Performance Study of Interference on Sharing GPU and CPU Resources with Multiple Applications

Shinichi Yamagiwa
INESC-ID
Rua Alves Redol, 9
1000-029 Lisboa, Portugal
yama@inesc-id.pt

Koichi Wada
Department of Computer Science
University of Tsukuba
Tenno-dai 1-1-1, Tsukuba, Ibaraki, Japan
wada@cs.tsukuba.ac.jp

Abstract

In the last years, the performance and capabilities of Graphics Processing Units (GPUs) improved drastically, mostly due to the demands of the entertainment market, with consumers and companies alike pushing for improvements in the level of visual fidelity, which is only achieved with high performing GPU solutions. Beside the entertainment market, there is an ongoing global research effort for using such immense computing power for applications beyond graphics, such as the domain of general purpose computing. Efficiently combining these GPUs resources with existing CPU resources is also an important and open research task. This paper is a contribution to that effort, focusing on analysis of performance factors of combining both resource types, while introducing also a novel job scheduler that manages these two resources. Through experimental performance evaluation, this paper reports what are the most important factors and design considerations that must be taken into account while designing such job scheduler.

1 Introduction

Current generation of graphics processing units (GPUs) have an higher floating point performance than current CPUs. Both GeForce series from NVIDIA and RADEON series from ATI can achieve up to 300 GFLOPS. Such raw performance numbers as already attracted researchers from all over world to tapping into these resources for scientific and general purpose applications [7].

There are job schedulers that already can dispatch processes to run on CPU resource(s) of machine(s), such as Condor [5] and OpenPBS [6]. Those schedulers have been used in parallel and distributed computing environment such as cluster computers and GRID environment. Those job schedulers can harvest the potential performance of a whole computer system, allowing the execution of demanding computational applications due to its effective assignment of computing resources.

Traditional job schedulers are targeted towards CPU-based applications. Therefore, task assignment is only performed for CPU related resources. However, due to the recent trends towards GPGPU usage mentioned previously, and because conventional job schedulers are oblivious to the existence of high performance resources such as GPUs, it is necessary to introduce means to efficiently manage all these resources and their scheduling among CPU- and GPU-based applications, while efficiently using the GPU and CPU resources available on any given machine.

This paper is focused on performance analysis, while introducing a new mechanism for assigning CPU and GPU resources to running applications. This mechanism is needed for a new job scheduler that is both aware of the GPU and CPU resources on a given machine. Moreover, this paper proposes parameters to be added to the job scheduler, that are to be used as hints for effective resource assignment, while analyzing any possible interference between GPU- and CPU-based applications from experimental performance evaluation.

This paper is organized as follows: Section 2 introduces the background of general purpose computation using GPUs and job schedulers. Section 3 explains CPU- and GPU-based applications and how to evaluate their performance. Section 4 performs experimental analysis and details the best methods for assigning resources effectively. Finally, section 5 concludes this paper.

2 Research Backgrounds

2.1 GPGPU

Most graphical applications, especially one using 3D graphics visualization techniques, have dramatically ad-

978-1-4244-3750-4/09/$25.00 ©2009 IEEE

vanced in the last decade. High performing GPU devices are now ubiquitous. Even commodity personal computers are fitted with these.

Nowadays, high performance computing has shifted most of algorithmically research effort towards using the GPUs as a replacement for traditional CPU as the workhorse for computational demanding applications. The application of General-Purpose computation on GPU (GPGPU) concept allows to achieve high performance levels [7]. Compiler-oriented support for programming general-purpose applications using GPU resources has already been proposed [2]. Moreover, a cluster-based approach for using PCs with high performance GPUs has been reported in [3].

We will now introduce the processing steps and architectural blocks that allow GPUs to generate graphics objects in order to be displayed on a screen. The GPU acts as a co-processor of the CPU via a peripheral bus, such as the AGP or the PCI Express bus. A Video RAM (VRAM) is connected to the GPU, which is to where the GPU reads/writes graphics objects. The CPU controls the GPU operation by downloading the required data objects into the VRAM and its associated program into the GPU.

Figure 1 shows the processing steps performed by GPU to create a graphical image, and store it in a frame buffer in order to be displayed in a screen. First, graphics data are prepared as a set of normalized vertices of objects on a referential axis defined by graphics designer (Figure 1(a)). The vertices are sent to a vertex processor, in order to change size or perspective of the object, which is performed by calculating rotations and transformations of the coordinates. In this step all objects are mapped to a standardized referential axis. In the next step, a rasterizer interpolates the coordinates and defines planes that form graphics objects (Figure 1(b)). Finally, a pixel processor receives the planes from the rasterizer, calculates composed RGB colors from textures of the objects and sends this color data to frame buffer (Figure 1(c)). The color data present in the frame buffer then can be finally displayed in a screen.

Recent GPUs, have programmable vertex and pixel processors. The designers of graphics scenes or applications can make programs targeting these specific processors in other to allow applications to display those data at a many frames per second. These processors have dedicated floating point processing pipelines that can be also used for GPGPU. However, the output part of the rasterizer, is composed by non-programmable hardware, and is exclusively sent to the pixel processor and can not be accessed by the CPU. Thus, for typical GPGPU applications only the computing power of the pixel processor is available, due to its programmability, capabilities and flexibility for I/O data control.

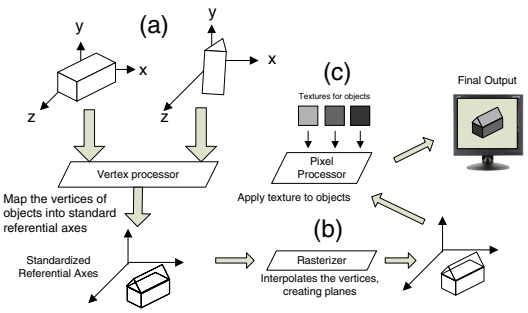

Figure 1. Processing steps for graphics rendering.

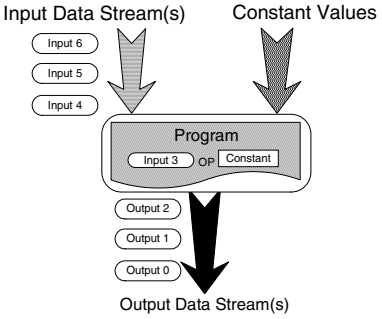

Figure 2. Structure of Flow-model.

2.2 Caravela platform

To easily use GPU resources in a general purpose computing application, the authors of this paper have developed a novel stream-based computing platform using GPUs, called *Caravela Platform* [10][11]. The Caravela platform uses the concept of *flow-model* for programming a task. An application running on the platform needs to use Caravela library for mapping the flow-model into the fragment processors of GPUs. As shown in Figure 2, the flow-model is composed of input/output data streams, constant parameter inputs and a program which processes the input data streams, generating the output data streams. The application program in Caravela is executed as a stream-based computation, just as the one of dataflow processor. However, the input data stream of a flow-model can be randomly accessed because the input data streams are mapped to two dimensional textures. On the other hand, the output data streams are sequences of data units that compose the stream. This corresponds to the frame buffer. The program needs to be written in one of the graphics shader language such as DirectX's HLSL [1] or OpenGL shader language (GLSL) [4].

The Caravela platform is mainly composed by a library that supports an API for GPGPU computing. The Caravela

Table 1. Basic functions of Caravela library

`CARAVELA_CreateMachine(...)` creates a machine structure
`CARAVELA_QueryShader(...)` queries a shader on a machine
`CARAVELA_CreateFlowModelFromFile(...)` creates a flow-model structure from XML file
`CARAVELA_GetInputData(...)` gets a buffer of an input data stream
`CARAVELA_GetOutputData(...)` gets a buffer of an output data stream
`CARAVELA_MapFlowModelIntoShader(...)` maps a flow-model to a shader
`CARAVELA_FireFlowModel(...)` executes a flow-model mapped to a shader

library has the following definitions for its processing units: *Machine* is a host machine of a video adapter, *Adapter* is a video adapter that includes one or multiple GPUs, and *Shader* is a GPU. An application needs to map a flow-model into a shader, to execute the flow-model. The shader corresponds to the fragment processors of GPUs. Table 1 shows basic Caravela functions for a flow-model execution. Using those functions, a programmer can easily implement target applications in the framework of flow-models.

When the flow-model is applied to an application that recursively feedforwards its output data to the input data at every iteration, data copy operations between CPU memory and VRAM increase, degrading the performance of the entire application. To avoid this overhead, Caravela library is extended with the *swap method* [12], which swaps pointers of the output and the input data buffers in VRAM obtained by the GPU without requiring copying the entire buffers. To support this method, Caravela library defines a data structure called *I/O pair* which has an input stream's index number and an output stream's index number of the flow-model. The I/O pair is created by `CARAVELA_CreateSwapIoPair()` function. To swap the I/O buffers, `CARAVELA_SwapFlowmodelIO()` function is called and an I/O pair is passed as an input parameter to the function. This mechanism improves the entire performance for recursive flow-model executions.

2.3 Job scheduling system

Job scheduling methods and systems are as old as multiuser/multijobs computers, and have had a long and continuous development history. The main aim of scheduling systems is to assign processes to computing resources in an effective manner. As scheduling related to CPU resources, the systems currently available can be split in the following categories;

1. Scheduling based on time
   This kind of system schedules process execution at a specified time given by a user. Well-known commands such as *at* and *cron* available in Unix systems are widely used to schedule daemons and server software actions. This system is targeted to just schedule execution of application in a timeline, and it does not have the ability to manage resources.

2. Batch execution
   In a mainframe computer, usually called supercomputer, users are not allowed to see the physical allocation of application resources such as files, because resources in mainframe can be shared by many users and the rules for resource usage are to be strictly managed by the system administrator. In order to schedule work in the mainframe, users write a script to dispatch processes to the mainframe. One of well-known job schedulers is IBM's Job Entry Subsystem 2/3 implemented on MVS. The user is required to write a batch script in the Job Control Language that is able to specify assignments between physical and logical file locations and desired computation duration. However, in a mainframe, because the assignment is targeted to CPU, the batch script specifies roughly a number of desired processors to be used by a command. After accepting the command, the scheduler assigns blocks of processors to the application.

3. Parallel and distributed resources
   Cluster computers have become popular in the last decade, where research has accelerated development of job schedulers to share cluster resources among multiple users and applications. For example, OpenPBS [6] and Condor [5] are available in the public domain, while LSF [8] is available with commercial support. Because cluster computers are typically organized and built around commodity personal computers or workstations, applications are designed to run on CPUs. Therefore, the schedulers are not aware of coprocessor resources next to CPU such as GPUs.

In summary, the conventional job schedulers introduced above mainly help to assign dispatched application to possible CPU resources, while accepting guidelines for execution at the specified priority by CPU's working time limit. However, the job schedulers do not target resources other than CPUs, which can work autonomously without any CPU's control.

In the case of GPU-based application, the CPU can become idle after it assigns application to GPUs connected next to the CPU. Then, another CPU-based application can be dispatched. If the number of applications targeted to

GPUs is larger than the one of GPU resources, resource starvation might occur, degrading the performance of executing applications.

Thus, for the case where both CPU- and GPU resources are available, applications using those resources need a novel job scheduler aware of those two resources. This paper aims to propose important parameters that affect performance in the case of a race condition between GPUs and CPUs resources, and also discusses those effects on the scheduler, also through experimental evaluations.

3 Performance Analysis for Task Scheduling

This section explains the methodology used in order to find relevant parameters that affect application performance usage of GPU and CPU resources. To investigate these parameters, this section begins by explaining resource classification in a typical machine with attached GPUs.

3.1 Classification of resources

Capable GPUs are attached to most personal computers. Therefore, this paper considers such machines with video adapter(s). Moreover, recent personal computers can have many multi CPU cores fitted into a single processor chip, allowing a larger number of processes to be assigned to the CPU. According to the discussion above, two resource categories are considered in this paper;

- Resources for CPU-based application
  A CPU-based application requires main memory's bandwidth and a CPU core. If one application performs intensive access to main memory, other applications' memory access requests might stall. On the other hand, when an application needs to share a CPU core with another application, performance can degrade due to the large number of context switching required. Therefore, those resources should be monopolized by an application until it finishes to be processed in the shortest time.

- Resources for GPU-based application
  After downloading shader programs into the GPUs, the CPU does not need to poll GPUs. While GPUs are working, they can access main memory to download texture data used as input data streams and to upload output data stream to the main memory after the shader program execution. Therefore, main memory bandwidth is required for uploading/downloading the I/O data. This can interfere with CPU-based application that also might need memory bandwidth.

When the two resources detailed above might be needed among multiple CPU- and GPU-based applications, performance degradation can occur due to competition of resource ownership. A novel job scheduler that supports CPU and GPU resources needs to eliminate this competing situation by specifying some performance factor or metrics. The factors should be related to one of the resources discussed above. Therefore, let us consider methodologies to find such factors.

3.2 Methodology for analysis

The following three effects and interference against performance must be considered to find the factors affecting performance;

1. Effect of competition related to CPU resources
   This effect was investigated over conventional job scheduling mechanisms and load balancing. There are two race conditions: One is competition for the CPU core. Another is sharing memory bandwidth between multiple applications.

2. Effect of competition related to GPU resource
   This effect is caused by races to GPU resources when multiple GPU-based applications are assigned to a given GPU. It is important to see performance effect in the race condition of GPU resources because the performance degradation is more than just the serialized execution, because there is also overhead of GPU resource management from the CPU.

3. Interference among CPU- and GPU-based applications
   There are two kinds of interferences related to these resources: computing availability and memory bandwidth. Regarding these resources, a CPU-based application might interfere with a GPU-based one, or vice versa when both applications are working concurrently.

Through experimental performance evaluations, the effects above are to be observed and the factors that cause such interferences will to be discussed. In the next section, applications used for the evaluation will be explained.

3.3 Applications used for analysis

3.3.1 1D FIR and IIR filters

FIR and IIR filters are well-known kernels applied to typical signal processing applications like image processing. These linear filters apply the following computation: $y = f(x)$:

$$FIR : y_n = \sum_{i=0}^{15} b_i * x_{n-i} \tag{1}$$

$$IIR : y_n = \sum_{i=0}^{7} b_i * x_{n-i} + \sum_{k=1}^{8} b_k * y_{n-k} \tag{2}$$

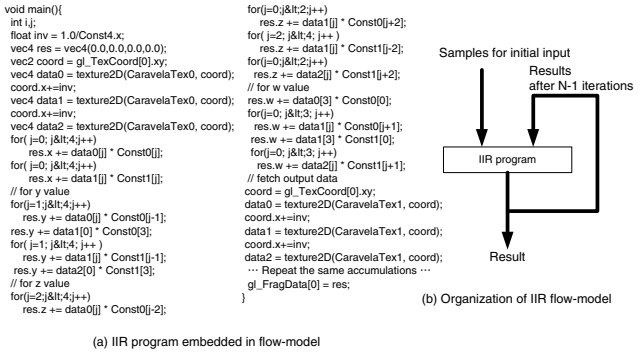

Figure 3. Flow-model for IIR filter kernel.

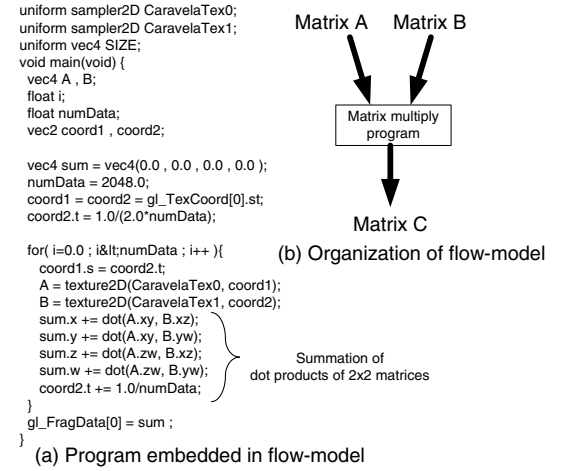

Figure 4. Algorithm and flow-model of matrix multiply.

The CPU-based versions implement these equations directly into loops. The complexity of both FIR and IIR are $O(N)$.

In case of GPU version, the filter coefficients (16 taps) are constant values available at input of flow-model. Flow-model for the FIR filter consists only of an input stream for the input signal while flow-model for IIR filter consists of two input streams; one for the input samples and another for the feedback path (y_{n-k} in eq. 2).

Programs in the flow-models were written in GLSL. The program fetches the input data stream controlling *sampler register* that obtains address information for the input data. Then, computation corresponding to equations (1) and (2) is performed.

Figure 3(a) shows the program, embedded into the IIR filter's flow-model. Organization of the flow-model is illustrated in Figure 3(b). As illustrated in the figure, this flow-model is targeted for recursive execution. Therefore, the program presents the accumulation of the results from multiply-and-add operations with coefficients provided by the constant inputs. In addition, the program uses the recursive input provided by the results after the $(N-1)$th iteration. Thus, the flow-model will provide results recursively processed by each execution by `CARAVELA_FireFlowModel()` function.

On both applications, steps from creation of flow-model to its mapping to a given shader are performed in exactly the same way. However, in IIR filter case, the feedback of output values is performed by an I/O pair, allowing reuse of the output results. In the flow-model depicted in Figure 3(b), the I/O pair is defined with the second input of the input data streams and the output data stream.

In this paper, the input signals of 8M (Mega=2^{20}) samples for both filters are given for both CPU- and GPU-based versions.

3.3.2 Matrix multiply

Matrix multiply is presented by the following equation:

$$C = AB, c_{ij} = \sum_{k=1}^{N} a_{ik} b_{kj} \quad (3)$$

This equation represents a very typical operation, which is also very expensive complexity-wise. The CPU-based version is implemented with three nested loops, and its complexity is $O(N^3)$. The goal for the GPU-based version is to have a faster implementation than the CPU.

To implement this calculation into a flow-model, we divided A and B into 2x2 matrices, and assign those four elements into a register in a shader. In this case the matrix multiply is equivalent to a set of summations of dot products calculated based on 2x2 dot products. Figure 4 shows the flow-model which has two input streams and an output stream as illustrated in (a). The flow-model embeds the GLSL code listed in (b).

In this paper, we assume that the matrices A, B and C have a dimension of 2048x2048.

3.3.3 Stream Benchmark

The stream benchmark [9] was developed to measure memory I/O performance. It intensively accesses a large memory region, tolling hugely the memory bus. Thus, this benchmark shows the effects of available memory bandwidth.

This application places an high load in the memory bus while another CPU- or GPU-based application is also working. Therefore, the effect of starvation of memory bandwidth can be seen.

Table 2. Experimental environment

PC	Opteron 2GHz, 2GB DDR SDRAM
Video adaptors	GeForce 7900@650MHz 512MB DDR3 GeForce 7300@550MHz 256MB DDR

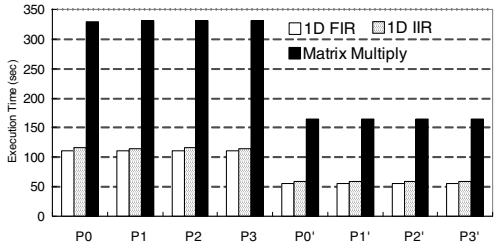

Figure 5. Performance of concurrent/serial executions of four CPU-based applications.

In this paper, the benchmark's parameter N is 32M, which means that total data transferred via main memory in the benchmark is 1.5GBytes.

4 Experimental performance analysis

Using the methods and applications explained in section 3.2 and section 3.3, this section shows experimental performance analysis.

Experiments are performed in the environment shown in Table 2. GeForce 7900 is a better GPU model. Therefore, GeForce 7300 shows lower performance. Because the graphics adapters are connected to the machines with the same hardware specification, the results related to GPU-based applications will show the effect caused by the difference of performance of the two GPUs models.

Experiments are performed in two conditions; concurrent and serial executions. The former invokes processes as background jobs simultaneously and measures execution time of each process. The latter invokes processes sequentially and measures the same time of the former one.

4.1 Analysis under competitive situation of CPU resources

We measured elapsed times of CPU-based versions of applications in two scenarios: The first scenario has four processes of the same application invoked in the PC and iterated for 10 times. Figure 5 shows the execution time of each process at concurrent execution (from P0 to P3) and serialized execution (from P0' to P3'). According to the result, when CPU resources are being competed by multiple CPU-based applications, the serial execution causes better response times to the user.

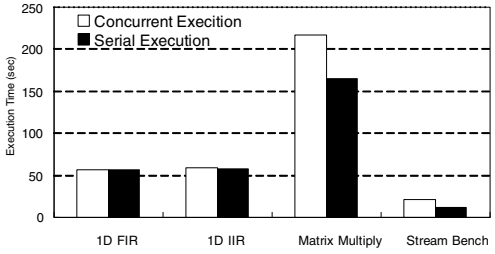

Figure 6. Performance of CPU-based applications running with/without stream benchmark.

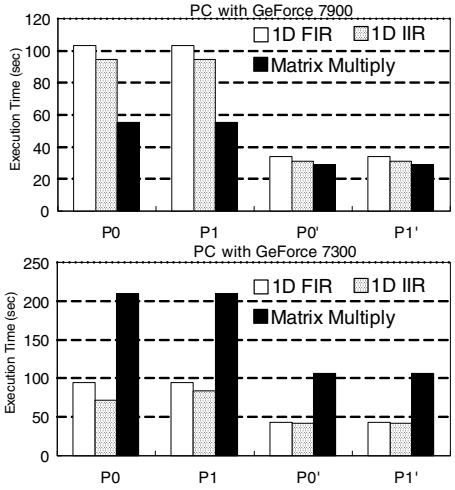

Figure 7. Performance of concurrent/serial executions of two GPU-based applications.

The second scenario has a process of CPU-based application invoked on a CPU core while the stream benchmark is running in another core of the Opteron CPU. Figure 6 shows the result of applications also iterated 10 times. The IIR and FIR filters do not show interference of the stream benchmark because those do not access large memory areas. However, the matrix multiply shows interference due to the starvation of memory bandwidth. One can think that, when two stream benchmark processes are running on two CPU cores, those elapsed times are double comparing with the one of serial execution due to the starvation of memory bandwidth.

4.2 Analysis under competitive situation of GPU resources

In this experiment, we measured elapsed time for each process of GPU-based application. Two processes were invoked in the PC. Those processes race to acquire GPU con-

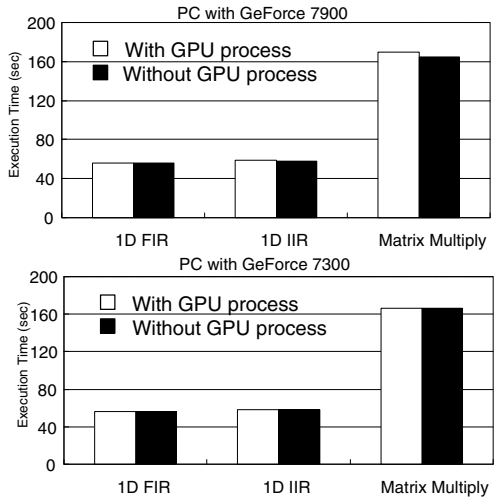

Figure 8. Performance of CPU-based application running with/without GPU-based one.

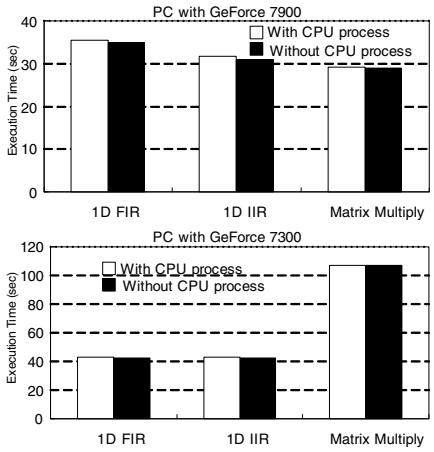

Figure 9. Performance of GPU-based application running with/without calculation intensive CPU-based one.

nected to the PC. Each application is iterated for 100 times. Figure 7 shows execution times of processes running concurrently or sequentially. The execution time of concurrent execution (P0' or P1') is almost double of the one of sequential execution (P0 or P1). Therefore, when two processes are running with sharing a single GPU, the execution is serialized. Thus, the task execution latency becomes large.

4.3 Analysis of interference among CPU- and GPU-based applications

We assumed two patterns of interference between CPU- and GPU-based applications. One is interference with CPU-based application while a GPU-based one is running. The other is the opposite situation. Figure 8 shows execution times for the applications invoked on CPU and iterated for 10 times. While the execution of the CPU-based version, GPU is occupied by the same application of GPU version. On the other hand, Figure 9 shows execution times of the applications invoked on GPU iterated for 40 times for the FIR/IIR filters and 50 times for the matrix multiply. CPU is used by the same application of CPU version that implements directly the equations illustrated in section 3.3 during the execution of the GPU version. On both results, we confirmed that there is no interference between CPU and GPU resources because the execution times are almost equal among with and without GPU/CPU processes.

We also focused in interference due to memory bandwidth applying the same experiment patterns as performed above. Figure 10 shows execution times of the stream benchmark with/without invoking one of the applications targeted to GPU. The result shows interference in memory accesses when invoking GPU-based application. Because the execution time of the FIR filter is small and almost all the execution time is just data transfer from/to main memory, the interference due to memory accesses from the FIR filter shows a large effect against the stream benchmark. In the opposite case, shown in Figure 11, execution time of the FIR filter of GPU-based application represents the same interference. Therefore, we confirmed that starvation of memory bandwidth affects the performances of both CPU- and GPU-based applications.

4.4 Summary of results

Given the results presented above, let us conclude by introducing interfaces for controlling execution priorities of CPU- and GPU-based applications in a job scheduler that is aware of both CPU and GPU resources.

From the results in section 4.2, we have confirmed that performance degrades when multiple GPU-based applications are running under race condition regarding GPU resources. Therefore, the job scheduler needs to only invoke the number of processes equals to the one of GPUs (i.e. the number of video adapters).

The results in section 4.3 illustrate that CPU and GPU resources are completely separated devices regarding computational capacity. Therefore, the job scheduler can invoke applications running on both CPU and GPU resources. However, if an application targeted to either CPU or GPU needs memory bandwidth, we have confirmed that application invoked in one resource will interfere with another according to the result on section 4.3. In this case, the job scheduler should provide an interface to accept hints regard-

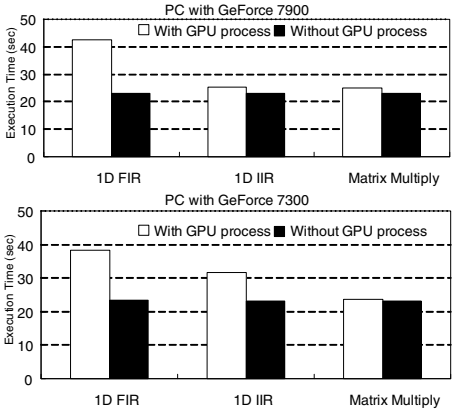

Figure 10. Performance of Stream benchmark running with/without GPU-based one.

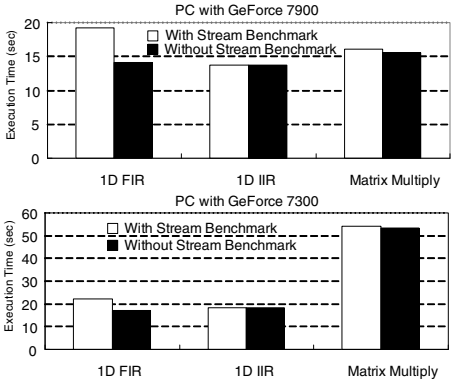

Figure 11. Performance of GPU-based application running with/without stream benchmark.

ing application's characteristics and how intensive memory accesses are expected.

5 Conclusions

This paper presented the results regarding investigation of factors and parameters that limit job scheduling in a novel job scheduler that supports and is aware of both CPU and GPU resources.

According to experimental performance analysis, we can conclude that the job scheduler needs to provide interfaces to accept the following two parameters: one that specifies where applications should be assigned between CPU and GPU, and another that should specify how much memory bandwidth an application needs. Analyzing these two parameters, we conclude that the best way for the scheduler is to keep the following rules at application's dispatch timing: 1) the scheduler needs to have two separate scheduling queues: one for applications using CPU resources and another for applications using GPU resources. 2) A larger number of CPU-based applications than the one of CPU cores are not dispatched. 3) A larger number of GPU-based applications than the one of GPUs are not dispatched. Finally, 4) a parameter to indicate how intensive the applications access to main memory must be given.

As future work, we are planning to implement the scheduler by applying the parameters and the rules presented above, and then will perform its dynamic performance evaluation.

6 Acknowledgment

This work is partially supported by the Portuguese Foundation for Science and Technology (FCT) through the FEDER program.

References

[1] DirectX homepage. http://www.microsoft.com/directx.

[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 Trans. Graph.*, 23(3):777–786, 2004.

[3] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU Cluster for High Performance Computing. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, page 47. IEEE Computer Society, 2004.

[4] J. Kessenich, D. Baldwin, and R. Rost. The OpenGL shading language. *3Dlabs, Inc. Ltd.*, 2006.

[5] M. Litzkow, M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[6] OpenPBS. http://www.openpbs.org/.

[7] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, Aug. 2005.

[8] Platform Computing Corporation. http://www.platform.com/.

[9] Stream Benchmark. http://www.streambench.org/.

[10] S. Yamagiwa and L. Sousa. Caravela: A novel stream-based distributed computing environment. *IEEE Computer Magazine*, 40(5):70–77, 2007.

[11] S. Yamagiwa and L. Sousa. Design and implementation of a stream-based distributed computing platform using graphics processing units. In *ACM International Conference on Computing Frontiers*, pages 197–204, May 2007.

[12] S. Yamagiwa, L. Sousa, and D. Antao. Data buffering optimization methods toward a uniform programming interface for gpu-based applications. In *Proceedings of the 4th international conference on Computing frontiers*, pages 205–212. ACM Press, 2007.