Data Buffering Optimization Methods toward a Uniform Programming Interface for GPU-based Applications

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

Leonel Sousa
INESC-ID/IST
Rua Alves Redol, 9, 1000-029
Lisboa Portugal
las@inesc-id.pt

Diogo Antão
INESC-ID/IST
Rua Alves Redol, 9, 1000-029
Lisboa Portugal
antao@sips.inesc-id.pt

ABSTRACT

The massive computational power available in off-the shelf Graphics Processing Units (GPUs) can pave the way for its usage in general purpose applications. Current interfaces to program GPU operation are still oriented towards graphics processing. This paper is focused in disparities on those programming interfaces and proposes an extension to of the recently developed Caravela library that supports stream-based computation. This extension implements effective methods to counterbalance the disparities and differences in graphics runtime environments. Experimental results show that these methods improve performance of GPU-based applications by more than 50% and demonstrate that the proposed extended interface can be an effective solution for general purpose programming on GPUs.

Categories and Subject Descriptors

D.1.3 [**Software**]: PROGRAMMING TECHNIQUES—*Concurrent Programming, Parallel programming*

General Terms

Performance

Keywords

Graphics processing unit, DirectX, OpenGL, general purpose processing

1. INTRODUCTION

The need for realistic graphics representations, especially in the entertainment market, has promoted GPU performance growth by leaps and bounds in the recent years. This growth-rate exceed the ratio defined by the Moore's law [13]. Since GPUs have become commodity components in almost all personal computers, instead of only being present in high-performance computing systems, researchers have been focusing their attention on GPU's potential computational performance. Therefore, GPUs are being regarded as new high performance computing available platforms, with the ability of speeding up processing and freeing Central Processing Units (CPU) for other tasks.

Contemporary GPUs are programmable, which allows their programming for general purpose applications. However, the burden of efficiency rests upon the users/programmers knowledge of graphics processing details in order to utilise the GPU resource efficiently. For example, the calculations on GPU are performed by rendering to a screen buffer. This mechanism is responsible for the main difference in the memory interface and in control of the GPUs, when comparing with CPU. Therefore, it is up to the programmer to go back and forth, between the controlling code for GPU and the one corresponding algorithm for the target application. Thus, to let the programmer focus on the algorithm without the need to known the details of the graphics runtime environment, we need to implement a uniform interface for GPU-based applications.

Although GPU hardware implementation differs among vendors, a common processing pipeline is provided, which is composed by a vertex and a pixel processor. The user interface for GPU is specified as a graphics application programming interface running on the host CPU. The most well-known interface software for *graphics runtimes* are the DirectX9 [3] and the OpenGL 2.0 [12]. However, those runtimes have different functionalities and different capabilities for graphics applications. Therefore, to design a uniform interface for GPU-based applications its required to address those interface differences, without degrading performance.

This paper discusses the main differences in capabilities and performance of DirectX and OpenGL graphics runtimes. In addition, a novel and uniform programming interface is proposed, which has been implemented as an extension to the *Caravela library* [1]. The Caravela library provides an API for stream-based computing using GPU resources developed by the authors of this paper. We also present the implementation details of an extension to the Caravela library that provides a common programming interface without degrading performance. Moreover, experimental performance evaluations are performed in order to confirm that the proposed extension is an effective and efficient way to program general purpose applications on GPUs.

This paper is organised as follows: The following section describes features of GPUs and details of its programming. In section 3, graphics runtime environments are compared and a novel interface extended in the Caravela library is proposed. The performance of the Caravela library and the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CF'07, May 7–9, 2007, Ischia, Italy.
Copyright 2007 ACM 978-1-59593-683-7/07/0005 ...$5.00.

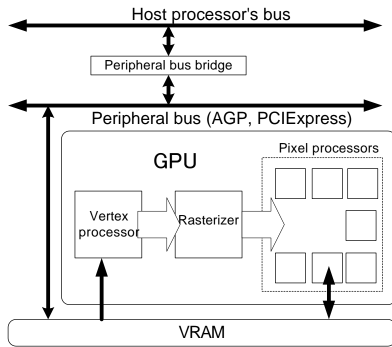


Figure 1: Organization around GPU.

proposed extension are experimentally evaluated and the results are presented in section 4. Finally, we will conclude this paper in section 5.

2. GENERAL PURPOSE PROCESSING ON GPUS

In today's computers, GPU can act as a coprocessor of the CPU via a peripheral bus, such as the AGP or the PCI-Express buses, as shown in Figure 1. CPU uses a VRAM (Video RAM) through a dedicated connection to read/write the processed graphics objects. To process graphics objects, the CPU sends object data to the VRAM and program code to the GPU.

2.1 GPU hardware and programming characteristics

Figure 2 shows the required processing steps on a GPU in order to create a graphical image in a frame buffer to be displayed in a screen. First, graphics data is prepared as a set of normalized vertices of objects on a referential axis defined by graphics designer (Figure 2 (a)). The vertices will be sent to a vertex processor to change size and/or perspective of the object, by calculating rotations and transformation of the coordinates. In this step, all the objects will be mapped to a standardized referential axis. In the next step, a rasterizer interpolates the coordinates and defines planes that form the graphic objects (Figure 2 (b)). Finally, a pixel processor receives these planes from the rasterizer and creates color data in the frame buffer by calculating composed RGB colors from textures (Figure 2 (c)). Each color data is written into the frame buffer, and thus the buffer will be output to the screen. The current GPUs, which have multiple pixel processors as shown in Figure 1, can process this color data concurrently for different parts of a screen.

In actual GPUs, vertex and pixel processors are programmable, and are composed of fast and dedicated floating point pipelined units, mainly programmed for graphics applications. The rasterizer is composed of fixed hardware and its output data can not be fetched by CPU, so the CPU usually uses only the computing power of the pixel processors.

The idea of using GPUs for replacing CPUs not only for graphical tasks originated a new research area designated by General-purpose Processing on Graphics Processing Units (GPGPU) [9] [11]. GPGPU applications control GPU hardware using a graphics runtime environment. These graphics runtime components operate in the CPU domain, in order to

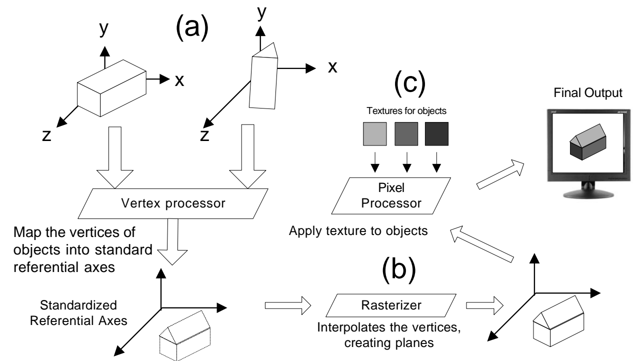


Figure 2: Graphics processing steps.

control the GPU. Because these runtime environments have been firstly designed for graphics processing, its interfaces are not familiar to scientific users/programmers in general. For example, when an application wants to output NxN matrix, the programmer needs to setup a frame buffer and he must understand that the output will be generated to an NxN pixel plane. Although this is the correct operation for a graphics applications it does not make sense for GPGPU applications. Thus, it is important to define a new Application Programming Interface (API) for GPGPU that hides the graphical legacy of GPU.

To solve the problem of disparities between graphics runtime environments and general purpose processing requirements and characteristics, some solutions have been proposed, such as Sh [4], Scout [10] and Brook [7]. Sh is a graphics processing interface with an object oriented interface for C++. The program for pixel processor is written in Sh language and Sh covers the difficult control of graphics runtime environments. Scout is another wrapper for graphics which uses a language based on C*. Although details of graphics runtime are hidden by these two systems, they are still targeted for visual applications. Therefore, the programmer can not completely eliminate graphical dependent environments or issues. Brook is a compiler-oriented interface for GPU-based applications for which programmer just needs to identify functions to be transposed to programs on a pixel processor (this program is called *pixel shader*) with a special keyword (*kernel*). Although this interface seems to be one of the best solutions for the problem mentioned above, it is hard in practice to tune the achieved performance, namely regarding memory access such as buffers' management.

Therefore, we need to address the problem of disparity between the programming interface and the graphics environment, and keep focused in its two main aspects: the level of complexity needed to program general purpose applications on GPUs and the achieved performance.

2.2 Caravela platform

Caravela platform [1] is an interface for stream-based computing implemented by the authors of this paper. The Caravela platform uses the concept of *flow-model* for programming a given task. Applications on the platform use Caravela library for mapping the flow-model into processing units.

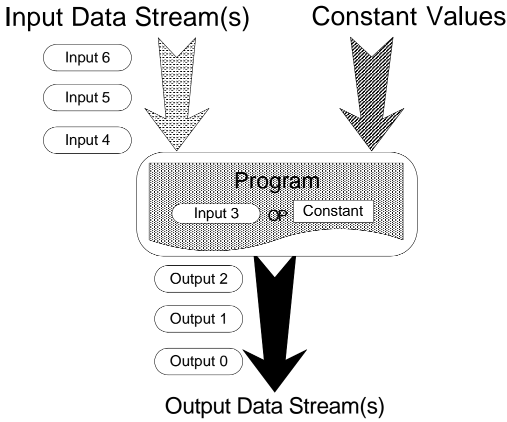


Figure 3: Structure of the flow-model.

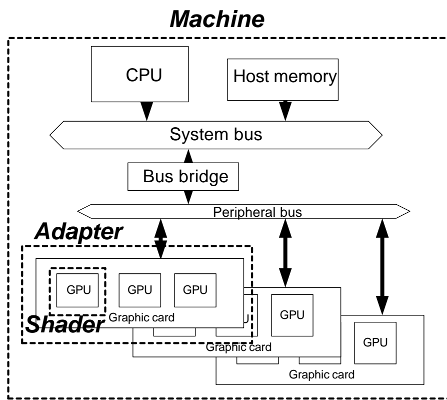


Figure 4: Resource hierarchy in the Caravela library.

2.2.1 *Representing application according to the flow-model*

As shown in Figure 3, the flow-model is composed of input/output data streams, constant parameter inputs and a program which processes the input data streams and generates the output data streams. The application program in Caravela is executed as a stream-based computation, such as the one of data-flow processor. However, the input data stream of a flow-model can be randomly accessed because the input data streams are memory buffers for the program that use this data. On the other hand, the output data streams are sequences of data units that compose the stream. Thus, the execution of the program embedded in the flow-model is not able to touch any other resources than the I/O data streams, which is an advantage in a security perspective.

The flow-model also has another advantage for distributed environments. All the methods to execute a task are encapsulated into a data structure. Therefore, the flow-model can be managed as a distributed object, which can be easily fetched anywhere based on the Caravela runtime environment. For example, when a flow-model is placed in a remote machine, it can be fetched by any other machines and the execution mechanism from the flow-model be reproduced.

The processing unit to be assigned to a flow-model program can be a software-based emulator, an hardware data flow processor, an hardware dedicated processor, etc. . Herein, in the scope of this paper, we are interested in GPU as the processing unit. The processing style of GPU, which reads texture inputs by stream and generates the output data as a stream, perfectly fits into the flow-model framework.

Table 1: Basic functions of Caravela library.

Function
`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 flowmodel to a shader.
`CARAVELA_FireFlowModel(...)` executes a flowmodel mapped to a shader.

2.2.2 *Caravela library*

The Caravela platform is mainly composed by a library that supports an API for GPGPU. The Caravela library has been adopted to the definitions for the processing units represented in Figure 4: *Machine* is a host machine of a video adapter, *Adapter* is a video adapter that includes one or multiple GPUs and finally *Shader* is a GPU. An application needs to map a flow-model into a shader, to execute the mapped flow-model.

Table 1 shows the basic Caravela functions for a flow-model execution. Using those functions, a programmer can easily implement target applications in the framework of flow-models, by just having to map flow-models into shader(s). Therefore, the programmer does not need to know about graphics runtime environment details, which means that Caravela library can become one of the solutions to relieve the chief problem of differences between graphical environments mentioned in the previous section.

However, to efficiently support multiple graphics runtime environments in the lower layer of Caravela library, care should be taken to understand their differences in methods and features. If a graphics runtime environment, which a programmer aims to instantiate from the Caravela library, has special performance tuning functions, the programmer would like to be sure that the interface provided by Caravela library takes advantage of that. Therefore, we need to consider extended functions for a uniform interface for GPGPU applications that tunes-up runtime procedures in order to achieve the best performance in different graphics runtime environments.

3. UNIFORM PROGRAMMING INTERFACE FOR GPU-BASED APPLICATIONS

First of all, let us consider the possible differences between DirectX9 and OpenGL 2.0, herein just designated DirectX and OpenGL respectively.

3.1 Functional comparison between runtimes

When a uniform interface is designed, one of the most important issues is compatibility of graphics runtime environments over different Operating Systems (OSs). The DirectX is embedded into Windows. Therefore, its functionality is supported trustfully as long as we are using Windows. However, DirectX is not compatible with other OSs such as Linux[1]. Therefore, the implementation of the uniform interface using the DirectX is relevant only to the Windows OSs.

On the other hand, OpenGL is ported to almost all available OS in the market. For Windows, the basic functions are available in the *opengl32* dynamic library. However, an application using vertex or pixel processor needs the functions of OpenGL Extention Specification implemented by GLEW (The OpenGL Extension Wrangler Library) [5] for instance, which provides an interface to create program object for the processors.

3.1.1 GPU resource management

When multiple GPUs are connected to a machine, the GPGPU applications may want to use those processing units concurrently. The uniform interface should individually manage these GPUs if they are available on the graphics runtime environments.

DirectX is able to manage multiple video adapters separately. It can allow resources, such as input textures, to be allocated separately. Therefore, the uniform interface must support the functionality of concurrently executing GPU programs. Contrarily, OpenGL uses a default video adapter via GLUT [6]. This interface does not allow to specify which video adapter is used. Thus, the uniform interface allows both situations but automatically switches to a single program mode at any time in a machine if OpenGL is being used.

3.1.2 Shader language and Compiler

GPU can execute a dedicated program written in a shader program language. The graphics runtime environments can accept high-level shader language or assembly language. The differences appear at the translation level of the shader code.

DirectX accepts both the assembly language and the high level language. The shader code written in the assembly language must follow the GPU model specified by Microsoft called *Shader Model*. The shader program begins with the shader version instruction, such as `ps_2_0` for Pixel Shader Model 2.0. The assembly-based shader code is assembled by `D3DXAssembleShader` function. On the other hand, the HLSL (High Level Shader Language) is available, which is a C language like interface for the shader programming. The compilation is performed by the `D3DXCompileShader` function. This function takes one of the shader model versions available on the target GPU. The shader version supported by a target GPU is dependent on the GPU architecture. For example, nVIDIA's GeForce7 supports Shader Model 3.0. However, Geforce6 only supports the version 2.0. This means that even if a shader program is written in HLSL, loop statements are unrolled in GeForce6, because it can not use the loop instruction supported by the Shader Model 3.0.

[1]Although the DirectX9 for WINE[2] emulates DirectX runtime, Linux itself does not have any native runtime for DirectX.

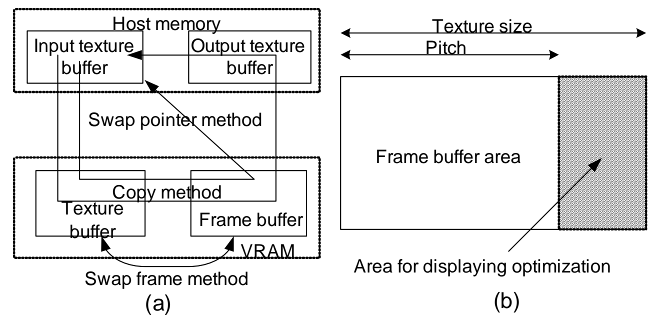


Figure 5: Buffer management for graphics: (a) data displacement on memory; (b) organization of the frame buffers.

DirectX provides a software emulator that supports the Shader Model 3.0 at runtime. This emulator is originally used for the reference design for driver developers. However, normal user can use this emulator as a substitute of a programmable GPU hardware. Thus, it is possible to check the shader code behavior even without GPU hardware.

OpenGL supports GLSL (OpenGL Shader Language) [8] to write shader programs. The compilation of a shader program is performed by `glCompileShaderARB` function, which will use the compiler functionality on video driver. Therefore, the syntax checking, restrictions and optimizations are dependent on the driver vendors, even though the basic syntax follows GLSL specification.

In OpenGL, global variable `GLEW_ARB_fragment_shader` indicates if the GPU is programmable. Because OpenGL does not provide any software emulator, it is impossible to check the behavior of the shader unless the programmable GPU hardware is available.

3.1.3 Buffers' sizes and memory management for graphics

Maximum sizes of input textures and output frame buffers are a constraint that defines the maximum problem size. Since the Video RAM (VRAM resource) connected to GPU is a finite resource, sizes of texture and frame buffer are defined by the graphics runtime environments. DirectX limits the sizes of texture and frame buffer to the maximum display size, such as 1024x768. OpenGL allows the maximum possible size to be independent of screen size, being possible to create 4096x4096 pixel textures with the recent GPU hardware.

The graphics runtime environments provide functions to allocate buffers for input textures and frame buffers. The buffers are allocated on host memory and on VRAM as shown in Figure 5(a). At the execution of a shader program, the input texture buffer allocated on the host memory is copied to the one allocated on the VRAM. Frame output buffers are allocated in the same way and data is copied back from the VRAM to the host memory.

With DirectX, input texture buffer and frame buffer (called *render target*) are created in runtime, namely through the `CreateTexture` method –*Usage* argument specifies the type of the texture, 0 for the input texture and D3DUSAGE_RENDERTARGET for a frame buffer, and have a special memory shape as shown in Figure 5(b). It includes an optimization area for displaying, where application is inhibited to

touch. Therefore, to initialize input data or to get output data, GPGPU application has to write/read the data considering the *pitch* length. This mechanism prevents dynamically exchange of a texture defined as an input texture with another defined as a frame buffer, because the buffer properties can only be modified by the `CreateTexture` method. Therefore, the data of input texture and the one of frame buffer must be copied to/from the VRAM when the GPGPU application needs to exchange data located in those buffers.

The situation with OpenGL is different, because it manages input textures and frame buffers by defining them just as "textures" which are linked to the target buffers. For example, when a frame buffer is allocated in VRAM by the `glTexImage2D` function, it will be attached to the frame buffer by the `glFramebufferTexture2DEXT` function. Applications can change the attachment of the input texture and output frame buffers by using the `glActiveTexture` and the `glBindTexture` functions. This means OpenGL is able to dynamically control the pointers to the input texture or the frame buffers in the GPU. Moreover, the data buffer for texture can be provided as a buffer dynamically allocated by application itself, which means that the buffer that holds data to be copied to VRAM is not allocated by the graphics runtime. For example, the `glTexSubImage2D` function copies the texture data to VRAM and the `glReadBuffer` function copies the texture data from VRAM. These functions can accept a buffer pointer dynamically allocated on host memory by application. Thus, applications have only to pass the pointer from the frame buffer to the input texture to feedback data for further calculations on GPU.

In summary, the static management for the GPU resources is similar between DirectX and OpenGL, except for the differences of the language types: the assembly or the high-level language. Therefore, the interface for the static functionalities is easily defined. However, management of VRAM buffers significantly differs with runtime environments. The data I/O operation between host memory and VRAM is a key operation to achieve high performance. It can be concluded that the uniform interface must have a capability for tuning buffering mechanism for data I/O. For example, considering a recursive application, which has to read generated output data as input data for subsequently computation, is possibility to optimize the feedback buffering mechanism. According to the mechanisms of data exchange found on the runtime environments referred above, let us categorize the possible methods to implement the feedback (see Figure 5(a)):

1. **copy method** - this method copies frame buffer data in VRAM to host memory, then copy this data to input texture buffer on the host memory and finally copy it to texture buffer back in the VRAM; both runtimes provide functions to implement this method;

2. **Swap pointer method**- this method just exchanges pointers for input texture and output texture buffers on host side; only OpenGL provides this method by passing the pointer from `glReadBuffer` function to `glTexSubImage2D` function;

3. **Swap frame method**- this is the most efficient method, that swaps output buffer and input texture buffer on GPU side; only OpenGL also provides functions to implement this method.

All these different copy methods have to be considered to design the uniform interface for GPU-based applications, in order to fully exploit the GPU's potential performance. Moreover, from the user perspective the interface must be unique but it has to implicitly perform the best in any machine and environment. In the next section, we show an extended interface in Caravela which hides the differences between runtimes and achieves high performance results.

3.2 Hiding runtime differences in Caravela library

3.2.1 Static functionality: resource management

Applications using Caravela library will try to acquire a shader for executing a flow-model. The runtime is specified by the argument of `CARAVELA_Initialize()` at the beginning of the application. Then the application will call `CARAVELA_CreateMachine()` to create a definition of host machine. Using the machine structure, it calls `CARAVELA_QueryShader()` trying to find a shader in the machine. Conditions to find the shader, such as the shader model version for DirectX, suitable for the flow-model created by `CARAVELA_CreateFlowmodelFromFile()` will be passed as the argument. If the conditions match, `CARAVELA_MapFlowmodelIntoShader()` is called to map the flow-model to the shader. This function creates input textures and frame buffers, compiles a program in the flow-model and downloads it into the Shader, by using one of the graphics runtime environments. `CARAVELA_GetInputData()` will create a buffer in host memory and return the buffer. Considering the compatibility for DirectX9, the returned buffer includes a value for the pitch length. The initialization of buffer will be performed by an access macro named `GETFLOAT32_1D` or `GETFLOAT32_2D` that will calculate an offset to the desired data. After the initialization of the input data, `CARAVELA_FireFlowModel()` will be called to execute the flow-model, and will generate output data from the flow-model. This function creates the output buffers on VRAM implicitly. Finally, `CARAVELA_GetOutputData()` copies the data from VRAM to a buffer on host memory, and returns the pointer to this buffer to the application. This buffer will be also accessed by the `GETFLOAT32_1D` or `GETFLOAT32_2D` macro. Considering the operations above, Caravela library is able to provide the same functionality both for the DirectX and the OpenGL.

3.2.2 Dynamic capabilities: buffer management

The execution steps described in the previous subsection cover only non-recursive applications, which means the outputs are never used for further processing. According to the section 3.1.3, the buffer management can be optimized in OpenGL, leading to the proposed extensions for the Caravela library.

In the Caravela flow-model, a recursive I/O will be presented as a connection from an output data stream to an input data stream. We define 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. In the uniform programming interface, the I/O pair is created by `CARAVELA_CreateSwapIoPair()` function. To swap the I/O streams, `CARAVELA_SwapFlowmodelIO()` function is called and an I/O pair is passed as an input parameter to the function.

The implementation of `CARAVELA_SwapFlowmodelIO()` differs from DirectX to OpenGL due to the availability of dif-

ferent buffer management mechanisms. The DirectX version performs the copy method presented above while the OpenGL version implements the swap pointer method or the swap frame method. The implementation of the swap pointer method is straightforward, by passing a pointer from the `glReadBuffer` function to the `glTexSubImage2D` function, thus avoiding copy operations, such as `memcpy()`. The implementation of the swap frame method is more tricky. OpenGL allows only a frame buffer that is already assigned to be used as the input texture buffer on GPU. Therefore, before exchanging the I/O pair, the input buffer must be considered as an additional frame buffer by using the `glFramebufferTexture2DEXT` function. After assigning the frame buffer, the I/O pair will be exchanged at every call of `CARAVELA_SwapFlowmodelIO()`. Performance tuning is performed by adopting the fastest method, which is experimentally identified in the next section.

Summarising, applications with a recursive algorithm initialises an input buffer of a flow-model registered to an I/O pair at the first iteration. After the execution of the flow-model, `CARAVELA_SwapFlowmodelIO()` is called to exchange the data of the I/O pair. After the second iteration, the application does not need to use `CARAVELA_GetInputData()`. Therefore, with reduction of data copy operations it is expected an improvement in processing ability.

4. EXPERIMENTAL EVALUATION

To evaluate performance of the Caravela library extensions, this section shows an experimental results using two machines which characteristics are presented in Table 2. Three experimental evaluations are considered in this section: (1) analysis of copy operation in GPU-based application, (2) performance of different buffering methods (3) overall performance analysis for real applications. For these evaluations, FIR (Finite Impulse Response) filters and IIR (Infinite Impulse Response) filters, that performs recursive computing, are considered. These are well known linear systems $y = f(x)$:

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

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

In both cases the filter coefficients (16 taps) are constant values available at the input of the flow-model. Therefore, the flow-model for FIR filtering consists only of an input stream for the input signal while the flow-model for IIR filtering consists of two input streams, one for the input samples and another for the feedback path.

The programs of the flow-model are written in the HLSL, and compiled to the Pixel Shader Model 3.0 profile of DirectX and in GLSL for OpenGL. The program fetches the input data stream controlling the sampler register that obtains the address information for the input texture. Then, the computation corresponding to equations (1) and (2) are performed. Note that a register (valuables for GLSL) obtains four single precision floating point values. Therefore, each output from the program includes four results.

On both applications, processes from the creation of the flow-model to its mapping to a shader are performed in exactly the same way. However, for the IIR filter case, the feedback of output values is performed by an I/O pair for reusing the output results.

Table 2: Environment for experiments

	Machine1	Machine2
Chipset	nForce4 Ultra	945GM Express
CPU	AMD Opteron 170@2GHz	Intel CoreDuo T2300@1.66GHz
Main memory	2x1GB DDR 400	2x512MB DDR2 533
Graphics board	MSI NX7300GS 256MB DDR	nVIDIA GeForce Go 7400 128MB DDR2
Display size	1280x1024	1280x800
OS	WindowsXP Pro	WindowsXP Home

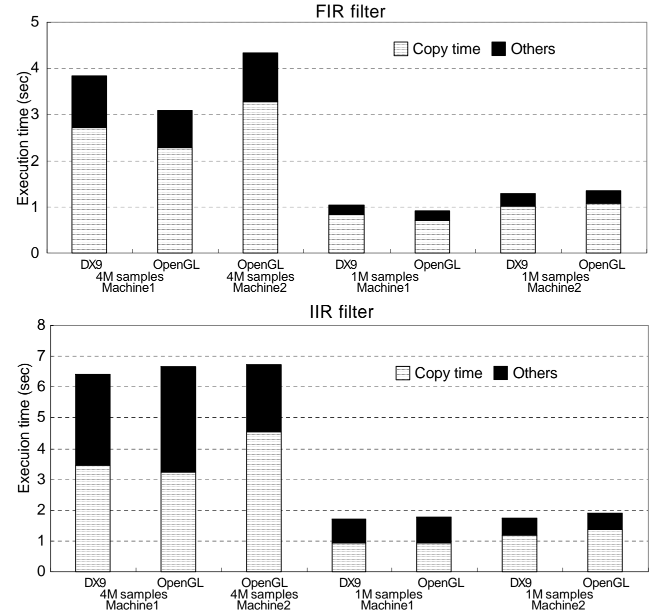


Figure 6: Overhead analysis for copy operation.

Evaluation results will be shown as an execution time of 30 iterations per application, varying the input signals between 1M(Mega=2^{20}), 4M and 16M samples.

4.1 Overhead analysis for copy operation

Before analyzing performance of buffering methods, it is important to know how much time the copy operations between host memory and VRAM take. This evaluation shows ratios of the copy time in the FIR and the IIR filter applications executed on DirectX and OpenGL. Therefore, the Caravela library with the copy method is applied to the recursive feedback operation. Due to the texture size restriction by screen size of DirectX, 4M samples can not be input to filters in Machine2.

Figure 6 shows the total execution times on Machine1 and Machine2, isolating the time for copy operations (shown as Copy time) from the remaining time. The remaining time includes the time for copy operation between buffers allocated on host memory, calculation time on GPU and setup time for GPU execution. The execution times on both runtime environments are similar and the time for copy operation between host memory and VRAM takes from 70% to 80%. Because the FIR filter does not need to feedback the output

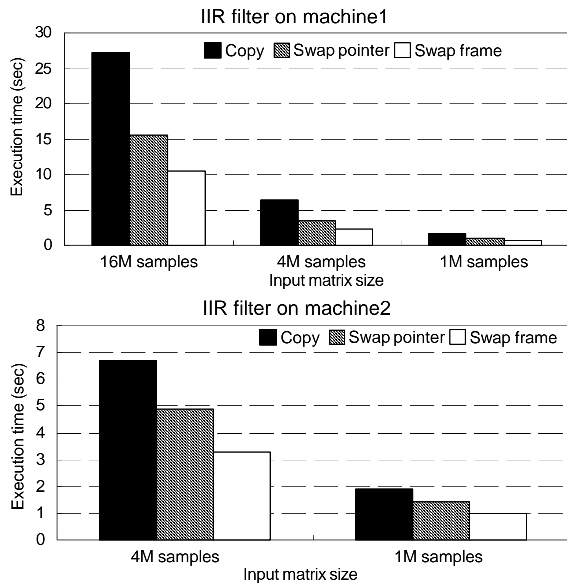
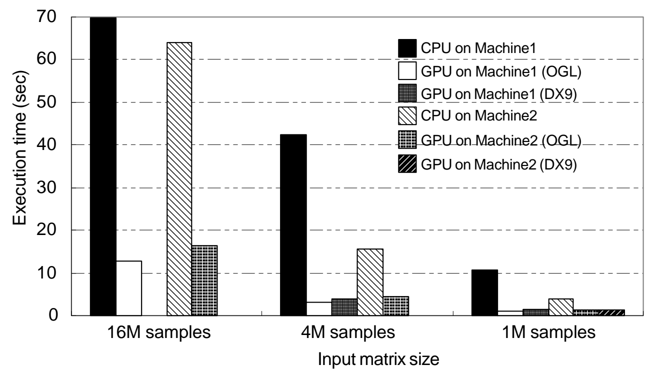


Figure 7: Buffering method comparison.



FIR filter (# of samples)	16M	4M	1M
CPU time(s) on machine1	169.69	42.31	10.58
GPU time(s) on machine1 (OGL)	12.72	3.09	0.92
GPU time(s) on machine1 (DX9)	NA	3.83	1.47
CPU time(s) on machine2	64.02	15.73	3.81
GPU time(s) on machine2 (OGL)	16.30	4.33	1.34
GPU time(s) on machine2 (DX9)	NA	NA	1.28

Figure 8: Overall performance comparison of GPU versus CPU performance to FIR filtering (NA: Not Available).

data to the input in host memory, the time ratio for copy operation shows larger than the one of the IIR filter. The IIR filter requires additional copy operations on host memory to move the result of calculation at every iteration from the output buffer to the input buffer. Although the copy time ratio is smaller than the one of the FIR filter, note that the copy time among the buffers allocated in the host memory is included in the remaining time (i.e. "Others" in the figure).

According to the result shown in Figure 6, we can conclude that it is essential to optimize the buffering operation in order to reduce the time for copy operations on GPGPU applications.

4.2 Buffering method comparison

This evaluation shows a performance comparison between the copy method, the swap pointer method and the swap frame method for recursive application, namely the IIR filter. The swap pointer and the swap frame methods are implemented on the `CARAVELA_SwapFlowmodelIO()` function. The setup with 16M samples is not available on the Machine2 because it shares its host memory with the GPU for rendering operations, and thus it does not have enough memory to execute it under the condition without swapping the content of the host memory to its Hard Disk.

Figure 7 shows the elapsed time of the IIR filter using the different methods, varying the number of input samples. The swap pointer method shows better performance than the copy method because the copy operations performed on the host memory are reduced. A performance improvement of 20% to 40% is achieved with the swap pointer method regarding to the copy method. Although this improvement is caused by the removal of the copy operations on the host memory, this is a real performance improvement. Moreover, a performance improvement about 30% is achieved with the swap frame method regarding to the swap pointer method. Therefore, the total performance by the swap frame method is an improvement about 55% to 60%. This is a remarkable performance improvement which has a profound impact in the processing time of recursive applications in GPUs.

4.3 Overall performance comparison

This section compares the performance of graphics runtime environments with CPU-based approaches for filtering application. For DirectX, the copy method, the only available, is applied in this run. For OpenGL, the swap frame method is applied, in order to obtain the best performance.

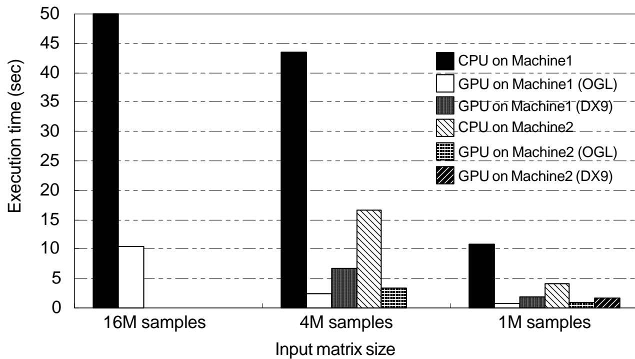
Figures 8 and 9 compare the GPU-based implementation with the CPU-based one. The 16M samples with DirectX runtime is not available on the Machine1 due to the screen size limitation. In addition, the CPU-based implementation and the OpenGL one for the IIR filter on Machine2 are not presented because the starvation of the memory implies a lot of swapping to Hard Disk.

Regarding FIR filter, the DirectX shows a little bit more overhead comparing to OpenGL and the performance of GPU-based implementation, using Caravela library, is 7 to 13 times and 3 to 4 times faster than the CPU-based implementation on the Machine1 and the Machine2, respectively. Regarding IIR filtering, the DirectX shows worse performance than OpenGL due to the overhead caused by the copy method. Considering the performance on both runtime environments, the GPU-based implementation for the IIR filter using Caravela library is 6 to 18 times and 2 to 5 times faster than the CPU-based implementation on the Machine1 and the Machine2, respectively.

4.4 Considerations about the results

According to the experimental results presented above, we can underline the following three most relevant issues in order to improve performance of GPGPU applications.

Copy operations on GPGPU applications are very frequent, due to the data migration from the host memory to the VRAM and vice-versa. Therefore, reducing the num-



IIR filter (# of samples)	16M	4M	1M
CPU time(s) on machine1	173.91	45.53	10.86
GPU time(s) on machine1 (OGL)	10.45	2.36	0.72
GPU time(s) on machine1 (DX9)	NA	6.67	1.78
CPU time(s) on machine2	NA	16.59	4.05
GPU time(s) on machine2 (OGL)	NA	3.28	0.99
GPU time(s) on machine2 (DX9)	NA	NA	1.72

Figure 9: Overall performance comparison of GPU versus CPU performance to IIR filtering (NA: Not Available).

ber of copy operations is important to improve performance of GPU-based processing systems. Another issue is related to the efficiency of shader program on GPU, since the copy operations in the GPGPU applications cause a large overhead. Therefore, a huge amount of calculations should be embedded to a shader program. Finally, the data buffering methods in GPGPU application are very important in order to improve the overall performance of GPU-based applications. When a uniform interface for GPU-based application is designed, the interface should include the functions to optimize data buffering. Particularly, for recursive processing applications, the swap frame method has to be applied.

The Caravela library prepares functions to control the I/O pair. Using these functions, implicit use of the best method for data buffering is guaranteed for different graphics runtime environments. Thus, the uniform interface for GPGPU applications, with flexibility for performance tuning, has been implemented as an effective extension to the Caravela library.

5. CONCLUSIONS

This paper proposed a uniform interface, that hides differences in graphics runtime environments and allows programmer of GPGPU to concentrate on the programming of algorithms instead of wasting time with details of graphical programming environments. The authors proposed a novel interface that provides an optimization technique, called the swap frame method. The method has been implemented as an extension of the Caravela library for the OpenGL runtime environment, using the functions handling the *I/O pair*, which implements recursive feedback in a flow-model.

According to experimental results, the extensions perform very well, achieving an improvement of about 60% performance compared with the one without the proposed optimization method. Finally, it can be concluded that the proposed method to provide a uniform interface to GPGPU applications operates effectively to exploit the potential performance of the GPU. The proposed extensions are prepared to accommodate new versions of the DirectX and the OpenGL graphics runtimes or even new runtime environments that turned out to be available in the future for GPUs.

6. ACKNOWLEDGEMENT

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

7. REFERENCES

[1] Caravela homepage. http://www.caravela-gpu.org/.
[2] DirectX for WINE homepage. http://directxwine.sourceforge.net/.
[3] DirectX homepage. http://www.microsoft.com/directx.
[4] Sh: A high-level metaprogramming language for modern GPUs. http://libsh.org/.
[5] The OpenGL Extension Wrangler Library. http://glew.sourceforge.net/.
[6] The OpenGL Utility Toolkit (GLUT) Programming Interface API Version 3. http://www.opengl.org/documentation/specs/glut/.
[7] 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.
[8] J. Kessenich, D. Baldwin, and R. Rost. The OpenGL Shading Language. *3Dlabs, Inc. Ltd.*, 2006.
[9] P. Kondratieva, J. Krüger, and R. Westermann. The Application of GPU Particle Tracing to Diffusion Tensor Field Visualization. In *IEEE Visualization*, page 10, 2005.
[10] P. S. McCormick, J. Inman, J. P. Ahrens, C. Hansen, and G. Roth. Scout: A Hardware-Accelerated System for Quantitatively Driven Visualization and Analysis. In *VIS '04: Proceedings of the conference on Visualization '04*, pages 171–178, 2004.
[11] K. Moreland and E. Angel. The FFT on a GPU. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 112–119, 2003.
[12] OpenGL Architecture Review Board, D. Shreiner, M. Woo, J. Neider, and T. Davis. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2.* Addison Wesley, 2005.
[13] 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.