the function. As such we designed a different experiment to study the impact of increasing the number of operations on the speedup. In this experiment we changed the *sum* function to have two ($c = a+b$), four ($e = a+b+c+d$), and eight ($i = a + b + c + d + e + f + g + h$) parameters. Then we increase the number of operations by replicating the sum operation n times, which we call the *effort*. Therefore, the code for the *sum* for four parameters and effort two is the one presented in Figure 8.

This experiment is again performed for the three fixed data set sizes as presented in the previous experiment. The results of the speedup for the GPU compared to the CPU

```
kernel void
sum(float a<>, float b<>, float c<>,
    float d<>, out float e<>)
{
   e = a + b + c + d;
   e = a + b + c + d;
}
```

Figure 8. **Main accumulate routine for four parameters and effort two.**

are depicted in Figure 9. In this Figure we can observe three different sub-charts, each one corresponding to a different number of *sum* operations: two (*c*=), four (*e*=), and eight(*i*=). On the x-axis we represent the different *effort* values ranging from 1 to 16. Each sub-chart includes three curves, each one representing a different data set size.

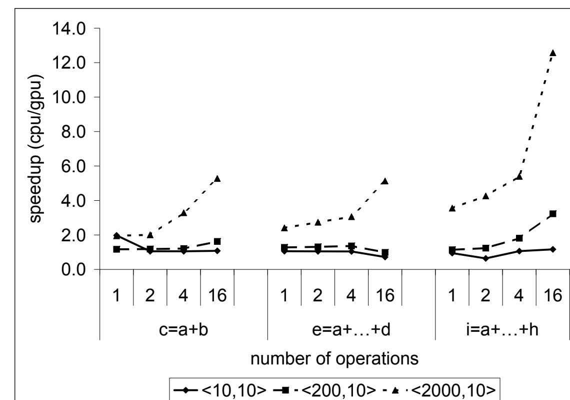

Figure 9. **GPU speedup for varying number of operations and data parameters with constant input data streams.**

From Figure 9 we observe again that the larger the data set the larger the speedup. Also, as in the previous experiment, an increase in the number of operations, in this case expressed by larger *effort* value, also shows an increase in the speedup. Finally, as we increase the number of parameters it is again possible to observe an increase in the speedup. Ultimately, for the case of the data set size of 20000 fp elements, eight parameters and effort of 16, the GPU achieves a speedup of 12.6 over the CPU. This is an impressive performance benefit of the GPU over the CPU. Notice though that in this extreme case the sum function has a total of 160000 fp elements as input and performs 2240000 fp additions. This results in a ratio of 14 operations per data element. This ratio may not exist, without any changes, in most applications. As such, we will observe

speedup values that are less than 12.6. Nevertheless this result is a strong motivation for trying to rearrange the code in order to increase the number of operations per data element so that we exploit the performance benefits of the GPU.

4.4. Data Types

Given the characteristics of the graphics applications, the GPU programming model offers some extra data types that are interesting to analyze. These data types are data structures with multiple float elements that are used in graphics applications to represent, for example, the position of a point in a multi-dimensional space. As such, the programming model offers the *floatN* types where N is a number between 2 and 4. For example, a float4 variable has four fields x, y, z, and w.

In our experiments we compared an input stream composed of regular float elements with an input stream composed of float2 and float4 elements. In order to study only the impact of the data type change, we maintain the total number of data elements constant, *i.e.* we compare a stream of <2048,40> float elements with a stream of <2048,20> float2 elements and one of <2048,10> float4 elements. The code for the *sum* function changes slightly again as now instead of the simple $c = a + b$ operation, we have to perform the operation on all components of the new type. The code for the *sum* function using the float2 data type is the one found in Figure 10.

```
kernel void
sum(float2 a<>, float2 b<>, out float2 c<>)
{
   c.x = a.x + b.x;
   c.y = a.y + b.y;
}
```

Figure 10. Main accumulate routine for float2 data type.

The experimental results comparing the speedup obtained for the GPU comparing to the baseline CPU (with float data type), for float, float2, and float4 are depicted in Figure 11.

The results in Figure 11 show an interesting behavior of the floatN data type. While using the original float type results in a steady speedup increase as the data size increases, the float2 results show very bad performance. The float4 are more encouraging as the speedup is substantially larger than the one obtained with the original float type. Nevertheless, as the input data set increases, the speedup decreases

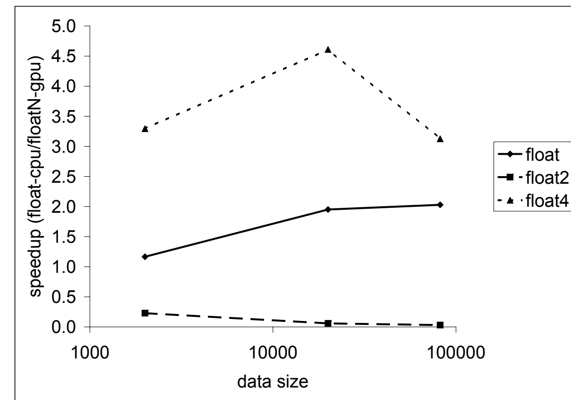

Figure 11. GPU speedup for varying the data type of the input data streams.

and almost reaches the level of the original float type. The reason for this behavior seems to be related to the transfer of the float4 results to the system's memory. The benefit of float4 in comparison with float is justified from a more efficient result transfer for the smaller data streams. A detailed analysis of the execution time shows that the transfer of the results accounts for 25% for float4 and the 2000 data element streams. For float the same transfer accounts for 33% of the total execution time.

4.5. Exploring Function Implementation Options

In this section we analyze, for the original code and fixed input data size, different implementation options. We may consider three different types of implementation of the same function using different techniques.

In the first technique, called *split*, we split the input streams into smaller ones. Considering that originally we have two input streams of size n, with *split2* we will have four input streams of size $n/2$. The rationale is to increase the number of operations performed within the function that executes on the GPU.

In the second technique, called *call*, the input streams are again split but instead of passing the data as extra parameters, the function will be called more times. In the same example as above, the *call2* technique would split the two input streams of size n into four streams of size n/s, but the *sum* function would be called with the smaller streams. The difference is that in order to complete the operations, it requires for the function *sum* to be called a second time with the second portion of the data streams.

Finally, the third technique is the one where the data type is changed from float to floatN as described in the previous section.

The results from these experiments showing the speedup of the GPU over the CPU are depicted in Figure 12. Notice

that the *call* technique is only used for the <2000,10> and <2048,40> data streams.

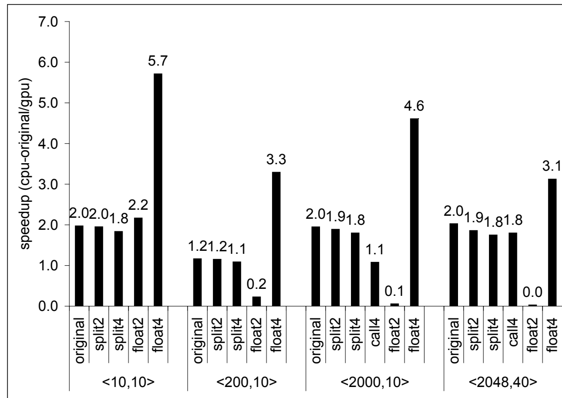

Figure 12. GPU speedup for function implementation options.

These results may be easily summarized as all different implementation options other than float4 result in a worse speedup than the original one. As such, only the technique of changing the input and output data type for float4 seems to be an alternative to the original implementation.

4.6. Upgrading Options

In this last section we analyze different upgrading options and their impact on the speedup. For this experiment our baseline system is the previously referred to as *low-end* system, a system equipped with an Intel Pentium III 733MHz and 512MB RAM (*P3*). The first upgrade we consider is to add the graphics card to the system (*P3+GPU*). An alternative upgrade is the one where the system is changed for a *high-end* system, one equipped with an Intel Pentium 4 3.2GHz and 1GB RAM (*P4*). The last upgrade is the ultimate where we consider the high-end system together with the graphics card (*P4+GPU*).

The different systems were tested with three sizes of input data stream: 100 (<10,10>), 2000 (<200,10>) and 20000 (<2000,10>) fp elements. The speedup results for these different system configurations, compared to the baseline *P3* system, are depicted in Figure 13.

From Figure 13 it is relevant to notice that only for the smaller data set size the *P4* configuration achieves a better performance than the *P3+GPU*. This means that for medium to large data set sizes the option of upgrading the system by only adding a graphics card of medium performance results in higher overall performance than upgrading the whole system to a high-end one. This result is obvious for the larger data set size tested as the *P4* system achieves

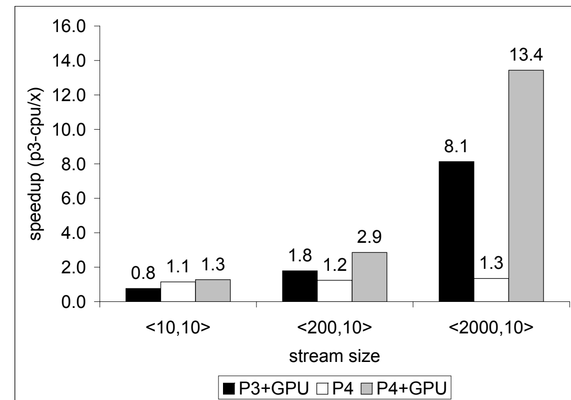

Figure 13. Speedup for different system configurations.

only a modest 1.3 speedup in comparison with the original *P3* system. In contrast, for the same data set size, adding the *GPU* to the *P3* system results in a speedup of 8.1.

These results are even more impressive if the prices of the upgrades are considered. From PriceWatch [13] it is possible to find that the NVIDIA 5700LE card costs approximately US$75 and the Pentium 4 US$200. This means that for the large data set the *P3+GPU* achieves a speedup that is larger by a factor of 6x at a cost that is only a little more than one third of the *P4* upgrade.

Furthermore, given today's concern about power-consumption, it is relevant to notice that while the NVIDIA chip consumes approximately 24W [16], the Pentium 4 3.2GHz consumes approximately 5.5x more power, as it consumes more than 130W.

Overall, the experimental results have shown that upgrading a modest system with a graphics card will result in a larger speedup than upgrading to a high-end system. This can effectively be used to extend the life-time of a system. In addition, the graphics card is less costly and consumes less power compared to the high-end CPU. If all factors are considered then the effective speedup achieved is overwhelmingly larger for the GPU comparing to a high-end CPU.

5. Graphics Processor Limitations

As a result of our experiments we observed some limitations of the GPU for general-purpose applications.

First, the GPU was designed to process images for the screen. As such, it may handle as many pixels as the maximum resolution of the image it can process. In reality, this means that the largest size for a dimension of a data stream is 2048 floating point elements. Also, as it is known, the size of the Video RAM limits the maximum resolution. For general-purpose programming this means that the number

of elements to be processed by a streaming operation is limited to the size of the Video RAM.

Another limitation regards the number of input parameters that can be passed to a GPU function. This number is limited to eight as currently to produce a realistic image it makes no sense to have more than eight pixel characteristics (e.g. coloring, smoothing, lighting, etc.).

One last limitation is the fact that the normal operation of the graphics card is to read image-related data from the main memory, process this data to produce an image, and output this image to the screen via the video output of the card. In regular graphics operations there is no need to transfer the results back to the main memory of the system. Nevertheless, this results in a serious limitation to general-purpose programming as the result must be stored in main memory. As such, while the GPU execution time is quite small, for large result streams the wait time for the transfer may be considerable. In our experiment we observed in some situations more than 50% of result transfer time. This limitation, though, is currently being lifted by the latest card models that connect to the PCI-Express ports and offer a higher-bandwidth for the return of data.

6. Conclusions

This work focused on how to exploit the GPU's performance for general purpose applications.

The contributions of the work presented in this paper are twofold. First, this work presents a comprehensive study of the performance of a simple application executing on the GPU. Several parameters are analyzed such as the "shape" and type of the data passed to the functions executing on the GPU, the input data size, and the number of operations per data element. Second, this work analyzes the extension of the life-time of a computer system by adding graphics cards as opposed to upgrading the whole system.

In our experiments we compared the execution on a mid-class GPU (NVIDIA GeForce FX 5700LE) with a high-end CPU (Pentium 4 3.2GHz). The results showed that to achieve high speedup with the GPU you need to: (1) format the vectors into two-dimensional arrays; (2) process large data arrays; and (3) perform a considerable amount of operations per data element. Finally, we studied the performance when upgrading a low-end system by simply adding a GPU. This solution is cheaper, results in smaller power consumption and achieves higher speedup (8.1x versus 1.3x) than a full upgrade to a new high-end system.

We are currently extending this work by performing the same analysis on different GPU models. In addition we are using the results presented in this work in order to build a model that determines at compile time if the code should be scheduled on the CPU or the GPU.

Acknowledgments

We would like to thank Elena Hadjikyriacou-Trancoso and Kyriakos Stavrou for the valuable reviewing effort. Also, we would like to thank the anonymous reviewers for their input on the work.

References

[1] J. Bolz, I. Farmer, E. Grinspun, and P. Schrooder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Transactions on Graphics*, 22(3):917–924, 2003.

[2] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Transactions on Graphics*, 23(3):777–786, 2004.

[3] I. Buck, T. Foley, D. Horn, J. Sugerman, P. Hanrahan, M. Houston, and K. Fatahalian. BrookGPU. http://graphics.stanford.edu/projects/brookgpu/, 2005.

[4] M. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[5] N. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast Computation of Database Operations using Graphics Processors. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 215–226. ACM Press, 2004.

[6] GPGPU. General-Purpose Computation Using Graphics Hardware. http://www.gpgpu.org/, 2005.

[7] J. Kessenich, D. Baldwin, and R. Rost. The OpenGL Shading Language, April 2004.

[8] J. Kruger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics*, 22(3):908–916, 2003.

[9] E. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 55–55. ACM Press, 2001.

[10] W. Mark, R. Glanville, K. Akeley, and M. Kilgard. Cg: a system for programming graphics hardware in a C-like language. *ACM Transactions on Graphics*, 22(3):896–907, 2003.

[11] NVIDIA. NVIDIA GeForce FX: Performance. http://www.nvidia.com/page/fx_5700.html, 2005.

[12] C. Peeper. DirectX High Level Shading Language. Microsoft Meltdown UK Presentation, Microsoft Corporation, 2002.

[13] PriceWatch. Price Comparison Search Engine. http://www.pricewatch.com, 2005.

[14] M. Segal and K. Akeley. The OpenGL Graphics System: A Specification (Version 2.0), October 2004.

[15] TechPowerUp. GPU Database. http://www.techpowerup.com/gpudb/, 2005.

[16] T. Tscheblockov. Power Consumption of Contemporary Graphics Accelerators. http://www.xbitlabs.com/articles/video/display/ati-vs-nv-power.html, 2004.