Design and Implementation of a Stream-based Distributed Computing Platform using Graphics Processing Units

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

ABSTRACT

Anonymous use of computing resources spread over the world becomes one of the main goals in GRID environments. In GRID-based computing, the security of users or of contributors of computing resources is crucial to execute processes in a safe way. This paper proposes a new method for stream-based processing in a distributed environment and also a novel method to solve the security matter under this kind of processing. It also presents the design of the distributed computing platform developed for stream-based processing, including the description of the local and remote execution methods, which are collectively designated by *Caravela* platform. The proposed *flow-model* is mapped on the distributed processing resources, connected through a network, by using the Caravela platform. This platform has been developed by the authors of this paper specifically for making use of the Graphics Processing Units available in recent personal computers. The paper also illustrates application of the Caravela platform to different types of processing, namely scientific computing and image/video processing. The presented experimental results show that significant improvements can be achieved with the use of GPUs against the use of general purpose processors.

Categories and Subject Descriptors

D.2.2 [**Software**]: SOFTWARE ENGINEERING—*Design Tools and Techniques, Modules and interfaces*

General Terms

Design

Keywords

Caravela, GPU, stream-based computing, GRID, flow-model, GPGPU.

CF'07, May 7–9, 2007, Ischia, Italy.

1. INTRODUCTION

Worldwide distributed computing has become one of the remarkable possible ways to use anonymous computing power, due to the development of distributed execution platforms. The platforms can be based on message passing computing, using a server software such as Globus [3], or a mobile agent-based one that migrates among the resources organized as a virtual network [9]. According to the research reports of those platforms [5, 11], worldwide distributed computing is effective in achieving an ultra computing power, by taking advantage of huge amounts of unused computing power from over the world. This computing style is named GRID computing [18].

For a GRID computing platform, one of most important issue that the developer must address is security, both for users and for contributors of the computing resources. For example, the users of the platform do not desire that someone steals programs or data dispatched to a computing resource, in the communication channel to the resource or in the resource itself. On the other hand, contributors of the computing resources also do not desire that a program assigned by the users touch their private resources, such as protected data, hardware and network connections. The contributors may call such a program executing in their computers a *virus*. It must be paid a lot of attention to avoid such situations in a GRID.

This paper describes the design and implementation of the *Caravela* platform [1] that provides a novel mechanism to execute programs in a GRID environment, which provides a suitable security level of execution.

The *Caravela* defines a data structure for a unit of execution named *flow-model* unit, which is composed by input data streams, output data streams and a program to process those I/O data streams. The Caravela will assign the flow-model unit(s) to resources in the GRID environment. The program in the flow-model unit processes the input data in a stream-based processing flow, such as in a dataflow processor. According to the proposed execution model, the program in the flow-model unit is not allowed to touch the resources around the processing unit to which the flow-model is assigned, except for its I/O data streams.

The flow-model execution method fits well into a Graphics Processing Unit (GPU) because the GPU supports stream-based computation using texture inputs. Moreover, the performance of GPU is much higher when compared with that of a CPU [17]. Therefore, this paper also shows an implementation of the Caravela platform which maps the flow-model on a GPU. Moreover, this paper describes an example

where flow-model units execute distributed on remote GPU resources.

This paper is organized as follows. The next section contains a detailed explanation of the GRID environment, its security issues and discusses the GPU's validity for general purpose application. Section 3 presents the design concepts for the Caravela platform. Section 4 shows an implementation of the Caravela platform, assigning flow-model units to local and to remote GPU resources. In section 5, an application example with the Caravela platform is discussed, and finally section 6 concludes the paper.

2. BACKGROUND AND DEFINITIONS

2.1 The GRID computing environment

Among the platforms for GRID computing, there exist several methods to implement mechanisms to release resources for users and to remotely use those resources. One of them is Globus [3], a well known message passing-based platform for GRID computing. The users of the Globus platform can write programs as MPI-based parallel applications [12]. Therefore, applications which have been parallelized with MPI functions can easily migrate from a local cluster, or a supercomputer-based environment to the Globus platform. Another platform example is the agent-based implementation Condor-G [9]. This kind of implementation is mainly used for managing resources in a GRID. The tasks performed in remote computing resources using such platforms are assigned anonymously. It is very difficult for users and contributers to trust each other and be sure that the tasks never damage the computing resource and are not damaged by some malicious access. Therefore, in any implementation of GRID platforms the security must be considered as one of the important issues.

To achieve trustful communication among users and contributors of computing resources, any GRID platform must address the following security issues:

1. **Data security exchanged among processing resources via network**
   When a program is assigned to a remote processing unit, it must be sent to the resource and, also, the data must be received by the program. The data transferred via the network can be snooped by a third person using tools such as `tcpdump`. This means that the users do not trust the system. This problem is also a security matter in web-based applications. Therefore, data encryption such as SSL (Secure Socket Layer) is applied to the connections between computing resources [8].

2. **Program and data security on remote resources**
   On GRID environments, users would assign their programs to unknown machines anywhere in the world. Therefore, the users don't want that the program content or data be snooped or stolen by the resource owners. For overcoming this problem, the GRID platforms force the creation of an account for the user which is managed by the administrator.

3. **Resource security during program execution**
   This is the most dangerous security problem in the platform. When a program is dispatched by the user to a remote computing resource, it may make use anything in there. This is just the behavior of a computer *virus*. The GRID environment must have capabilities to restrict the permissions of user programs. Therefore, a GRID platform generally has resource management tools such as GRMS [4].

The first security problem above is solved by encrypting the data exchanged among resources and users. The second problem can be solved by the administrator of the computing resources, for example creating user accounts. However, in what respects the third problem, although some solutions tackle the user program access to resources, such as Java's RMI (Remote Method Invocation) mechanism [10] by restricting the available resources to the program in the virtual machine, it is very hard to configure the restrictions for all the applications. For this reason, in some applications which need to touch special resources on a remote host, Java allows the user program to open a security hole by using JNI (Java Native Interface) [15]. This is inconsistent with the security wall of the virtual machine. Therefore, we need to address the third problem by using a new execution mechanism for the programs.

2.2 GPUs

2.2.1 Computing power of GPUs

Graphical applications, especially 3D graphics visualization techniques, have drastically advanced in this decade. Even in a commodity personal computer we have available very high quality graphics created by real-time computing. This is mainly due to the GPU connected to the personal computer. The power of GPUs is now growing drastically: for example, floating point computation performance of nVIDIA's Geforce7 achieves 300 GFLOPS, which compares very well to 8 GFLOPS of Intel Core2Duo processors. This is a remarkable computational power available for applications which demand huge computation load.

Nowadays, researchers of high performance computing are also focusing the research on the performance of GPUs, and investigating the possibility for its usage as a substitute of CPUs. For example, using GPUs, GPGPU (General-Purpose computation on GPU) achieves a high level performance [14] [16]. A cluster-based approach using PCs with high performance GPUs has been reported in [7]. Moreover, compiler-oriented support for GPU resources has also already been proposed [6].

2.2.2 General purpose computing on GPUs

Let us explain the mechanism in GPUs that processes graphics objects in order to show them on a screen. The GPU acts as a coprocessor of the CPU via a peripheral bus, such as the AGP or the PCI Express bus. A VRAM (Video RAM) is connected to the GPU, which reads/writes the VRAM to process the graphics objects. To do this the CPU sends the object data to the VRAM, sends a program to the GPU, and controls the overall execution.

Figure 1 shows the processing steps done by the GPU to create a graphical image in a frame buffer in order to be displayed in a screen. First, the graphics data is prepared as a set of normalized vertices of objects on a referential axis defined by the graphics designer (Figure 1(a)). The vertices will be sent to a vertex processor to change the size or the perspective of the object, calculating rotations and transformations of the coordinates. In this step all the objects will be mapped to a standardized referential axis. In the

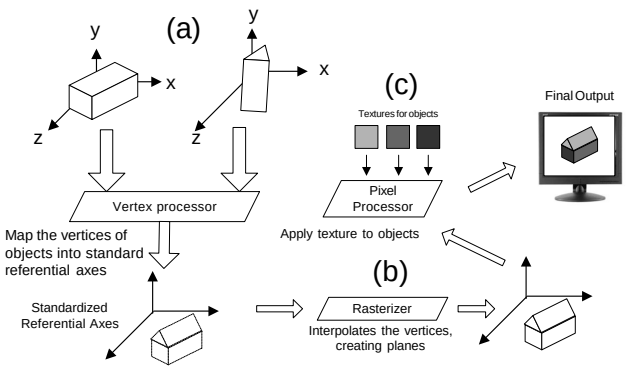


Figure 1: Processing steps for graphics rendering.

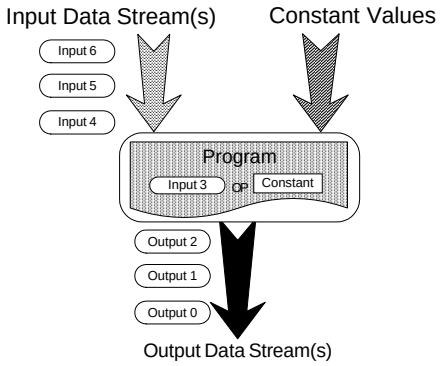


Figure 2: Structure of the flow-model.

next step, a rasterizer interpolates the coordinates and defines the planes that form the graphics objects (Figure 1(b)). Finally, a pixel processor receives these planes from the rasterizer and creates color data to send to the frame buffer, after calculating the composed RGB colors from the textures of the objects (Figure 1(c)). The color data is written into the frame buffer, which outputs it to the screen.

In recent GPUs, the vertex and pixel processors are programmable. The designers of graphics scenes can make programs for the processors, specific for its graphics effects. It is very important that the programs run fast in order to achieve a huge number of frames per second. Therefore, the GPUs have dedicated floating point processing pipelines in these processors and GPGPU applications make strong use of these processors. However, the rasterizer is composed of fixed hardware, and its output data can not be controlled. Moreover, the output data from the rasterizer is just sent to the pixel processor and can not be fetched by CPU. Thus, it is reasonable for GPGPU applications to use the computing power of the pixel processor due to its programmability capabilities and flexibility for I/O data control.

The focus of this paper is not only the performance of GPUs, but also the execution paradigm on GPUs. As shown above, the pixel processor does not touch any resources and the data sent to it is input as a stream of massive data. Then it processes each data unit (pixel color data) and outputs a data stream. This means that the program on the GPU works in a closed environment. Moreover, it is possible to write programs in the standard languages such as the DirectX assembly language, the High Level Shader Language (HLSL) [2] and the OpenGL Shading Language [13]. Thus, the program can run on any GPU connected to any computer.

According to the discussion above, it can be concluded that the security concerns about the resources touched by programs on a GRID platform can be solved by the GPU's execution mechanism, due to its stream-based processing. Thus, we aim to develop an execution mechanism on the GRID environment based on stream-based computation using GPUs' power.

3. DESIGN OF CARAVELA PLATFORM

The execution unit of the Caravela platform is defined as the *flow-model*. As shown in Figure 2, the flow-model is composed of input/output data streams, of constant parameter inputs and of a program which processes the input data streams and generates output data streams, by fetching each input data unit from the input streams. The application program in Caravela is executed as stream-based computation, such as the one of a dataflow processor. However, the input data stream of the flow-model can be accessed randomly because the input data streams are just memory buffers for the program that uses the data. On the other hand, the output data streams are sequences based on the unit of data in the stream. Thus, the execution of the program embedded in the flow-model is not able to touch other resources beyond the I/O data streams.

The presentation of the flow-model is one of the issues for the discussion. Although the same available representation for the PetriNet graph can be used, due to its dataflow-like processing style, the flow-model admits finite loops supported by the Caravela runtime. The Caravela runtime operates as a resource manager for flow-models. To implement a loop with the flow-model, the output data stream(s) are connected to the input data stream(s), providing data migration among the stream buffers.

The flow-model provides the advantage when applied in distributed environments of encapsulating all the methods to execute a task into a data structure. Therefore, the flow-model can be managed as a task object distributed anywhere, and can be fetched by the Caravela runtime. For example, when a flow-model is placed in a remote machine, an application over Caravela platform can fetch and reproduce the execution mechanism from the remote flow-model.

Regarding the processing unit to be assigned to a flow-model program, any software-based emulator, hardware dataflow processor, dedicated processor hardware, or others, can be applied.

3.1 Caravela runtime environment

The flow-model execution requires a managing system, which assigns and loads the flow-model program into a processing unit, allocates memory buffers for input/output data streams, copies the input data streams to the allocated buffers and triggers the start of the program. In addition, after program execution, the runtime may need to read back the output data from the output stream buffers to forward it to the next flow-model or to store it. The Caravela runtime defines two functionalities for flow-model execution: the local and the remote execution functions.

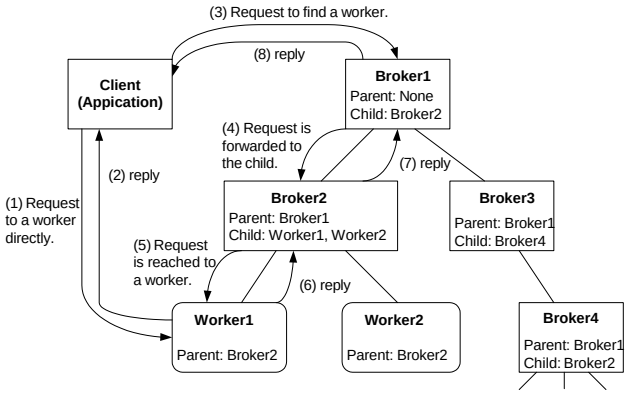


Figure 3: Caravela network example.

The execution in a local processing resource corresponds to following the steps referred above. The runtime checks if the program in the flow-model matches the specification of a local processing unit.

To support the remote execution of the flow-model, the Caravela runtime needs a function to respond to requests sent by Caravela platforms located in other remote resources. The servers placed in the remote resources are categorized into two types: *worker* and *broker* servers.

- Worker server
  The worker server acts as a processing resource that assigns one flow-model to its local processing unit. This server communicates with its client to send/receive output/input data of the flow-model. If an execution request from a client does not include the flow-model itself but an information of the location of the flow-model, the server will fetch it from the address. Then, the server will assign the flow-model to the local processing unit.

- Broker server
  The broker server performs as a router to reach the worker servers. The worker servers, after activation, send a request to register its route to one of broker servers. The broker server can have a parent broker server that accepts to register the route to a child broker. This mechanism creates a tree shaped worker network with broker servers as trunks of a tree. We call this tree-based virtual network the *Caravela network*.

Figure 3 shows an example of the Caravela network. The workers (Worker1 and Worker2) "belong" to the Broker1 and Broker2. Assume that the client tries to find a worker in the Caravela network and sends a request to the Worker1 directly (Figure 3 (1)). In this case, the client must know the route to the Worker1. The reply will be returned to the client directly (Figure 3 (2)). On the other hand, when the client sends the same request to the Broker1 (Figure 3 (3)), the Broker1 will forward the request to the Broker2 according to the routing information about the child(Figure 3(4)). The Broker2 knows that the Worker1 is its children and forwards the request to it (Figure 3 (5)). Finally, the request is processed and an answer returns by the opposite direction of the request and reaches the client(Figure 3 (6)(7)(8)).

Regarding resource limitation for security matters, the broker and the worker servers may not accept flow-models that specify larger data streams than the limits configured by the servers. This mechanism protects the contributors' environment from the trouble caused by memory consumption. Moreover, the worker servers may specify a time limitation based on a unit of a flow-model execution. When the time spent by our application exceeds the limit, the worker server may cancel the subsequent flow-model execution. This mechanism allows the resource contributors to quantify the percentage of his/her contribution for anonymous computing on Caravela network. Thus, these capabilities of the worker and the broker servers implement a secure environment for GRID computing.

3.2 Application Interface in Caravela platform

The interface for applications of Caravela platform is defined as a collection of functions to manage computing resources and to assign flow-models to the available computing resources. Applications on the Caravela platform need to follow the steps below:

1. **Initialization of the platform**
   In the beginning, the application initializes its context in the Caravela platform. This step creates a local temporal space for the subsequent management tasks.

2. **Reproduction of flow-model(s)**
   The Application fetches flow-model which may be in a remote location.

3. **Acquisition of processing unit(s)**
   To assign the flow-model, the application needs to acquire a processing unit that matches the conditions needed for the flow-model execution. If the application targets execution in a local processing unit, it queries directly the local resource. On the other hand, if the application needs to query the processing units of remote resources, for example when the requirements for the flow-model execution do not match the specification of the local processing unit, it sends a query request to worker or broker servers. If the application queries the worker, the worker will return its availability for flow-model execution. In this case, the application will send the requests directly to it. If the server is a broker, it will tell about all the available processing units it knows. In this case, the application will communicate to the broker server to execute the flow-model. Then the broker server will propagate the following requests to the worker server using its routing information.

4. **Mapping flow-model(s) to processing unit(s)**
   The application needs to map the flow-model to the processing unit reserved in the previous step. In the current step it will assign a program, I/O buffers and constant parameter inputs included in the flow-model. If the targeted processing unit is remote, the application exchanges requests with the worker servers.

5. **Execution of flow-model(s)**
   Before the execution in the processing unit starts, input data streams must be initialized. The execution of the flow-model is called "firing", which corresponds to activating a program in the flow-model and generating output data.

6. **Releasing processing unit(s) and flow-model(s)**
   After the execution of the flow-model, it is unmapped from the processing unit. Because the flow-model and the processing unit are not necessary in the next steps, they are released by the application.

7. **Finalization of the platform**
   Finally, the application needs to be terminated to exit from the Caravela environment.

The design considerations mentioned above are able to build a distributed processing platform using the flow-model framework. Because the flow-model includes enough information for independent execution, it performs stream-based processing without touching the resources in the host machine. Application in the Caravela platform is able to execute flow-models in a processing unit through secure execution mechanisms.

4. IMPLEMENTATION OF CARAVELA BASED ON GPUS

The Caravela platform has been implemented using GPU as the processing unit. First, we need to consider the content of the flow-model.

4.1 Packing flow-model

When GPUs are used as processing units of the Caravela platform, the flow-model unit includes a pixel shader program, textures as the input data streams, constant values of the shader program as the input constants and the frame buffers for output of the shader program as places to put the output data streams.

The flow-model unit needs to include also other important items related to the requirements for the program execution. The requirements consist mainly of the program's language type, its version and accepted data types, and an assembly version that shows significant differences, such as loop instruction available on Pixel Shader Model 3.0 or floating-point-based frame buffers.

We use the name "pixel" for a unit of the I/O buffer because the pixel processor processes input data for every pixel color. For example, a multiplication is performed with two registers that include ARGB elements as its operands, and outputs a register formed by ARGB elements.

In conclusion, the flow-model defines the number of pixels for the I/O data streams, the number of constant parameters, the data type on the I/O data streams, the pixel shader program and the requirements for the aimed GPU. To give portability to the flow-model, these items are packed into an Extensible Markup Language (XML) file. This mechanism allows the application in a remote computer to fetch just the XML file and easily execute the flow-model unit.

To help defining the flow-model, we have implemented a GUI-based tool, called *FlowModelCreator*, that is available in the Caravela package.

4.2 Applying GPU to processing unit

The application in the Caravela platform is supported by the Caravela runtime environment referred in section 3.1, which is running on the CPU of a commodity computer. Therefore, the application is a program which transfers and fires the flow-model unit execution in the GPUs according to the steps referred in section 3.2. For executing the flow-model in the GPU we need to define the resource hierarchy in the computer. Figure 4 shows a classification of the resource hierarchy in a computer. The group composed by the CPU and peripheral components, such as the host memory and the graphics boards, is defined as the "machine". A graphics board in the machine is defined as the "adapter". A GPU's pixel processor on the adapter is defined as the "shader". In summary, a *machine* may have multiple *adapters*, and the adapter may have multiple *shaders*. The application needs to get the shader to map the flow-model to be executed on a pixel processor.

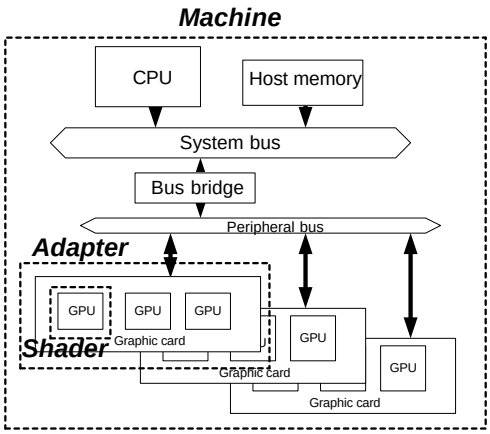


Figure 4: Resource hierarchy in a processing unit.

To control the pixel processor we need runtime software. Our first implementation of the Caravela platform uses Direct3D of the DirectX9 API and OpenGL. These runtimes provide functions to control the pixel processor. However, the interface is dedicated for graphics applications. Therefore, a rectangle plane object must be defined to present the output data streams of the pixel shader program. The plane acts as output target for multiple output data streams from the pixel shader program. However, on VRAM the buffers are separated into individual memory space. To save the output data stream, the runtime software fetches the spaces from the VRAM. According to this technique, a loop of a flow-model, or the connections of multiple flow-models, can be implemented with copy operations from the output data streams to the input data streams.

4.3 The Caravela library

To control the flow-model execution, implicitly controlling the GPU, the application uses the Caravela library functions programmed in C language.

The `CARAVELA_Initialize()` function performs the initialization of one of the graphics runtime specified by the argument and prepares the context to use the Caravela platform, while `CARAVELA_Finalize()` is called to release those resources.

The `CARAVELA_CreateFlowModelFromFile()` function is called to build a flow-model from an XML file. An address of a flow-model is defined as a URL. Therefore, the function accesses a flow-model placed in a remote resource by using the Hypertext Transfer Protocol (HTTP).

The `CARAVELA_CreateMachine()` function is called when the application needs to define a machine data structure. If the function returns successfully, the `CARAVELA_QueryShader()` function is called to acquire a shader.

After application has prepared a flow-model and a shader, it can call the `CARAVELA_MapFlowModelFromShader()` function. This function assigns the program in the flow-model unit to the pixel shader, allocates the I/O streams to the VRAM and returns a "fuse" to be used for triggering the flow-model execution. After receiving the "fuse", the `CARAVELA_FireFlowModel()` function sends commands to the pixel processor to execute the flow-model.

The `CARAVELA_GetInputData()` function prepares an input data stream in the host memory and the VRAM as texture data that will be input to the pixel processor. This function returns a buffer pointer for the input data stream which is used by the application to initialize the input data. On the other hand, the `CARAVELA_GetOutputData()` function returns an output data stream allocated in the VRAM.

Using the functions explained above, an application in the Caravela platform can locally execute a flow-model using the GPU's computation power. When the application needs more shaders, or the shader acquired does not match the requirements of the flow-model unit, it needs to query other shaders in the remote worker servers. In this case, the functions described in the next section are used.

4.4 The remote execution mechanism

The broker and the worker servers are implemented by a piece of software called *CaravelaSnoopServer*, runnning on a remote CPU. The CaravelaSnoopServer can be configured as a broker or as a worker server.

The requests for CaravelaSnoopServer are received by the WebServices via Simple Object Access Protocol (SOAP). The address of the WebServices is specified by a WSDL file placed in an predefined address of the server. Two service functions are provided by the server: `putRequest()` and `getReply()`. The request and the reply exchanged between the server and an application are formatted in XML. The `putRequest()` function saves the XML description into a file where the CaravelaSnoopServer can pick it. According to this mechanism, requests are sent by the application. The `getReply()` function returns the reply from the CaravelaSnoopServer after processing the corresponding request. The application, or other servers, call this function periodically to receive the reply.

When the application sends a request about shaders to a broker server, the request is saved in the server and processed by the CaravelaSnoopServer. If the server is a worker, the request will be processed and the `getReply()` function is called. If the server is a broker, the request will be forwarded to the next server until it reaches a worker server. Then, a reply from the worker will be fetched by the previous requester. Thus, the reply will be propagated till the application. When a broker server is invoked in a bridge between a Wide Area Network (WAN) and a Local Area Network (LAN), the request and the reply are also able to be exchanged among the servers connected to the different networks and the application successfully.

The Caravela library implements the mechanism mentioned above to execute the process remotely.

When the application calls the `CARAVELA_CreateMachine()` function with `REMOTE_MACHINE` or `REMOTE_BROKER` arguments, which indicates a worker or a broker server respectively, the function creates a machine structure of remote resources. If it is a `REMOTE_MACHINE`, the execution steps of the flow-model follow the same steps as the ones in the local execution. If it is a `REMOTE_BROKER`, the application tries to acquire workers by executing the `CARAVELA_GetRemoteMachines()` function and by using the broker machine structure. This function returns all the worker machines that are "seen" by the broker. Then, the application can select the appropriate worker machines and can map the flow-model(s) into the selected worker(s). After this step the application is able to follow the steps appropriate for local execution.

Requests and replies from and to the worker server are directly exchanged if the application selected a worker server. On the other hand, if it is a broker server, all the requests and replies are exchanged via the broker server. This mechanism has the advantage of being transparent for the applications.

Table 1: Environment for local execution.

CPU	AMD Opteron 170 @ 2Ghz
Host memory	2x1GB DDR
GPU	MSI NX7300GS
VRAM	256MB DDR2
OS	WindowsXP SP2
Graphics API	DirectX9c

5. EXPERIMENTAL EXAMPLE

Now, let us examine an experimental example that consists in a two dimensional Finite Impulse Response (2D FIR) filter algorithm. The 2D FIR filter is mainly used to perform image or video processing, like sharpening, or edge detection of an image/frame. The type of filtering is changed by using different taps values in the coefficient matrix, where common dimensions are 3x3 or 7x7. Here, we illustrate the programming of a filter with a 3x3 coefficient matrix:

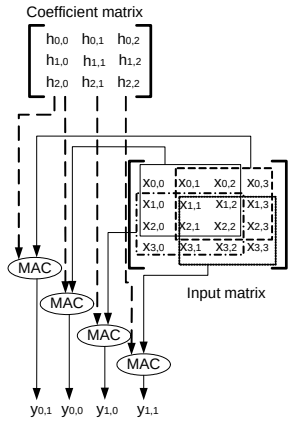
$$y_{k,l} = \sum_{i=0}^{2} \sum_{j=0}^{2} h_{i,j} x_{k+i,l+j} \quad (1)$$

h is the 3x3 coefficient matrix and x is the input matrix, size MxN, with k and l being integers in the range from 0 to M-3 and to N-3, respectively.

The calculation steps followed in this example are shown in Figure 5(a) for $M = N = 4$. The first step consists in multiplying each element of the sub-matrix (window) with corners in x_{00} and x_{22} with the correspondent elements of the coefficient matrix, and then adding all of them to get the output as the result y_{00}. This arithmetic operation is usually called *Multiply and ACcumulate* (MAC). In the second step, the input matrix' window is shifted to the right, and this step is repeated with the output going to y_{01}. Repeating this operation for every element of the input matrix, except for the elements belonging to the last two columns or rows, the application calculates the result of applying the filter to the input matrix.

Figure 5(b) shows the program to be embedded into a flow-model. This program is written in the DirectX' HLSL whose syntax is very similar to the C language. To be fit into the GPU's hardware architecture, the program assumes that M equals to $4 \times N$ due to the register characteristics, but it can be generalized.

The `main()` function is the routine executed in the GPU. We need to be careful about the arguments of the function, because the input to the pixel processor are the coordinates



(a) 2D FIR filter with 4x4 input matrix

```
sampler s0;
float4x3 c;

void main(
 in float2 t0:  TEXCOORD0,  // dcl t0.xy

        out float4 oC0: COLOR0 )
        {
  float inv = 1/c[3][0];
  float4 input_row0;
  float4 input_row1;

  int i,j;
  float2 coord = t0;
  oC0 = 0;
  for(i=0;i<3;i++,coord.y+=inv){
        input_row0 = tex2D(s0, coord);
        coord.x += inv;
        input_row1 = tex2D(s0, coord);
        oC0.x += (input_row0.x * c[0][i] + input_row0.y * c[1][i] + input_row0.z * c[2][i]);
        oC0.y += (input_row0.y * c[0][i] + input_row0.z * c[1][i] + input_row0.w * c[2][i]);
        oC0.z += (input_row0.z * c[0][i] + input_row0.w * c[1][i] + input_row1.x * c[2][i]);
        oC0.w += (input_row0.w * c[0][i] + input_row1.x * c[1][i] + input_row1.y * c[2][i]);
        coord.x= t0.x;
  }
```

(1)

(b) Program embedded in flow-model

Figure 5: The calculation steps and the program of a flow-model corresponding to a 2D FIR filter.

of the texture's pixels (`t0`). The output contains the pixel colors on the buffer for the screen (`oC0`). This function will be executed in parallel on the multiple pixel processors because the pixel values are independent, and outputs a pixel color by each input texture's coordinate.

The code shown in Figure 5(b)(1) corresponding to the calculation of equation (1), and accesses not sequentially the texture data by adding the offset `inv` to `coord` for each texture pixel. The `coord` is the 2D address of the texture (i.e. input matrix) and the `tex2D` function fetches the texture values. Therefore, the input data of this application is randomly accessed.

The values of the texture returned by `tex2D` function (i.e. the input matrix), the output `oC0` and the coefficient matrix `c` include four floating point values and calculates four elements of the output matrix `oC0`. Thus, the pixel processor also performs parallel processing.

5.1 A local execution example

The code using the runtime functions of the Caravela for local execution of the flow-model of 2D FIR filter is shown in Figure 6(a). At the beginning, the machine is created in step (1). The flow-model will be reproduced in step (2), from a path to an XML file defined in the `FLOWMODEL_FILE` macro. Using the machine structure, step (3) queries a shader from the local machine. If it is successful, the flow-model will be mapped to the shader in step (4). Here the input data stream is initialized as shown in step (5). After initialization, the flow-model will be fired in step (6). This function will block the subsequent execution until its execution has been finished. Therefore, the code for getting the output in step (7) is executed right after the firing. Finally, the flow-model and the shader are released in step (8).

Thus, the interface for executing the flow-model execution in the local machine is simple and transparent. Therefore, the programmer can write application for the Caravela platform without accounting for the details of the processing unit which is used.

5.2 A remote execution example

There exist two ways for remote execution of the flow-model. One of them requests a processing unit to a specific worker server as shown in step (9) of Figure 6(b). In this case, a remote machine is created as `REMOTE_MACHINE` with the `URL` for the remote worker. All the processes, such as querying shaders and mapping the flow-model, are performed by the machine structure returned by `CARAVELA_CreateMachine()`. On the other hand, when the request for acquiring a processing unit is performed via a broker server, as shown in step (10) of Figure 6(b), the machine structure will be created by passing `REMOTE_BROKER` to the `CARAVELA_CreateMachine()` function with the `URL` for a remote broker server, and the machine structure returned by the function will be passed to the `CARAVELA_GetRemoteMachines()` function. Finally, the available machines returned by the function will be used by the application, but the communication itself will be performed via the broker server.

In both cases, the processing steps after machine creation are the same as those presented in the description of local execution in Figure 6(a). Thus, it is easy for the application designer to migrate it from a local execution situation to a

```
CARAVELA_Initialize(RUNTIME_DIRECTX9);
CARAVELA_CreateMachine(LOCAL_MACHINE,NULL,&machine);                    (1)
CARAVELA_CreateFlowModelFromFile(FLOWMODEL_FILE,NULL,&flowmodel,&flowmodel_err);  (2)
CARAVELA_QueryShader(machine,&flowmodel->ShaderCondition, &shader);     (3)
CARAVELA_MapFlowModelIntoShader(shader,flowmodel,&compile_err,&fuse);   (4)
CARAVELA_GetInputData(flowmodel,0,&input_matrix);
for(i=0;i<NUMDATA;i++)
        for(j=0;j<NUMDATA*4;j++)                                        (5)
                GETFLOAT32_2D(input_matrix,NUMDATA,i,j) = input_martix_orig[i][j];
CARAVELA_FireFlowModel(fuse);                                           (6)
CARAVELA_GetOutputData(flowmodel,0,&output_matrix);
printf("output[%u][%u]=%f\n", NUMDATA-3,NUMDATA*4-3,                    (7)
        GETFLOAT32_2D(output_matrix,NUMDATA,NUMDATA-3,NUMDATA*4-3));
CARAVELA_UnmapFlowModelFromShader(flowmodel);
CARAVELA_ReleaseFlowModel(flowmodel);                                   (8)
CARAVELA_ReleaseMachine(machine);
CARAVELA_Finalize(RUNTIME_DIRECTX9);
```

(a) Caravela runtime code for local execution

```
CARAVELA_Initialize(RUNTIME_DIRECTX9);
#ifdef REMOTE_IS_WORKER
CARAVELA_CreateMachine( REMOTE_MACHINE, URL, &machine);                 (9)
#else // REMOTE_IS BROKER
CARAVELA_CreateMachine( REMOTE_BROKER, URL, &machine);                  (10)
CARAVELA_GetRemoteMachines(machine,&num_machines,&worker_machines);
#endif
... the rest  is the same way as the local execution.
```

(b) Caravela runtime code for remtoe execution

Figure 6: An example of application code on the Caravela platform for performing local and remote flow-model execution.

remote execution situation by changing only a small part of the code for machine creation.

5.3 Performance Considerations

To evaluate the performance of the local execution of the FIR filter depicted in Figure 5(b), we measured execution times when using the computing environment referred in Table 1, with 100 iterations on a 1024x1024 pixel input matrix (i.e. 1024x4096 floating point values at the input of the 2D FIR filter). In order to have a comparison reference, we have also implemented the 2D FIR filter on the CPU side. The input matrix size and the number of iterations are the same in both experiments.

The calculation time on the Caravela platform is 10.6 s, which compares with 23.5 s on the CPU-based version. The Caravela platform achieves about 2.2 times higher performance than the host CPU. Thus, we can conclude that, by providing a secure environment and a transparent interface for programmers and resource contributors, the current implementation of the Caravela platform smoothly assigns the flow-model to the pixel processor on a GPU for example. It also implements a high performance stream-based computing environment.

6. CONCLUSIONS

This paper has described the Caravela platform where applications can be invoked by the secure execution mechanism with the proposed flow-model. Moreover, it presented the design and the implementation of the Caravela platform using GPUs as its processing unit. The application interface to the Caravela platform is transparent to program the execution steps for the local and the remote execution mechanisms.

An experimental example running in the Caravela platform has also been shown. According to the obtained results, it can be concluded that the Caravela platform, namely under local execution, is able to significantly improve the performance of stream-based computation, when compared to CPU-based execution.

Regarding the evolution of the Caravela platform, it is planed to implement a pipelined execution mechanism of the flow-models, called *meta-pipeline*, which will distribute the flow-models in the Caravela network. This mechanism will create a virtual meta network of flow-model units and will execute them with data received at the input of the virtual network.

7. ACKNOWLEDGEMENTS

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

8. REFERENCES

[1] Caravela homepage. http://www.caravela-gpu.org/.

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

[3] Globus alliance. http://www.globus.org/.

[4] Gridlab resource management system http://www.gridlab.org/workpackages/wp-9/.

[5] D. Bernholdt, S. Bharathi, and et al. The Earth System Grid: Supporting the Next Generation of Climate Modeling Research. *Proceedings of the IEEE*, 93:485–495, 2005.

[6] 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.

[7] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU Cluster for High Performance Computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 47, Washington, DC, USA, 2004. IEEE Computer Society.

[8] A. O. Freier, P. Karlton, and P. C. Kocher. *The SSL Protocol Version 3.0*. Netscape communications corporation, 1996.

[9] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC10) San Francisco*, 2001.

[10] W. Grosso. *Java RMI*. O'Reilly Media, 2001.

[11] R. Jacob, C. Schafer, I. Foster, M. Tobis, and J. Anderson. Computational Design and Performance of the Fast Ocean Atmosphere Model, Version One. In *2001 Intl Conference on Computational Science*, 2001.

[12] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing*, 2003.

[13] J. Kessenich, D. Baldwin, and R. Rost. The OpenGL Shading Language. *3Dlabs, Inc. Ltd.*, 2006.

[14] P. Kondratieva, J. Krüger, and R. Westermann. The Application of GPU Particle Tracing to Diffusion Tensor Field Visualization. In *IEEE Visualization*, page 10, 2005.

[15] S. Liang. *Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley Professional, first edition, 2001.

[16] 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.

[17] 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.

[18] D. A. Reed, C. L. Mendes, C. da Lu, I. Foster, and C. Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure - Application Tuning and Adaptation*. Morgan Kaufman, 2003.