Meta-Pipeline: A New Execution Mechanism for Distributed Pipeline Processing

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

Tomás Brandão
IT-Lisbon/ISCTE
Av. Rovisco Pais, 1,
1049-001 Lisboa, Portugal
tomas.brandao@lx.it.pt

Abstract

The Caravela platform has been proposed by the authors of this paper to perform distributed stream-based computing on general purpose computation. This platform uses a secured execution unit called flow-model that prevents remote users to touch local information in a computer. The flow-model is assigned to local or remote processing units that execute its program. This paper is focused on a new execution mechanism that defines a pipeline composed by flow-models, called meta-pipeline, and is designed as a set of additional functions of the Caravela platform. The pipeline is executed automatically by the meta-pipeline runtime environment. This paper describes the execution mechanism and also presents an application example.

1 Introduction

Distributed computing environment, where computers, clusters and supercomputers are connected via the Internet for anonymous usage of resources, has spread over the world [13]. The environment for such a huge computing resource is called GRID environment. The GRID environment is accessed via a wide area network by users' computers located anywhere in the world. Although the accessibility and the huge horsepower are attractive, the environment needs to support distribution of the tasks by the different computers and privacy for the users and safety for the contributors to the computing resources. For instance, an application on GRID environment might touch a computing resource illegally, and it also might make the resource down, or it might steal private information stored in the resource, if the environment has security holes.

To support stream-based computing on the recent computers and simultaneously avoid the security matters in GRID environment, the *Caravela* platform has been proposed [2][14]. The Caravela platform defines *flow-model* as depicted in Fig. 1, which is implemented through a data structure that holds I/O data streams, constant parameters and a program. The program in a flow-model generates output data streams by processing every unit of the input data streams. Since the flow-model is defined as a data structure, the application can fetch it from any of the remote machines and may reproduce its execution also in any machine. Therefore, the flow-model can be assigned into any processing resource that fits into the stream-based execution style. Due to this characteristic, the Caravela platform prepares servers located in any processing unit to execute flow-models. Because a program in a flow-model is not able to touch any resource except its I/O data streams, the security matters in the GRID environment referred above are solved.

The Caravela platform accepts an application that concurrently executes multiple flow-models in different servers. However, after remote execution of a program in a flow-model, the current execution mechanism returns the output data stream(s) to the application. The flow-model's I/O streams can not be connected to each other, which causes large communication overhead between the application and the servers. Therefore, with an higher level execution model that can represent larger calculations by connecting the I/O data streams of multiple flow-models, it can cooperate to solve a given application. By creating a processing pipeline to connect flow-model's I/O streams, the output data streams can be directly propagated from one server to the other, and thus the application can be autonomously executed in distributed computing resources.

This paper is focused on an execution mechanism called *meta-pipeline*. It defines the *pipeline-model* that is implemented with flow-models whose I/O data streams are directly interconnected. This paper also describes details of the execution mechanism for implementing a pipeline-model in the Caravela platform by using multiple distributed processing units. For evaluating the proposed meta-pipeline mechanism implemented in the Caravela platform this paper also shows a realistic application example.

The paper is organized as follows. The next section shows the background of this research and presents the Car-

avela platform. Section 3 describes the design of the metapipeline mechanism and section 4 illustrates its implementation. Section 5 discusses an application using the metapipeline mechanism. Finally, conclusions are drawn in Section 6.

2 Background

2.1 GRID Computing environment

There exist two types of implementations for the GRID platform: message passing based and agent-based. Currently, the most popular example of the former one is the Globus [7]. The applications working on Globus use well known MPI-based message passing to communicate among distributed processes. An example of the latter one is Condor-G [6], whose agents go around the GRID environment assigning tasks to the computing resources.

Tasks will be assigned anonymously to remote resources by using the platforms above. This anonymous use of the resources forces the programmers or the contributors to think about security matters on the platform. The security issues can be categorized as: 1) data security in the network, 2) program or data security on remote resources and 3) resource security during task execution. The first problem can be solved by encryption, and the second one can be solved by administrating the resource by creating a dedicated account for the users and the processes. The third problem exists because the application may illegally access private devices on remote resources, or may monopolize the use of memory or CPU resources. Although Java RMI mechanism [9] and GRMS [8] have been proposed to solve this problem, it is impossible for the resource manager to provide a completely secured environment. Java allows to create a security hole by JNI [11] and the GRMS needs to be configured without security holes in the software installed in the remote resource. Therefore, this problem must be addressed using a new mechanism for executing a program in a remote resource.

2.2 Graphics Processing Units

Graphical visualization techniques using Graphics Processing Units (GPUs) connected to personal computers have drastically advanced in the last decade. For instance, the floating point computation performance of nVIDIA's Geforce7 achieves about 300GFLOPS, comparing to 8GFLOPS of Intel Core2Duo processors. This is a remarkable performance for applications that demand huge computation power.

High performance computing researchers are also focused on GPU's performance, and investigating the possibility of using it as an alternative to the Central Processing Unit (CPU). For example, the GPGPU (General-Purpose computation on GPU) approach achieves high level performance [12]. Using PCs with high performance GPUs, a cluster-based approach has been reported [5]. Moreover, compiler-oriented support for GPU resource has also been proposed [1].

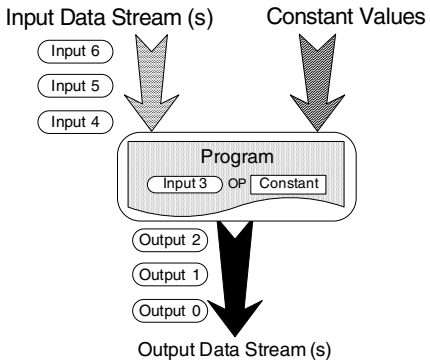


Figure 1. Structure of the flow-model.

Let us consider the processing steps on a GPU to create a graphical image in a frame buffer, which can be displayed in a screen. First, the graphics designer prepares a set of normalized object vertices on a referential axis. The vertices are sent to a vertex processor which is responsible for coordinate transforms (e.g. rotation, translation). In the next step, the rasterizer interpolates the coordinates and defines the planes. Finally, the pixel processor receives these planes and creates pixel color data for the frame buffer, calculating composed RGB colors from textures. Because the recent GPUs are programmable, the pixel color generation is adoptable to any target graphics effects. The program executed on the pixel processor is called *shader* program.

This paper is focused not only on the performance of GPU, but also on the program and execution styles of GPUs. In GPGPU algorithms are adopted to shader programs and textures act as input data and the pixel color data corresponds to the output results. The pixel processor does not touch any resources except I/O data buffers and the processor input data is formatted as a data stream. Then it processes each data unit (pixel color data) and outputs data stream(s). This means the shader on the GPU operates in a closed environment. Moreover, standard programming languages such as the DirectX assembly language, the High Level Shader Language (HLSL) [4] and the OpenGL Shading Language (GLSL) [10] are available to describe shader programs. Thus, the shader can run on any GPU in any computer.

According to the discussion above, it can be concluded that the security issue about the resource touching on the GRID computing can also be solved by the proposed stream-based execution mechanism.

2.3 Caravela platform

To implement stream-based processing on GPUs and to solve the security problems in GRID environment, we have developed a stream-based computing platform named *Caravela* [2][14].

2.3.1 Stream-based computing using the flow-model

An execution unit of Caravela platform is defined as a *flow-model* unit. As shown in Fig. 1, the flow-model is composed by input/output data streams, constant input parameters and a program that processes the input data streams to generate output data streams. An application program in Caravela is executed as a stream-based computation, such as one in a dataflow processor. However, the input data stream in a flow-model can be randomly accessed because the input data streams are in memory buffers where the program reads. On the other hand, the output data streams must be the sequential streams of data units. Thus, the execution of the program embedded in the flow-model is not able to touch other resources except the I/O data streams.

The flow-model data structure can be packed to an Extensible Markup Language (XML) file. The XML file contains properties of a flow-model, namely the number of I/O data streams and a program. The XML-based flow-model has an advantage for a distributed environment because all the methods to execute a task are encapsulated 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 environment.

2.3.2 Caravela runtime environment

The execution of a flow-model requires a managing system, which assigns and loads a flow-model program into a processing unit, allocates the memory buffers for the input/output data streams, copies the input data streams to the allocated buffers and triggers to start the program. Moreover, after the program execution, the runtime may need to read back the data from the output stream buffers in order to pass it to the next flow-model or to store it. The first implementation of the Caravela platform applies GPUs as the processing units.

The Caravela runtime environment defines local and remote execution functions for flow-model. To support the remote execution of a flow-model, the Caravela runtime needs a function to serve requests that are sent to flow-models located in other remote resources. The servers are categorized into *worker* and *broker* servers. The worker server acts as a processing resource that assigns a flow-model to its local processing unit, namely GPU(s). If an execution request of a flow-model from a client does not include the flow-model's content, but a URL to the flow-model, the server will fetch the flow-model from the corresponding address. On the other hand, the broker server acts as a router to reach the worker servers. A worker server sends a request to register its route to one of the broker servers whenever worker server is activated. The broker server can have a parent broker server that accepts to register the route to this child broker. This mechanism creates a tree shaped worker network with the broker servers in the trunks of the tree. We call this tree-based virtual network the *Caravela network*.

The SOA protocol is used for implementing message passing among the application, the worker and the broker servers. When it is a client, it invokes a function of the Web-Services' function on the server side and fetches the result of the requests.

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 flowmodel to a shader
`CARAVELA_FireFlowModel:` executes a flowmodel mapped to a shader

2.3.3 Caravela library

The Caravela platform is mainly composed by a library that adopts the following definitions for the processing units: *Machine* is the host machine of a video adapter, *Adapter* is a video adapter that includes a single or multiple GPUs, and finally *Shader* is a GPU. The application needs to map a flow-model into a shader, which, in turn, executes the mapped flow-model. Table 1 shows the basic Caravela functions for flow-model execution. Using those functions, the programmer implements target applications in the framework of flow-models, and the application just has to map the flow-model(s) into the shader(s). For remote execution of a flow-model, the `REMOTE_MACHINE` keyword is passed to the `CARAVELA_CreateMachine` function. In this case, the shader(s) returned by `CARAVELA_QueryShader` function belong(s) to the worker server(s).

Thus, using the flow-model framework, the Caravela platform has implemented a secure and high performance stream-based computing environment, applying GPUs as

the processing units. However, the application must manage flow-model execution and data forwarding between the flow-models. Even if flow-models for a target algorithm can be executed in a pipeline manner, a large communication overhead is imposed to feedback the reply to the application. This paper is focused on an extension of the execution mechanism to a pipelined processing mechanism, directly connecting flow-models assigned to multiple distributed processing units in the Caravela network.

3 Design of meta-pipeline

The mechanism that executes multiple flow-models whose I/O data streams are connected in the Caravela platform is called *meta-pipeline*. The meta-pipeline applies an execution model, named *pipeline-model*, as the execution unit in the system. Let us begin by explaining the meta-pipelining mechanism from the definition of the pipeline-model.

3.1 Defining pipeline-model

As depicted in Fig. 2, flow-models whose I/O data streams are connected can create a pipeline-model. A pipeline-model may have its own I/O *ports*, which can be seen as its own I/O data streams. Input data streams (1) in Fig. 2 are called *ENTRANCE* ports and output data streams (2) in Fig. 2 are called *EXIT* ports. Because the pipeline-model is defined to compute programs, it must fulfill the following conditions: 1) one or more EXIT ports must exist; 2) one or more flow-models are included; and 3) all the flow-models are connected by at least one I/O data stream. The first condition means that at least an output data stream is provided as the result of the pipeline-model, because otherwise the algorithm would be useless. The second and third conditions mean that a pipeline-model that includes two or more independent processing pipelines is invalid because multiple independent processing pipelines can be separated into different pipeline-models. Note that assumptions about the number of input ports are omitted from the above conditions. This is because the pipeline-model can have a self-generated feedback input stream data that results from the output data stream. This organization can be created for algorithms that generate special data streams, for example the recursive sequence of numbers like Fibonacci numbers.

When all input data streams of a flow-model included in a pipeline-model are ready, the flow-model can be executed. After checking this condition for each flow-model, the execution mechanism continues to invoke a pipeline-model. For example, in the pipeline model depicted in Fig. 2(a), data for the ENTRANCE ports (1) are prepared. Then, 'flowmodel0' is executed and generates the output data stream needed to 'flowmodel1'. At this time, the input

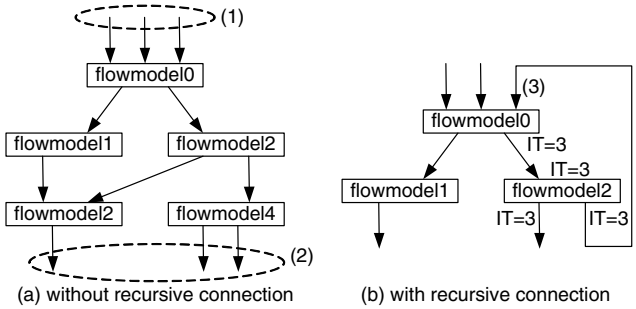


Figure 2. Examples of pipeline-model

data stream of 'flowmodel1' can be prepared. This execution style will be repeated in the subsequent flow-models, and thus flow-model execution will be pipelined. In the example of Fig. 2(b), at the beginning of the execution the input data streams of 'flowmodel0' never become all ready unless the data stream of the feedback connection is initialized. This way causes a deadlock of the pipeline-model's execution (i.e. stalls). Therefore, this kind of input data stream, called *INITONCE* port, is a special port that must be initialized before pipeline execution starts.

For increase flexibility, we also defines a limit number of iterations without initializations of input data for INITONCE port. For example, if the input data stream (3) is defined as an INITONCE port and the *iteration limit* is three, as illustrated in Fig. 2(b), the port must be initialized every three times the output data is generated by the 'flowmodel2'. In addition to the INITONCE ports, iteration limits are applied to ENTRANCE and EXIT ports. In the case of ENTRANCE port, an iteration limit restricts input data initialization. To say in the other words, the ENTRANCE port is initialized every "number of iteration limit" executions. On the other hand, EXIT port with iteration limit generates output data every "number of iteration limit". Moreover, we call *INTERMEDIATE* port to the I/O streams which are in any one of the categories referred above. The concept of iteration limit is also applied to the INTERMEDIATE ports. For instance, illustrated in Fig. 2(b), when the iteration limit (in the figure, shown as IT) is set, the output data from 'flowmodel0' is generated every three executions and the input/output data of 'flowmodel2' are initialized/generated every three iterations of 'flowmodel2'. A merit of the iteration limit is to define an execution set. In this example, 'flowmodel2' can be iterated without initializing the input data stream and without generating the output data stream. This allows the remote processing unit assigned to a flow-model not to send/receive data at every execution. Thus, redundant data communication among processing units is avoided.

In summary, flow-model execution in a pipeline-model is repeated while its flow-model's ENTRANCE port(s) or

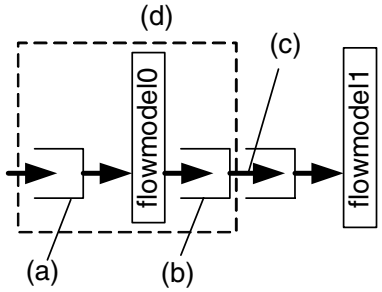


Figure 3. Execution model in pipeline-model

INITONCE port(s) are initialized correctly. Moreover, the executions of flow-models can be parallelized on independent processing units, and the iterated processing being optimized through the iteration limits. Thus, the pipeline-model will behave in a distributed environment as a suitable autonomous stream-based computing unit.

3.2 Runtime environment

The meta-pipeline mechanism needs a runtime support for executing a pipeline-model on processing units. By checking if all input data streams of flow-model are ready, the runtime provides the maximum number of processing units to execute the flow-models. Although a connection between flow-models' input and output data streams can be prepared as a shared buffer that receives the output data stream and allows it to be used as the next input data stream, the execution of flow-models according to the pipeline-model becomes sequential. A flow-model which is going to be executed can not output its data to the buffer if the buffer is occupied by data for the subsequent flow-model. To avoid this serialization, we define an execution model as depicted in Fig. 3. Each flow-model has buffers corresponding to its input and output data streams (Fig. 3(a)(b)). When the input data streams are initialized and 'flowmodel0' is invoked, its output data streams are stored to the output buffer once (Fig. 3(b)). When the subsequent flow-model's input buffer becomes empty, content of the output buffer will be moved to the input buffer (Fig. 3(c)). This mechanism avoids serial execution, in this case between 'flowmodel0' and 'flowmodel1' in Fig. 3, by using independent buffers for input and output data streams.

Each group of a flow-model with input/output data stream(s) and its buffers, such as depicted in Fig. 3(d) can be invoked as a single flow-model execution using the Caravela library shown in Table 1. Therefore, mapped to any processing unit in the Caravela platform, this group is able to be invoked independently.

When an output buffer and an input buffer are connected between flow-models and those buffer sizes are different, the data will be copied using the smaller buffer size. For example, when the buffer size of Fig. 3(c) is smaller than the one of Fig. 3(b), the data in Fig. 3(b) is resized to Fig. 3(c) and is copied to the destination. Therefore, even if different sized buffers are connected in a pipeline-model, the I/O data will be smoothly propagated.

3.3 Application interface

An application interface for the meta-pipeline is prepared as additional functions of Caravela library. To invoke a pipeline-model, application follows the eight steps presented below;

(1) Looking for processing units: this step uses the conventional Caravela functions shown in Table 1 and acquires processing units; the processing units can be located locally or remotely.

(2) Creating flow-models: this step is also available through the functions associated to the conventional Caravela platform; flow-models that are used in a pipeline-model are reproduced from local or remote flow-model XML files in this step.

(3) Creating a pipeline-model data structure: a pipeline-model is created as a data structure in the Caravela system; in this step, application creates the data structure.

(4) Registering flow-models and processing units to pipeline-model: this step registers into the pipeline-model data structure the flow-models produced in the 2nd step and the processing units queried by the 1st step; a pair flow-model/processing unit is named *stage* of the pipeline; application needs to create all the stages in a pipeline-model in this step.

(5) Creating connections among flow-models: this step defines connections among I/O data streams of flow-models registered to the different stages of pipeline-model; to fulfill the conditions to be a valid pipeline-model, application must connect appropriate I/O data streams of flow-models.

(6) Defining INITONCE ports and iteration limits: regarding to the connections defined in the previous step, if a connection creates a loop, it must be marked as an INITONCE port at the corresponding input data stream of the flow-model; iteration limits associated to ports in the pipeline-model must be also specified in this step.

(7) Implementing pipeline-model: this step checks if the pipeline-model satisfies the conditions and if the pipeline-model is available to be executed; if so, all the processing units registered to the pipeline-model are reserved. If the resources are located in remote machines, connections to communicate data among stages are established; moreover, the flow-models associated to the processing units are sent to them; thereafter, each flow-model becomes ready to be executed waiting for input data via INITONCE/ENTRANCE ports.

(8) Invoking pipeline-model: the invocation of the pipeline-model is automatically made by sending input data to its ENTRANCE/INITONCE ports; this operation must be performed on the application side; when input data are provided at the first stage of a pipeline-model it is executed and provides output data to the next stage; this execution mechanism is propagated until EXIT ports appear in a stage; the application needs to keep sending input data as long as ENTRANCE/INITONCE ports are waiting for input data; stages are executed while data is received and when data reaches the EXIT ports it must be received by application; due to the pipeline execution mechanism, while the output data is not completely received by the application the stages associated with the EXIT ports will stall; therefore, as soon as the output data is ready at the EXIT ports, it must be received by the application.

Following the steps above, the application sets up a pipeline-model and distributes flow-models registered to the pipeline-model by remote resources. Moreover, the pipeline-model is implicitly executed by the meta-pipeline's runtime, as the application provides input data to the ENTRANCE/INITONCE ports. Thus, pipeline-model programmers do not need to explicitly schedule flow-model execution in the application program.

4 Implementation of meta-pipeline

4.1 Library functions

The meta-pipeline is implemented as a set of C-based functions. The required functions not included in the meta-pipeline are provided in the conventional function set. For example, machine creation and shader acquisition are performed by the conventional functions as listed in Table 1.

After application acquires shaders, a set of meta-pipeline functions is used. First, `CARAVELA_CreatePipeline` function is called to create a data structure for the pipeline-model. At the next step, `CARAVELA_AddShaderToPipeline` function adds a shader acquired by `CARAVELA_QueryShader` through the conventional Caravela library to the pipeline-model data structure. Associating the shader to the pipeline-model, a flow-model created by the `CARAVELA_CreateFlowModelFromFile` is added by the `CARAVELA_AttachFlowmodelToShader` function. Here, a pair of a shader and a flow-model has been registered in the pipeline-model. Thus, stages in the pipeline-model have been defined.

To define connections among stages, `CARAVELA_ConnectIO` function is used. Its arguments are a flow-model registered in a pipeline-model and input/output data streams' indices for the flow-model. After the definition of connection is successful, the I/O data streams are marked as INTERMEDIATE ports. If needed,

```
// Preparing input data
Input_data = ... ;
While(1){
        // Sending input data to pipeline-model.
        if(CARAVELA_SendInputDataToPipeline(
                pipelinemodel,  // a pipeline-model
                port,           // an ENTRANCE port to be initialized
                input_data,
                number_of_data) == CARAVELA_SUCCESS){
                        // Preparing the next input data
                        Input_data = ... ;
        }
        // Getting output data from pipeline-model.
        if(CARAVELA_ReceiveOutputDataFromPipeline(
                pipelinemodel,  // a pipeline-model
                port,           // an EXIT port to be initialized.
                &flowmodel,     // a flow-model of the port (output from function)
                &index,         // an index of output stream of flow-model
                                                        (output from function)
                &output_data,   // (output from function)
                &number_of_data // (output from function)
                ) == CARAVELA_SUCCESS){
                        // Processing output data
        }
}
```

Figure 4. Code for pipeline-model execution

`CARAVELA_SpecifyInitOncePort` is called after making a connection, to specify that an INTERMEDIATE port is an INITONCE port. Regarding to the iteration limit, as mentioned in the previous section, after creating connections among stages, `CARAVELA_SpecifyEntrancePort`, `CARAVELA_SpecifyExitPort` and `CARAVELA_SpecifyIntermediateInput/Output` functions can be called when an application needs to specify iteration limits at some ports. The setup of the pipeline-model was finished, and the application starts to implement the pipeline-model.

The function `CARAVELA_ImplementPipelineModel` implements the pipeline-model itself. It assigns flow-model/shader pairs (i.e. stages) to machines equipped with shaders. This function returns arrays of ENTRANCE and EXIT ports. This function does not execute any stages in the pipeline-model. Those executions are performed by `CARAVELA_SendInputDataToPipeline`, that sends input data to ENTRANCE ports, and `CARAVELA_ReceiveOutputDataFromPipeline` function that receives output data from the EXIT ports. These functions have internally a routine for executing the stages. If condition for executing a stage is satisfied, the routine executes the stages one after another. This execution mechanism is categorized in two ways: local execution and remote execution. The detailed implementation of this mechanism is described in the next section.

4.2 Execution mechanism

Because execution of a pipeline-model is fired by input data in ENTRANCE ports, input data must be provided to the ports repeatedly after the ports has consumed the data. Moreover, results generated from EXIT ports must be read by application side to avoid stalls. Therefore, coding

style of pipeline-model execution becomes the one shown in Fig. 4, which creates an iteration that tries to input new data to ENTRANCE ports and to get new results from EXIT ports.

For executing stages in local shaders, `CARAVELA_SendInputDataToPipeline` and `CARAVELA_ReceiveOutputDataFromPipeline` functions have the chances to execute stages by utilizing the repeated executions shown in Fig. 4. In our implementation, a routine that finds stages with all the input data streams available is called by these functions. Thus, these functions execute available stages while any ENTRANCE port is not empty and any EXIT port can output data.

In our implementation, for executing stages on remote shaders, `CARAVELA_ImplementPipelineModel` function distributes flow-models associated to stages in a pipeline-model to worker servers. Worker servers prepare its shader resources for receiving flow-models and wait for data to the respective ENTRANCE ports. When the worker servers receive input data for flow-models and execute it, they forward the output data to the other worker servers that have the subsequent flow-models in the pipeline-model. Execution of flow-model in a worker server is fired by input data. Therefore, application does not need directly to activate each flow-model distributed in remote shaders.

As mentioned above, the meta-pipeline mechanism allows applications to define stages, which are implicitly executed in local or in remote machines.

5 Application example

To evaluate the meta-pipeline mechanism, this section discusses a simple but realistic application example. Here, let us apply the mechanism to compute a 2D discrete wavelet transform.

5.1 2D Discrete Wavelet Transform

Discrete Wavelet Transform (DWT) [3] is attracted as a powerful tool for image processing applications, such as compression used in JPEG2000 standard, denoising, edge detection and feature extraction. The DWT decomposes an input signal $S(i)$ into two sub-band coefficient sets: a set of low frequency coefficients $L(i)$ and a set of high frequency ones $H(i)$. After using linear low-pass and high-pass filters to the input signal $S(i)$, a decimation process is pipelined. Representing a k-th low-pass filter coefficient by $l(k)$ and a high-pass filter one by $h(k)$, the i-th DWT coefficient in the corresponding sub-band is computed by:

$$L(i) = \sum_{k=0}^{K-1} S(2i+k)l(k), H(i) = \sum_{k=0}^{K-1} S(2i+k)h(k)$$

(a) 2D-DWT sub-bands (b) input image (c) Output of 2D-DWT

Figure 5. 2D-DWT example with 2 decomposition levels

where K is the number of filter taps. Here, the decimation is already performed by the equations above, thus the number of coefficients per sub-band becomes half the number of samples of the input signal.

Due to the separable property of DWT, two dimensional DWT (2D-DWT) can be performed by sequentially applying the equations above through the horizontal and vertical image directions. It generates 4 sub-bands (i.e. LL, HL, LH and HH as shown in Fig. 5(a)). Each sub-band corresponds to a possible combination of direction (horizontal/vertical) and filter response (low/high-pass). To generate four new sub-bands, the same calculation is applied to the previous LL sub-band. This recursive calculation is iterated until the given number of decomposition levels is achieved (typically 3 to 5 levels). Fig. 5(c) shows a result of 2D-DWT with 2 decomposition levels calculated from the input image in Fig. 5(b).

5.2 Pipeline-model for 2D-DWT

As mentioned above, the recursive nature of N-level DWT decomposition suggests a pipeline-model organization where each pipeline stage corresponds to one decomposition level. Each stage in this pipeline can be a single kernel program that generates the LL_n sub-band to be used in the next stage:

$$LL_n(i,j) = \sum_{k=0}^{K-1}\sum_{m=0}^{M-1} LL_{n-1}(2i+k, 2j+m)l(m)l(k),$$

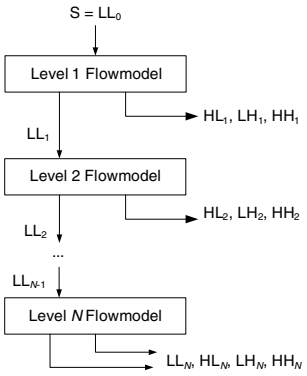
and also the remaining sub-bands (HL_n, LH_n and HH_n), using the correspondent filter combinations. Therefore, a flow-model with an input stream and two output streams is applied to each stage in the pipeline: one of the output streams corresponds to LL_n and the remaining sub-bands form the other.

The program included in the flow-model is shown in Fig. 6(a). The program is written in GLSL. It receives an input data stream that includes the previous LL sub-band and generates two output data streams.

```
uniform sampler2D CaravelaTex0;
uniform sampler2D CaravelaTex1;
// Daubechies-4 low-pass filter coefficients
uniform vec4 const0;
// Daubechies-4 high-pass filter coefficients
uniform vec4 const1;

void main()
{
 float delta = 1/NUMDATA;
 vec4 tmp, tmp0, tmp1, result;
 vec2 coord = gl_TexCoord[0].xy;
 vec2 caux;
 int i;
 coord += coord;
 caux = coord;
 // horizontal direction
 for (i=0; i<4; i++, coord.y += delta){
   tmp.x = texture2D(CaravelaTex0, coord).x;
   coord.x += delta;
   tmp.y = texture2D(CaravelaTex0, coord).x;
   coord.x += delta;
   tmp.z = texture2D(CaravelaTex0, coord).x;
   coord.x += delta;
   tmp.w = texture2D(CaravelaTex0, coord).x;
   tmp0[i] += dot(tmp, const0);
   tmp1[i] += dot(tmp, const1);
   coord.x = caux.x;
 }
 // vertical direction
 result.x = dot(tmp0, const0);
 result.y = dot(tmp0, const1);
 result.z = dot(tmp1, const0);
 result.w = dot(tmp1, const1);
 // LL sub-band stream
 gl_FragData[0] = result;
 // LH, HL and HH sub-bands stream
 gl_FragData[1] = result;
}
```

(a) kernel program of flow -model



(b) pipeline -model for DWT

Figure 6. Flow-model and pipeline-model of 2D-DWT

Using this flow-model, the pipeline-model illustrated in Fig. 6(b) can be defined. Each stage in the pipeline-model consists of the flow-model in Fig. 6(a). As the number of level increases, the input and output data stream sizes become 1/4 of the previous level's. In the flow-model of level1, the input data stream becomes an ENTRANCE port of each pipeline-model. One of the output data streams of the flow-model is connected to the next stage (i.e. INTERMEDIATE port). The other one generates the sub-bands at the corresponding decomposition level, which is marked as EXIT port.

The pipeline-model is created by the meta-pipeline functions defined in section 4. After implementing the pipeline-model and by providing the image data to the ENTRANCE port, the Caravela runtime executes each stage when the input data becomes available. If it is performed in a local shader, each stage is assigned to the shader, and is replaced to the other stage automatically. When it is executed on remote worker servers, each flow-model is assigned to the worker, and waits for the input data that is propagated from the previous flow-model.

6 Conclusions

This paper is focused on a pipeline execution mechanism implemented by a set of flow-models in the Caravela platform, called meta-pipeline. Additional functions are defined in the Caravela library to support the pipeline processing. As shown in the application example section, pipeline-model can be transparently defined, and the execution is automatically controlled at runtime. In addition, the flow-model is a secured execution unit in a remote worker server. Therefore, we can conclude that the meta-pipeline mechanism can be proposed as a new processing style for stream-based computation on a distributed environment.

Acknowledgment

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

References

[1] 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.

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

[3] I. Daubechies. *Ten Lectures on Wavelets*. Number 61 in CBMS/NSF Series in Applied Math. 1992.

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

[5] 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.

[6] 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.

[7] Globus Alliance. http://www.globus.org/.

[8] GridLab Resource Management System http://www.gridlab.org/WorkPackages/wp-9/.

[9] W. Grosso. *Java RMI*. Oreilly Media, first edition, 2001.

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

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

[12] 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.

[13] 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, San Francisco, CA, second edition, 2003.

[14] S. Yamagiwa and L. Sousa. Design and Implementation of a Stream-based DistributedComputing Platform using Graphics Processing Units. In *ACM International Conference on Computing Frontiers*, May 2007.