In:
Editor:
ISBN:

Chapter 1

STREAM-BASED PARALLEL COMPUTING METHODOLOGY AND DEVELOPMENT ENVIRONMENT FOR HIGH PERFORMANCE MANYCORE ACCELERATORS

Shinichi Yamagiwa[1*]***, Gabriel Falcao***[2†]***, Koichi Wada***[3‡] ***and Leonel Sousa***[4§]
[1]Faculty of Engineering, Information and Systems,
University of Tsukuba, JAPAN
[2]Instituto de Telecomunicações, Department of Electrical
and Computer Engineering, University of Coimbra, Portugal
[3]Faculty of Engineering, Information and Systems,
University of Tsukuba, JAPAN
[4]INESC-ID, Instituto Superior Técnico,
Universidade de Lisboa, Portugal

PACS 07.05.Bx, 07.05.Wr.
Keywords: Stream-based Computing, Dataflow, Massively Parallel Computing, Graphics Processing Units, Manycore Processors, Caravela
Keywords: Stream-based Computing, Dataflow, Massively Parallel Computing, Graphics Processing Units, Manycore Processors, Caravela

AMS Subject Classification: 94-04, 94-02, 94C.

Abstract

The latest supercomputers incorporate a high number of compute units under the form of manycore accelerators. Such accelerators, like GPUs, have integrated processors where a massively high number of threads, in the order of thousands, execute concurrently. Compared to single-CPU throughput performance, they offer higher levels of parallelism. Therefore, they represent an indispensable technology in the new era

*E-mail address: yamagiwa@cs.tsukuba.ac.jp
†E-mail address: gff@uc.pt
‡E-mail address: wada@cs.tsukuba.ac.jp
§E-mail address: las@inesc-id.pt

of high performance supercomputing. However, the accelerator is equipped via the peripheral bus of the host CPU, which inevitably creates communication overheads when exchanging programs and data between the CPU and the accelerator. Also, we need to develop both programs to run on these processors that have distinct architectures. To make it simpler for the programmer to use the accelerator and exploit its potential throughput performance, this chapter describes the Caravela platform. Caravela provides a simple programming interface that overcomes the difficulty of developing and running parallel kernels not only on single but also on multiple manycore accelerators. This chapter describes parallel programming techniques and methods applied to a variety of research test case scenarios.

1. Introduction

Many of the advanced supercomputers listed in the Top500 utilize the huge performance enhancement provided by manycore accelerators such as GPUs. The application programmer targeting the use of a supercomputer needs to be able of performing hybrid programming, not only for the conventional parallel computing code development and optimizations on the CPU side, using for example MPI-based message passing techniques, but also for the parallelization of algorithms in the accelerators, using massively parallel programming environments like the stream-based computing OpenCL framework and language. The key to exploit the potential performance of supercomputers lies in programming techniques for heterogeneous systems, with the capacity of distributed processing nodes augmented with accelerators.

This chapter introduces *Caravela* [48][41][49], a novel programming environment for high performance computing using accelerators. Caravela virtualizes the interface for multiple accelerators and invokes the task with thousands of threads in parallel. The task unit of Caravela, denoted flow-model, includes the I/O data streams and the program code for the particular task. The program code is written in a stream-based programming language such as originally DirectX, OpenCL and CUDA. By connecting multiple flow-models, a large tasks can be organized.

This chapter introduces the programming environment using Caravela and presents the performance evaluation for massively systems with parallel accelerators. Caravela provides both a C-based programming library and a command line execution interface called *CarSh* [52].

Moreover, the chapter introduces I/O optimization techniques when the recursive or iterative flow-models are included in the processing task, designated as swap method [51][25]. It is extended to a pipelined execution model using multiple accelerators organized in a meta-pipeline structure [50]. By using the swap method, this chapter also introduces a new execution mechanism for accelerators called the scenario-based execution [46]. It is capable of executing multiple flow-models of recursive or iterative nature on the accelerator side without burdening the CPU with unnecessary communications. Therefore, it invokes multiple flow-models without introducing communication overheads among the accelerator and GPUs, and thus achieves very high performance.

Regarding the application of the referred techniques on Caravela, the chapter reports performance evaluations using simple applications and kernels. Finally, the chapter proposes future work and the expected direction of these technologies in high performance

stream-based computing for supercomputers.

1.1. Stream Computing

Multicore/manycore architectures have gained traction due to the saturation of Moore's law related to the growth of transistor count per silicon platform. The manycore architecture is organized by graphics processing demands that need fast computations to achieve a high frequency framing on dynamic graphics, especially for entertainment markets [32]. It requires the concurrent processing of multiple pixels computed by hundreds of processing units. It also exploits the potential fine-grained parallelism from application programs [36]. Thus, the Graphics Processing Unit (GPU) has become an indispensable device to implement a high performance computing platforms.

In the GPUs, each processing unit identifies its target computing element regarding a processor index. For example, assuming a vector summation $r_c = r_a + r_b$, a programmer needs to consider that the calculation is separated into each element of vectors like $r_c[id] = r_a[id] + r_b[id]$, where id represents the index of the vector element. Each calculation of $r_c[id]$ is assigned to a compute unit, thus the summations of elements in the vector are performed in parallel. Optimistically, the vector summation needs only the processing time to calculate the "+" operation when the number of computing units is larger than the length of r_c. Thus, a programmer certainly needs to consider the indexing of computing elements and also independent processing for each computing element assigned to a unit. Because of the processing style based on the indexed processors and the continuous streaming of data, it is called *stream computing*.

An accelerator in the manycore architecture works as a co-processor of a host CPU connected via a peripheral bus. Therefore, for programming the accelerator, the host CPU is indispensable and controls the configuration and behavior of the accelerator. Thus, the programmer must write both computing programs on the accelerator side and the controlling program on the host CPU side unavoidably. To reduce the difficulty of such double programming situation, there are available programming languages and the respective runtimes. For example, the recent standard ones are NVIDIA's CUDA [33] and OpenCL [31].

1.2. Computing Power of Graphics Processing Units

Graphical applications, especially 3D graphics visualization techniques, have drastically advanced in this decade. Even a commodity personal computer now processes very high quality graphics processed in real-time. This is mainly due to the GPU connected to the personal computer. The power of GPUs is growing drastically: for example, floating-point processing performance on 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 that demand high workloads.

Nowadays, researchers of high performance computing are also focusing their efforts on the performance of GPUs, and investigating the possibility for its usage as a substitute of CPUs. For example, GPGPU (General-purpose processing on GPU) allows achieving a high level of performance [24] [29]. A cluster-based approach using PCs with high performance GPUs has been reported in [9]. Moreover, compiler-oriented support for GPU resources has also been proposed [6].

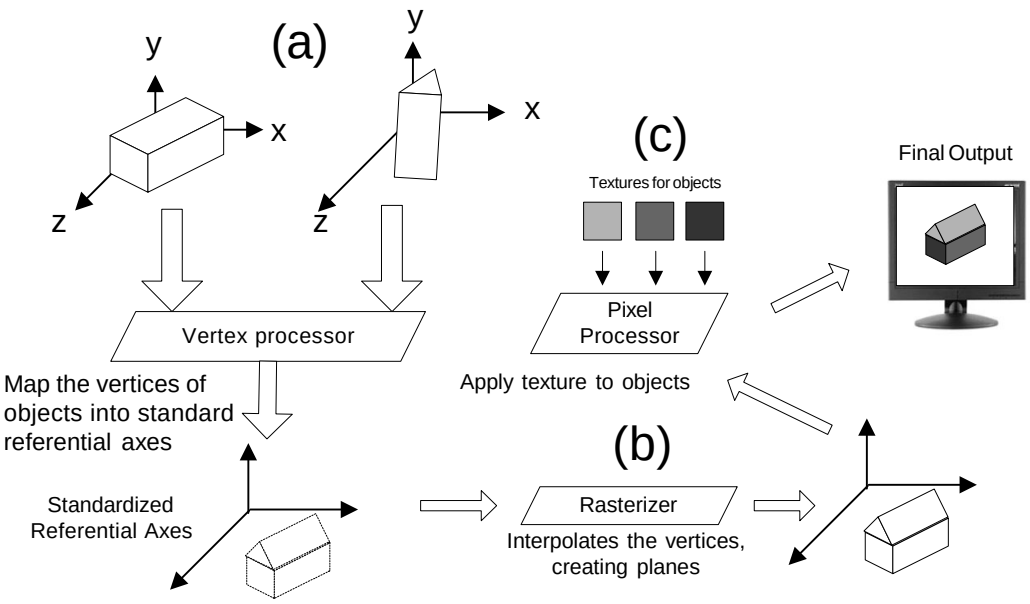


Figure 1. Processing steps for graphics rendering.

1.3. The Conventional Graphics Processing on GPU

This subsection introduces the processing steps and architecture that allows GPUs to generate graphics objects in order to display them on a screen. The GPU acts as a co-processor of the CPU via a peripheral bus, such as the AGP or the PCI Express. A Video RAM (VRAM) is connected to the GPU, which reads/writes graphics objects from/to the VRAM. The CPU controls the GPU operation by sending the object data to the VRAM and a program to the GPU.

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

1.4. General-purpose Computing on Graphics Processing Units

In recent GPUs, the vertex and pixel processors are programmable. The designers of graphics scenes can make programs for the processors specific for the desired graphics effects. It

is very important that the programs run fast in order to achieve a huge number of processed frames per second. Therefore, 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 cannot be controlled. Moreover, the output data from the rasterizer is just sent to the pixel processor and cannot be fetched by the 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 chapter is not restricted to the performance of GPUs, but also addresses 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 elements. 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 standard languages such as DirectX assembly language, High Level Shader Language (HLSL) [1] and OpenGL Shading Language [23]. Thus, the program can run on any GPU connected to any computer.

According to the discussion above, it can be concluded that 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 nature. Thus, we aim to develop an execution mechanism on the GRID environment based on stream-based computing using GPUs' power.

2. Caravela: A Stream Computing Environment

2.1. Preface

Worldwide distributed computing has become one of the remarkable possible ways to use anonymous processing 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 [2], or a mobile agent-based one that migrates among the resources organized as a virtual network [12]. According to the research reports of those platforms [5, 20], worldwide distributed computing is effective in achieving an ultra computing power, by taking advantage of huge amounts of unused computing power from all over the world. This computing style is named GRID computing [40].

For a GRID computing platform, one of the 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 section describes the design and implementation of the *Caravela* platform 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 [35]. Therefore, this section also shows an implementation of the Caravela platform which maps the flow-model on a GPU. Moreover, this section describes an example where flow-model units execute distributed on remote GPU resources.

2.2. 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 [2], a well known message passing-based platform for GRID computing. The users of the Globus platform can write programs as MPI-based parallel applications [22]. 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 [12]. 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 contributors 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 [11].

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 of 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 [3].

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 [16] 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) [27]. 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.

2.3. 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 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.

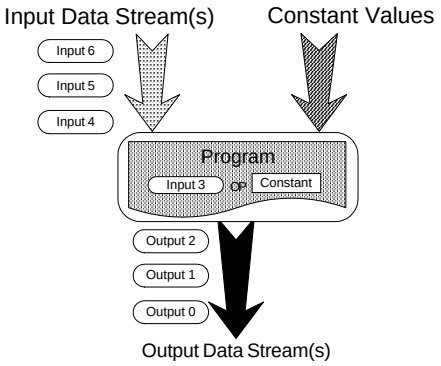


Figure 2. Structure of the flow-model.

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

2.4. 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.

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.

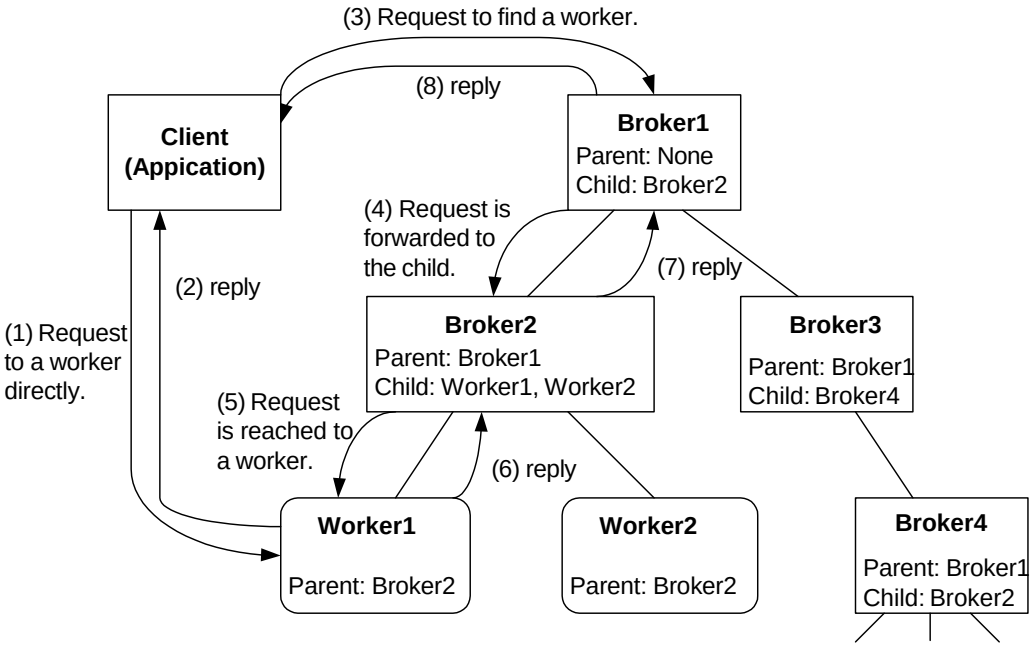


Figure 3. Caravela network example.

> 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 child 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.

2.5. 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 the 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. The application in the Caravela platform is able to execute flow-models in a processing unit through secure execution mechanisms.

2.6. 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.

2.6.1. 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 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.

2.6.2. Applying the GPU to a processing unit

The application in the Caravela platform is supported by the Caravela runtime environment referred in section 2.4., 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 2.5.. 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.

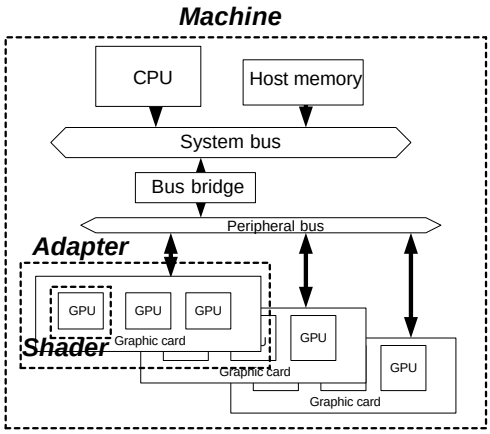


Figure 4. Resource hierarchy in a processing unit.

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.

2.6.3. The Caravela library

To control the flow-model execution, implicitly controlling the GPU, the application uses the Caravela library functions programmed in C language as listed in Table 1.

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 the 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

Table 1. Basic functions of Caravela library

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

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.

2.6.4. 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.

2.7. 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 in the coefficient matrix, where common dimensions are 3x3 or 7x7. Here, we illustrate the programming of a filter with a 3x3 coefficient matrix:

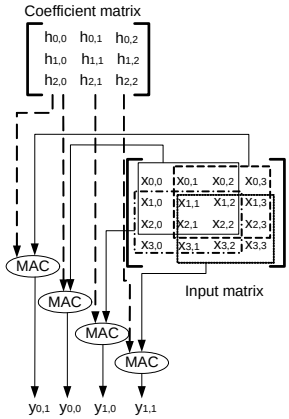
$$y_{k,l} = \sum_{i=0}^{2} \sum_{j=0}^{2} h_{i,j} x_{k+i,l+j} \tag{1}$$

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.

Table 2. 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



(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 *
        oC0.y += (input_row0.y * c[0][i] + input_row0.z * c[1][i] + input_row0.w
        oC0.z += (input_row0.z * c[0][i] + input_row0.w * c[1][i] + input_row1.x
        oC0.w += (input_row0.w * c[0][i] + input_row1.x * c[1][i] + input_row1.y
        coord.x= t0.x;
  }
```

(b) Program embedded in flow-model

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

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 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 pro-

cessing.

2.7.1. 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.

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

2.7.3. 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 2, 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.

```
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.

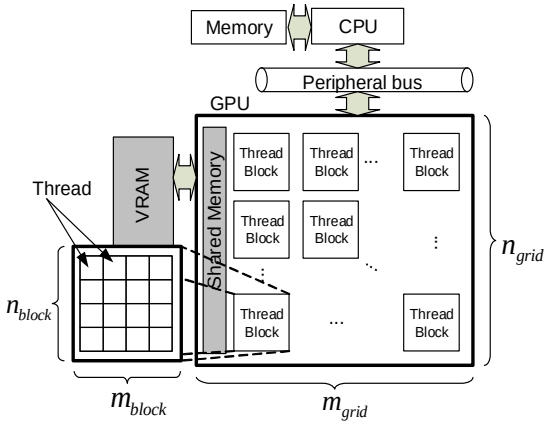
The calculation time on the Caravela platform is 10.6 s, which compares with 23.5 s on the CPU-based version. The Caravela platform achieves about 2.2 times higher performance than the host CPU. Thus, we can conclude that, by providing a secure environment and a transparent interface for programmers 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.

3. Commandline Based Execution of Flow-models

3.1. Stream-oriented Programming Environment

To reduce the difficulty of the double programming situation, there exist programming languages and the runtimes. The recent de fact standard ones are NVIDIA's CUDA [33] and OpenCL [34][31][47].

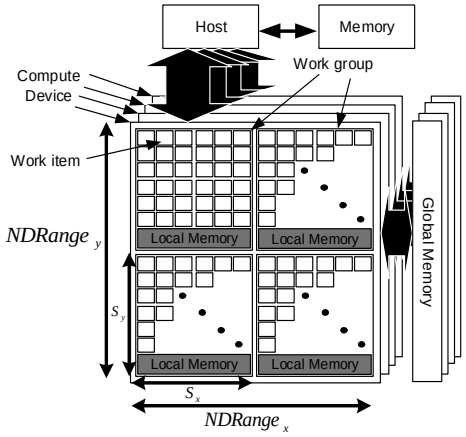
The CUDA assumes an architecture model as illustrated in Figure 7 (a). The model defines a GPU which is connected to a CPU's peripheral bus. A VRAM maintains data used for calculation on the GPU. The kernel program is downloaded by the host CPU to GPU and the data is also copied from the host memory. The program is executed as a thread in a



(a) Execution model of kernel threads in CUDA

```
int main(){
  float *A, *B, *C;  n_block
  ...          m_grid  n_grid
  dim3 dimBlock(      ,      );
  dim3 dimGrid(       ,      );
  KernelFunc  <<< dimGrid , dimBlock >>>(A, B, C);
  ...
}

__global__ void KernelFunc (float *a, float *b, float *c){
  int i = blockDim.x * blockDim.y * (gridDim.x * blockIdx.y
            + blockIdx.x) + threadIdx.x;
  c[i] = a[i] + b[i];
}
```

(b) Example CUDA code for array summation



(c) Platform model of OpenCL

```
...
hContext = clCreateContextFromType (···,
CL_DEVICE_TYPE_GPU,···);
...
hProgram = clCreateProgramWithSource (···, sProgramSource, ···);
clBuildProgram(hProgram, ···);
hKernel = clCreateKernel (hProgram, "VectorAdd", 0);
...
float * pA = new float [···];
... initialize pA array ...
hDeviceMemA = clCreatebuffer(hContext, ..., pA, ...);
...
clSetKernelArg(hKernel, ..., hDeviceMemA);
---
clEnqueueNDRangeKernel(···, hKernel, ···);
clEnqueueReadBuffer(···, hDeveiceMemC, ···, pC, ···);
...
char sProgramSource = "
__kernel void VectorAdd(
  __global const float *a, __global const float *b,
  __gloabal float *c,  int iNumElements)
{
        int iGID = get_global_id(0);
        if(iGID >= iNumElements) return;
        c[iGID] = a[iGID] + b[iGID];
}";
```

(d) Example array summation in OpenCL

Figure 7. CUDA and OpenCL architecture models and its vector summation examples.

thread block grouped with multiple threads. The thread blocks are tiled in a matrix of from one to three dimensions. In the figure, thread blocks are tiled in two dimensions which size is $n_{grid} \times m_{grid}$. Each thread block consists of $n_{block} \times m_{block}$ threads. The program shown in Figure 7 (b) is an example of vector summation written by using CUDA. The kernel program is defined with the `__global__` directive so that it is executed on GPU. In the function, the global variables named `gridDim`, `blockDim`, `blockIdx`, `threadIdx`, implicitly declared by the CUDA runtime, are available to be used to specify the size of the grid and the thread block, the indices of the thread block and of the thread respectively. The function is called by the host CPU program specifying the number of threads with `<<< >>>`.

The OpenCL defines a common platform model that includes the processing element and the memory hierarchy. Figure 7(c) illustrates the platform model for the processing

element. The host CPU is connected to the *OpenCL device*. The OpenCL device consists of the individual processing element called *compute unit*. The compute unit includes one or more *work groups* that include the *work items*. The work item is identified by the unique ID and processes the corresponding result using the related input data associated by the ID. The total number of work items is given by the program using a parameter called *NDRange* that can be defined in from one to three dimensions. The example in the figure includes $NDRange_x \times NDRange_y$ work items. The OpenCL program is written in C as the host CPU side shown in Figure 7(d). The resources in the OpenCL are obtained by the *context* created by the runtime function. In the figure, the context for a GPU is defined by specifying `CL_DEVICE_TYPE_GPU` as its argument. The argument is variable to select different types of accelerators. The kernel program is provided by a source string defined as an array of char. The string is passed to the runtime functions to compile and prepare the executable code in the accelerator. The buffers for I/O data streams are allocated using the conventional functions such as "new" or "malloc" in the CPU side. The programmer can select if the buffers are accessed directly from the accelerator or if the buffer mirrors are allocated in the accelerator's memory. And then, the argument pointers of the kernel function that point to the actual buffers in the host and/or in the compute device are passed to the accelerator. Finally, the kernel is executed by the `clEnqueueNDRangeKernel` function and the output data streams in the accelerator's memory are copied to the host CPU side.

The assumed architectures in CUDA and OpenCL are very similar. The major difference between those is the kernel compilation mechanism. The CUDA program is compiled whole code including the kernel function by using the *nvcc* compiler. Then the executable for the host CPU downloads the program implicitly to the GPU. Besides, the OpenCL one passes the source string of the kernel code to the runtime function. To separate the kernel code in CUDA as performed by OpenCL, *nvcc* can output the assembly language (PTX) of the kernel code. The assembly language code is also able to be loaded by a runtime function of CUDA. To keep the code compatibility among both runtimes, it is important to develop a unified interface for the accelerator that loads and executes the kernel code without developing the host side program using different runtime functions.

The stream-based computing environment provided by the accelerators brings a powerful parallel processing environment due to the exploited concurrency implicitly from the program. Moreover the runtime environments such as OpenCL and CUDA let the programmer easily develop the application on the accelerators. However, the programmer needs to design and implement both the host CPU program and the kernel program for the accelerator. Caravela platform provides a model-based execution mechanism using flow-model. However, it still has the duty for the double programming separating the CPU program using the Caravela library and the flow-model with the kernel program. So that the programmer concentrates to write the kernel program on the accelerator, it is indispensable to invent a new programming environment that avoids to the development the host CPU program. As the related work, *barracuda* [4] on ruby extension for OpenCL provides wrapper methods for hiding the difficulty of the host CPU program. And StreamIt [43] provides a unified language to resolve the double programming difficulty at compiling time. However those also need to code the scenario for the kernel execution timings. If the programmer needs to invoke multiple kernel programs concurrently, those programs must include a detailed schedule for multi-thread execution on the CPU side. Therefore, it is important to develop

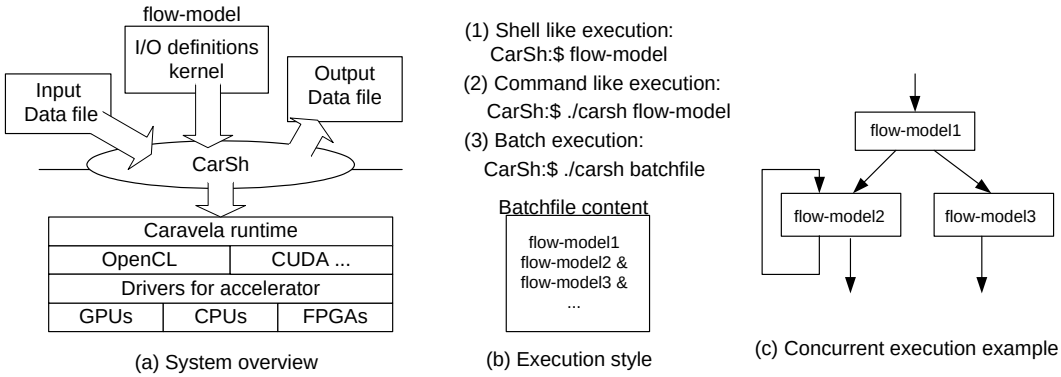


Figure 8. CarSh system overview.

a kernel execution environment for intuitive programming of complex applications utilizing accelerator's power, where the programmer only considers the kernel program, the I/O kernels and parallelization parameters. And then the execution timings should be automatically and implicitly decided at runtime. In this section, using the flow-model definition to resolve the problem, we propose a novel shell-like and commandline-based programming environment for the stream-based accelerators called *CarSh*.

3.2. Design of CarSh

When we execute any kind of commands in a conventional CPU-based system, we often use a shell such as *bash*. We just rely on the execution timings and managements of the processes such as background/foreground execution. We consider the same execution mechanism on the stream-based programs for accelerators.

Figure 8(a) shows the system overview of CarSh. CarSh receives flow-models and the corresponding input data files. After the execution of the flow-model the output data are also stored into files. As shown in Figure 8(b) CarSh executes flow-models directly (1) from the commandline of CarSh like a shell prompt, (2) from an argument of CarSh and (3) from a batch file of an execution schedule of multiple flow-models. The batch file implements a pipeline execution of multiple flow-models, for example, shown in Figure 8(c). The data I/O for the pipeline execution are provided by files or internally allocated buffers in CarSh. In the pipeline, *flowmodel2* and *flowmodel3* can be invoked simultaneously. Here, CarSh needs a background concurrent execution mechanism of the flow-models. If a flow-model defines the recursive I/O using the swap pair, CarSh needs automatically to detect the property and executes the iteration for the swap pair. According to the consideration for the interface, we introduce the design considerations below for CarSh.

3.3. Management of Flow-model Execution

CarSh needs to execute a flow-model automatically detecting the definitions and execute it over the Caravela framework. It needs to define an executable format for the execution. The format includes 1) I/O buffer definition with the data type, 2) flow-model XML file, 3) target lower level runtime such as OpenCL, 4) optional functionality definitions such

as the *swap function*. CarSh prepares the input data from the I/O definitions, executes the flow-model applying the optional functions and finally saves the result of the output data from the flow-model. This scenario is packed in a single executable format and passed it to CarSh.

In addition to the single execution of a flow-model, CarSh needs a batch execution mode for supporting the pipeline execution of multiple flow-models as shown in Figure 8(c). The batch execution follows a formatted scenario with the execution steps of the flow-models. To implement this batch execution, CarSh needs a background execution mode and a synchronization function for the previous flow-model executions. For example, to invoke the pipeline in the figure, CarSh executes *flowmodel1* and wait the execution. After that, *flowmodel2* and *flowmodel3* can be concurrently executed as the background tasks. CarSh also needs to have a synchronization function to wait for finishing previously executed flow-models.

The flow-model does not define any behavior regarding iterative execution. Moreover, a set of flow-model (i.e. batch) execution must have possibility to be repeated as it would get the results after specified steps such as the LU decomposition algorithm. Therefore, CarSh needs a function to repeat execution of a flow-model or a batch scenario for a specified time.

3.4. Management of I/O Data Streams

The flow-model execution inputs data and after execution completes it saves the output data. CarSh introduces two mechanisms for saving/restoring the I/O data. One is a simple mechanism reading/writing the input/output data from a file. In this case, CarSh sets the data read from the file to the corresponding input buffer and uses it for the flow-model execution. The output data are also saved into a file that can be used as the input again. Thus, through one flow-model execution to another in a pipeline structure the files are passed and received. Another mechanism is *virtual buffer* that works as a virtual space for I/O data provided by CarSh inside. The virtual buffer is first prepared by CarSh before the flow-model execution. During the execution, it is used as the place for saving the I/O data of flow-models. The content data of the virtual buffer can be loaded from a file or saved to a file. To manage the virtual buffer, CarSh needs the management functions for *creating*, *deleting* the buffer, and also functions for *filling* and *dumping* data in the buffer from/to files.

According to the functions for flow-model execution and the I/O data, CarSh will provide a shell-like stream computing environment just giving the kernel programs and the execution scenario is packed into the executable or the batch. During execution of the scenario, CarSh fulfills the input data from files or the virtual buffer and passes the execution result to the next file or the virtual buffer. The next flow-model can read the result from the buffer to continue the execution. Thus, the flow-model execution conveys data from one buffer to the next ones. CarSh provides also the iteration of flow-model or batch. Thus, the programmer who uses CarSh does not consider the host CPU program at all. He/she finally becomes available to focus on designing just the kernel programs and the dataflow scenario.

```
<?xml version="1.0" encoding="ASCII"?>
<CarshEx>
 <ModelFile>Flow-model XML file name</ModelFile>
 <Input>
  <Name>Input valuable name in kernel function</Name>
  <DataFile>CSV file name</DataFile>
 </Input>
 <Input>
  ....
 </Input>
 <Output>
  <Name>Output valuable name in kernel function</Name>
  <DataFile>CSV file name</DataFile>
 </Output>
 <SwapPair>
  <Input>Input valuable name in kernel function</Input>
  <Output>Output valuable name in kernel function</Output>
  <NumSwap>Number of swaps</NumSwap>
 </SwapPair>
</CarshEx>
```

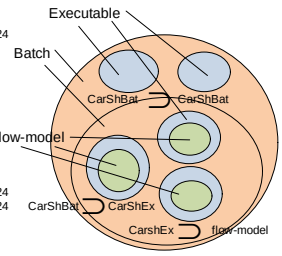
(a) CarSh executable format in XML

```
<?xml version="1.0" encoding="ASCII"?>
<CarshBat>
virtbuf create int 1024 buf_a
virtbuf create int 1024 buf_b
virtbuf fill sample 1_a.csv buf_a
virtbuf fill sample 1_b.csv buf_b
virtbuf create int 1024 buf_c
sample_virt &amp;amp;
sync
repeat 10 sample_virt.xml
sync
virtbuf dump c_tmp.csv buf_c
virtbuf swap buf_a buf_c
virtbuf delete buf_a
virtbuf delete buf_b
virtbuf delete buf_c
exit
</CarshBat>
```

(b) CarSh batch format in XML

```
$ ./carsh
Starting CarSh (Caravela Shell) ...
CarSh: $ virtbuf create int 1024 buf_a
CarSh: $ virtbuf list
1: Name=buf_a, DataType=int, Length=1024
CarSh: $ virtbuf create int 1024 buf_b
CarSh: $ virtbuf fill sample1_a.csv buf_a
CarSh: $ virtbuf fill sample1_b.csv buf_b
CarSh: $ virtbuf create int 1024 buf_c
CarSh: $ sample_virt &
CarSh: $ sync
CarSh: $ repeat 10 sample_virt.xml
CarSh: $ virtbuf dump c_tmp.csv buf_c
CarSh: $ virtbuf delete buf_a
CarSh: $ virtbuf list
1: Name=buf_c, DataType=int, Length=1024
2: Name=buf_b, DataType=int, Length=1024
CarSh: $ virtbuf delete buf_b
CarSh: $ virtbuf delete buf_c
CarSh: $ exit
```

(c) An example of commandline executions



(d) relations among executable, batch and flow-model

Figure 9. Implementation of CarSh.

3.5. Implementation of CarSh

Our first implementation of CarSh employs *process*-based task management using *fork* system call on Linux environment. Thread-based implementation is also possible and would achieve better performance. A process is assigned to each flow-model execution by giving an executable XML file as shown in Figure 9(a) to CarSh commandline. When '&' is added in the last of the commandline, the process is executed in background. This implements the concurrent execution of multiple flow-models. For the synchronization of one or more process executions our implementation introduces *sync* command to wait all the process execution including the background processes using `waitpid` system call. The batch scenario is written in an XML file with `<CarshBat>` tag. Given in the commandline in CarSh, the batch XML file includes the steps of the scenario like the example of Figure 9(b).

I/O data for the flow-model are loaded and saved from/to CSV files. The file is directly assigned by `<DataFile>` tag in `<Input>` and `<Output>` tags. The `<DataFile>` tag accepts also the virtual buffer name as the input for the flow-model. The I/O arguments of the kernel code in the flow-model are linked in the executable file of CarSh. Thus, the virtual buffers are connected to the I/O in the flow-model. This means that the I/O data inputted/outputted to/from accelerator will be passed to/from the CSV files.

The virtual buffer is implemented by the POSIX shared memory object. The first process of CarSh (here calls this process *master*) creates the shared objects that correspond to the virtual buffers using `shm_open` system call and the buffer sizes are truncated by `ftrancate` system call. The master process folks other children processes of flow-models that open the shred memory objects, and then the children processes use the shared memory object as the I/O buffers. The content of the shared object is saved into a file in CSV format. While the flow-model execution processes are working, the virtual buffers are never removed because those are used for the flow-model execution. Then the master process can delete the buffers using `shm_unlink` system call. Thus the shared object is created and deleted by the master process.

Figure 9(c) shows an example of CarSh commandline execution. First the *virtbuf* command manages the virtual buffers with the argument *list* that shows all allocated virtual buffers and the argument *create* that allocates a virual buffer. After executing an executable or a batch file named *sample_virt.xml* in the background it synchronizes using *sync* com-

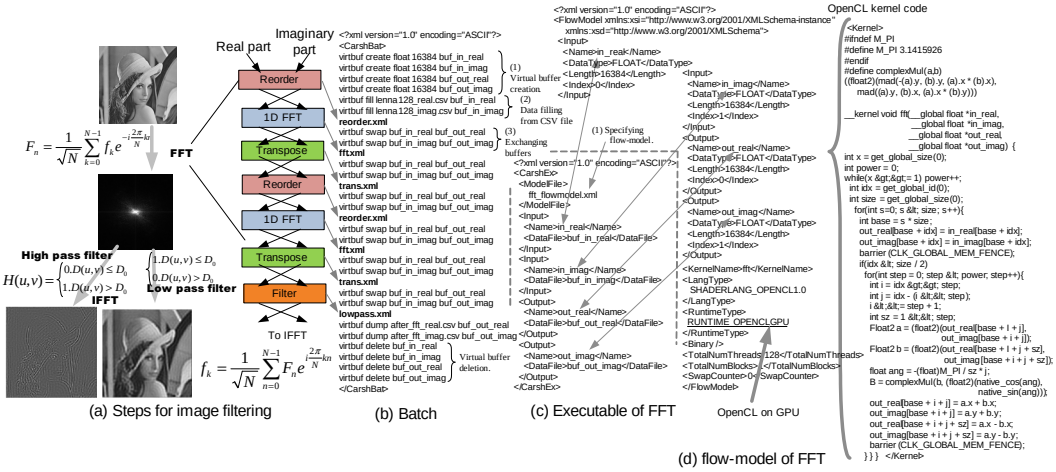


Figure 10. Implementation of image filtering using FFT on CarSh.

mand. *repeat* performs the iterative execution of the same XML file for 10 times. The *virtbuf* command can save the buffer contents to files using *dump* argument. The *swap* argument of the command exchanges buffer names. Finally the *delete* argument deletes all the allocated virtual buffers.

The inclusion relations among the flow-model, the executable and the batch are illustrated in Figure 9(d). The executable only includes the flow-model directly. The batch includes the executables and the other batches. If a programmer needs to repeat execution of a processing pipeline with flow-models, he/she can pack the pipeline to a batch and then prepares another batch that includes the batch. Therefore, by nesting the executable and the batch, we can implement any combination of processing flows.

3.6. Evaluations

Let us explain evaluations of CarSh focusing on performance and programmability. Here we use a typical image filtering using 2D FFT performed often in the image operations. Figure 10(a) shows the processing steps used in the evaluation. An image (*Lena*), which is transformed by the FFT, is passed to a high or low pass filter, and finally transposed by IFFT. This simple process is composed by five flow-models: *reorder* performs butterfly exchanges, *transpose* inverses the rows and the columns, *filter*, *FFT and IFFT*. the whole calculation is defined by a CarSh batch XML file listed in Figure 10(b). It uses virtual buffers for the real and imaginary parts inputted/outputted to/from the subsequent processes managed by *virtbuf* command such as Figure 10(b)-(1)(2). After every flow-model execution, those buffers are exchanged such as Figure 10(b)-(3). After the executions of the flow-models, the virtual buffers are deleted. Here, each flow-model execution is called from the CarSh executable XML file, for example, FFT shown in Figure 10(c). The I/O data for initialization/resulting values are passed via the virtual buffers. The flow-model of the FFT is shown in Figure 10(d). The I/O arguments of the kernel program correspond to the real and imaginary parts. Those match each other among the executable and the flow-model. Finally, CarSh will execute the batch XML file to get a filtered image.

Table 3. Performance of Image filtering using FFT on CarSh over OpenCL runtime.

Image size (GPU)	FFT	Filter	IFFT	Total (sec)	Image size (CPU)	FFT	Filter	IFFT	Total (sec)
128^2	0.766	0.128	0.764	1.658	128^2	1.969	0.281	1.971	4.221
256^2	0.78	0.128	0.779	1.687	256^2	2.002	0.266	1.999	4.267
512^2	0.825	0.135	0.826	1.786	512^2	2.029	0.302	2.032	4.363
1024^2	0.985	0.153	0.988	2.126	1024^2	2.216	0.329	2.218	4.763

3.6.1. Programmability of CarSh

This example is a typical pipelined application with multiple flow-models. The pipeline is organized by passing buffers from one kernel program to the next. This must be implemented by buffer management functions on the host CPU side if we apply the conventional double programming method using the OpenCL runtime for accelerators. Moreover, CarSh provides the virtual buffer. It is easy for the programmer to allocate buffers used for the I/O data from/to the flow-model. Thus, buffers are easily defined in the batch XML file by specifying the names. Those are fulfilled from CSV files to the buffer, and simply passed via the names of the buffer among flow-models in the executables.

Without considering the timings for kernel executions and also the buffer management among the host CPU and the accelerator, the programmer can perform a straightforward programming using CarSh framework. The programmer is able to focus on brushing up the stream-based algorithm written in the kernel program. Moreover, due to the mode the code for CarSh is written, it is very highly portable among different combinations of a host CPU and an accelerator. This enables a remote development environment where the code compatibility is guaranteed. For example, when we use different kinds of accelerators over OpenCL, we can just change `RuntimeType` tag in flow-model as shown in Figure 10(d). This mechanism makes the performance check much easy as we just change the string in the flow-model XML description.

3.6.2. Performance of CarSh

We have measured performances of the image filtering example with varying the accelerator types and the image sizes. Our platform of the performance test is a PC with a Core i7 930 2.80GHz and an Nvidia Tesla M2050 GPU. Table 3 shows the comparison among the GPU-based and the CPU-based executions of the image filtering batch file on CarSh. Both executions use the same kernel functions and the CarSh related XML files. The number of parallelism in OpenCL is set to 1024 where the OpenCL runtime distributes 1024 concurrent threads on the GPU and CPU. The CPU-based execution is performed on the Intel OpenCL runtime using the multiple cores of the CPU. According to the performances listed in the table, the GPU-based performance almost duplicates that of the CPU-based one. This implies that we can control the performance of a set of CarSh executable XMLs. Therefore, if we introduce a new powerful accelerator, we can easily upgrade the performance changing the runtime type description in the flow-model.

As we explained in this section, CarSh brings a simple and transparent programming style for the high programmability on the stream computing employing XML-based pack-

aging for the kernel function invoked in the accelerator. It is easy to control the performance by changing the runtime type description defined in the flow-model. Thus, we have confirmed that CarSh overcomes the programming complexity on the current stream-based accelerators that enforces the double programming. Moreover, CarSh provides the novel commandline interface for executing the kernel function. This promotes high productivity of programs on manycore architectures.

4. Parallel Execution of Multiple Flow-models

4.1. CaravelaMPI

4.1.1. Parallel Computation with MPI

There are two major paradigms for programming and implementing parallel applications: *i*) *message passing paradigm* that exchanges messages between processes that run concurrently; and *ii*) *shared memory paradigm* that shares data between processes through physical, or virtual, shared memory. Interfaces and libraries such as the Parallel Virtual Machine (PVM) [13] and the MPI [44] have been developed to support parallel programming based on message passing. The OpenMP [19] has been designed to program shared memory systems. This section is focused on parallel programming based on the message passing paradigm, namely the MPI.

To program an efficient parallel application based on message passing, the programmer has to consider both the computation and communication aspects of processing algorithms and systems. Therefore, when performance of parallel applications running on an MPI-based cluster has to be improved, two main approaches can be adopted: *i*) upgrading the system, namely upgrading the processor hardware, increasing the amount of memory and upgrading the network platform (not only at the hardware level but also in the software layers [21]) whenever parallel applications require intensive communication; and *ii*) redesigning algorithms to exploit further parallelism in applications, increasing computation instead of communication time, or by overlapping these two parts. In the latter approach, programmers must try to find parts of application where communication can be overlapped with computation, using MPI asynchronous communication functions such as `MPI_Isend` and `MPI_Irecv`. These functions do not block the subsequent computation and implicitly send/receive messages to/from the receivers/senders. For instance, as illustrated in Figure 11, the `MPI_Irecv()` function in Proc0 executes its communication implicitly while computations are performed. After the computation is finished, the communication is synchronized by using the `MPI_Wait()` function. This mechanism works with a communication request data structure that is received from the communication function (i.e. `MPI_Irecv()`) and is passed into the synchronization function (`MPI_Wait()`). According to the implicit communication operation, the communication time can be concealed by the computation time and consequently the processing efficiency increases. Thus, it is very important to try to overlap communication with computation when using message passing paradigms, so that performance of entire system can be increased without having to invest more powerful hardware resources.

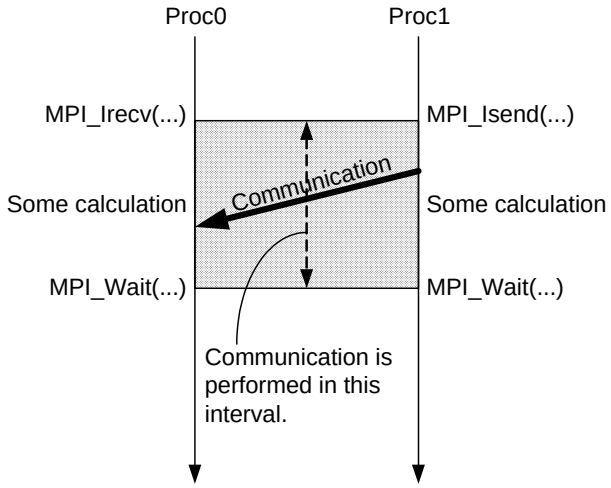


Figure 11. Overlapping communication with computation.

4.1.2. Discussion

With the huge computing capacity of GPUs it is likely that GPU-based clusters will become important to design high performance computing platforms. The experience of building a GPU cluster [9] has already been reported as a breakthrough in cluster computing platforms. Although the parallel algorithms originally devised for conventional cluster computers will have to be reworked for GPU-based clusters, it can be quite advantageous in terms of gains, if we are able to exploit the huge computing power provided by GPU clusters.

In spite of such exciting forecast, the programming of GPU-based clusters has to become feasible, and a de facto standard Application Programming Interfaces (API) will have to be developed in order to make usage of those parallel systems accessible while achieving high processing efficiency. Programming on such raw system is a very complex task, which requires mixing GPU control functions provided by graphics runtime with MPI functions executed by CPU under control of operating system.

Herein, we propose a unified library named *CaravelaMPI* that encapsulates communication and GPU control procedures and functions. CaravelaMPI combines MPI and Caravela library, hiding the details of the graphics runtime environment and allowing the programmer to concentrate himself in a goal of achieving an efficient parallel implementation for his application.

Design and implementation aspects of the CaravelaMPI and associated software library will be presented and discussed in the next section.

4.1.3. Design of CaravelaMPI

Parallel processing based on message passing paradigm has three different stages: message reception, computation and message transmission. Exact timing for receiving messages is usually statically unpredictable, since it is dependent on characteristics of systems and of running programs. Programmer uses asynchronous receiving function to reserve data recep-

tion and puts other calculation in waiting time for the reception. For the transmission after the calculation, the same technique can be used to send result data applying asynchronous transmission function, without synchronizing to finish data transmission in network hardware.

According to the patterns between communications and calculations in message passing of parallel application mentioned above, we propose the following interface models in CaravelaMPI: *i*) *Recv-Calculate-Send*; *ii*) *Recv-Calculate*; *iii*) *Calculate-Send*; and *iv*) *Synchronization for asynchronous communication*.

Recv-Calculate-Send interface model provides a mechanism to receive I/O data streams as messages from remote processes, to execute a flow-model using the messages and to send results from the flow-model execution to remote processes. This interface model is provided with both synchronous and asynchronous versions. In the synchronous version, this interface model blocks the subsequent program operations until arrival of input data messages, where the flow-model execution resumes and the transmission of the output data is performed. On the other hand, if the calling mode is asynchronous, this interface model does not block the subsequent operations. Therefore, the asynchronous version of this interface model just reserves communications and computation with flow-model, issuing a request. To confirm the request completion, programmer uses the synchronization interface model that will be explained later.

Recv-Calculate interface model receives I/O data messages for a flow-model and executes it using the received data. When an application needs to perform both reception and calculation, such as the case when it must gather all results from other processes and calculate the final result, programmer can use this kind of interface model. This interface has also two versions: synchronous and asynchronous. The synchronous version blocks the subsequent operations until the flow-model execution has been completed, while the asynchronous version does not block, issuing a request. To check the request completion, the synchronization interface model mentioned later is invoked.

Calculate-Send interface model corresponds to complementary part of the Recv-Calculate-Send interface model. The flow-model execution is immediately performed after this interface model is invoked. This interface model has synchronous version and asynchronous one. For sending output data streams to other processes, the former one performs the flow-model execution and the communication without returning from the corresponding interface function. The latter version returns immediately only issuing a request. Therefore, to confirm completion of the request, synchronization interface model must be invoked.

Finally, in addition to MPI functions for synchronization such as `MPI_Wait()` or `MPI_Test()`, a synchronization interface model is required for managing the completion of communication and calculations performed by the interface models proposed above. This interface blocks all the subsequent calculations until the corresponding request has been completed.

In all models, except the synchronization one, to perform recursive execution of the flow-model, the interfaces can accept the I/O pair data structure for the buffer swapping functions in Caravela, and also the number of iterations. This allows to loop a flow-model for a given number of times, while exchanging the I/O data streams specified in the swap I/O pairs.

By using the proposed interface models, application can thus efficiently implement

Table 4. CaravelaMPI functions

Management functions
`CaravelaMPI_Init(...)`
Initialization of CaravelaMPI environment.
`CaravelaMPI_Finalize(...)`
finalization of CaravelaMPI environment.
Synchronous communication functions
`CaravelaMPI_Recvsend(...)`
Synchronous Recv-Calculate-Send interface.
`CaravelaMPI_Recv(...)`
Synchronous Recv-Calculate interface.
`CaravelaMPI_Send(...)`
Synchronous Calculate-Send interface.
Asynchronous communication functions
`CaravelaMPI_Irecvsend(...)`
Asynchronous Recv-Calculate-Send interface.
`CaravelaMPI_Irecv(...)`
Asynchronous Recv-Calculate interface.
`CaravelaMPI_Isend(...)`
Asynchronous Calculate-Send interface.
Synchronization functions
`CaravelaMPI_Wait(...)`
Synchronization interface to complete a request.
`CaravelaMPI_Test(...)`
Synchronization interface to check a request completion.

communications and parallel execution with the GPU's resources. When programmer tries to overshadow GPU calculations with communications, the interfaces can be moved to the other timings. During the migrations of communication timings, calculations on GPUs will be moved together because the GPU calculations are encapsulated into the interface models using the flow-model framework. Moreover, because the completion of GPU calculations can be checked by the synchronization interface model, flexible control for execution timings on GPUs becomes available, considering communication timings simultaneously. Thus, the interface disparity between communications and program execution on GPU discussed in the previous section will be addressed by the CaravelaMPI, and also the performance tuning regarding communication timing will cause an speedup of parallel applications.

4.1.4. Implementation of CaravelaMPI

CaravelaMPI was implemented by using both MPI and Caravela library functions. The set of functions provided by CaravelaMPI to program parallel applications is presented in Table 4. Four main categories of functions are implemented: *i*) management functions, *ii*) synchronous communication functions, *iii*) asynchronous communication functions, and *iv*)

synchronization functions for *iii*).

Because the synchronous functions in CaravelaMPI are implemented with combining the asynchronous functions, let us explain the asynchronous and the synchronization functions first.

`CaravelaMPI_Irecvsend()` corresponds to the Recv-Calculate-Send interface model. Its arguments are arrays of I/O data streams, arrays of source/destination ranks, arrays of sending/receiving messages, a communicator for the MPI runtime, a flow-model, an array of swap I/O pairs, the number of iteration for flow-model execution and a request data structure. The arrays of I/O data streams can be specified as NULL when the subsequent calculations do not need that intermediate data. The arrays of ranks are used for receiving input data streams and for sending output data streams to/from the flow-model. The communicator is applied for defining communication group in MPI runtime such as `MPI_COMM_WORLD`. `CaravelaMPI_Irecvsend()` receives messages that correspond to given input data streams of the flow-model and executes the flow-model using the swap I/O pair data structure and sends the output data streams to the remote processes. The send and receive operations are implemented by `MPI_Isend()` and `MPI_Irecv()`. By including the argument information for the function, the request also obtains the MPI-related request data structure. The function never blocks the subsequent calculations. Therefore, the function returns immediately after saving the request data structure which obtains type of requested interface and the argument information. Each operation performed in the function (message reception, calculation on GPU and message transmission) is completed when the synchronization functions are called.

`CaravelaMPI_Irecv()` and `CaravelaMPI_Isend()` correspond to the asynchronous versions of the Recv-Calculate and the Calculate-Send interfaces respectively. These functions perform a part of `CaravelaMPI_Irecvsend()` except buffer management for I/O data streams in flow-model. The former function needs to receive an array of memory areas as its arguments for the output data streams because it returns the results of the flow-model execution. On the other hand, the latter function needs to receive an array for the input data streams. Moreover, these functions also do not block the subsequent calculations and just saves argument information to a request data structure. The completion is also performed in the same way as the one of `CaravelaMPI_Irecvsend()`.

The synchronization functions have interface to receive request as its argument which is returned by the asynchronous functions. These functions immediately return after checking the completion of the request. However, while checking the request's status, when some operations have been finished even if the operations are not related to the request, it executes the next operations of those requests. For example, if receiving messages for input data streams related to `CaravelaMPI_Irecv()` has been finished, flow-model execution that is the next step of the function is performed in `CaravelaMPI_Test()`. The function returns a status data structure that contains arrays of I/O data streams used by the flow-model execution and status data structure for MPI functions. On the other hand, `CaravelaMPI_Wait()` blocks the subsequent calculations, until all operations related to its requested arguments have been completed. While checking the completion, this function also tries to finish other pending requests which may have been issued by other functions.

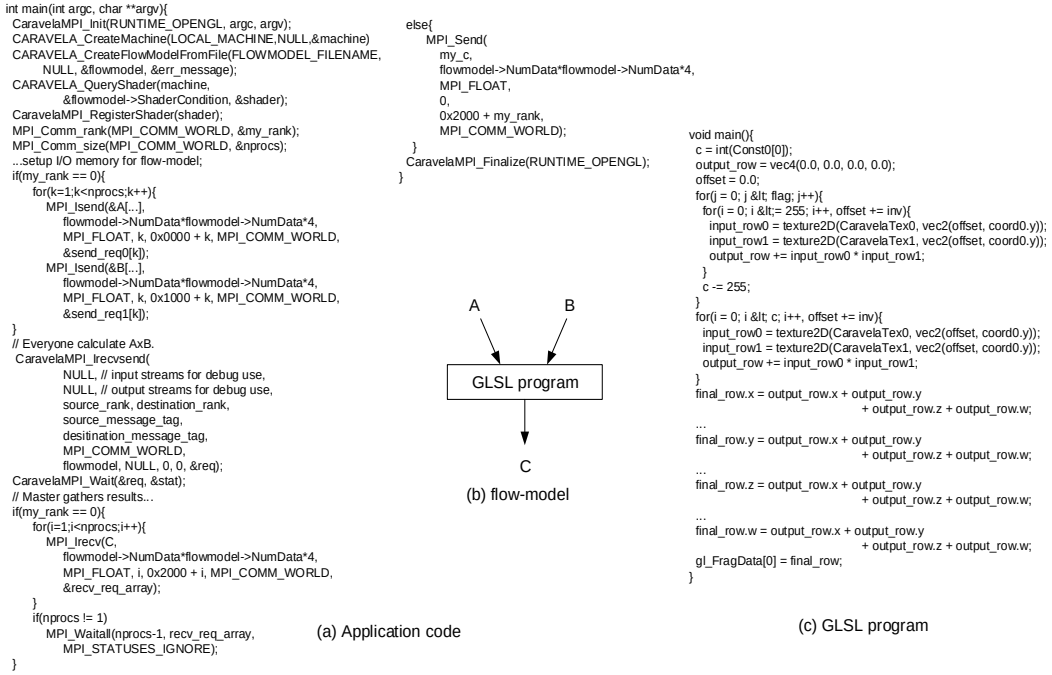


Figure 12. C code and flow-model for matrix multiply.

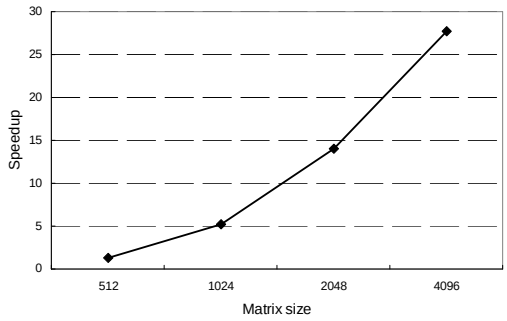
The communication operations included in the request are synchronized with `MPI_Test()` and `MPI_Wait()`, applying the MPI-related request data structure to the functions.

By using the CaravelaMPI functions that combine communications for exchanging data with MPI functions and calculations based on GPUs, an application programmer can concentrate to construct a target algorithm of application without taking into consideration the details for the graphics runtime. Moreover, the application can fully and transparently use the massive computational power of GPUs. Thus, the CaravelaMPI will act as an efficient and high performance message passing library on a GPU cluster.

4.1.5. Experimental Performance and Evaluation

To experimentally validate the proposed interface and to evaluate performance of the CaravelaMPI, experimental results were obtained and are presented in two ways: one evaluates performance using a simple parallel kernel for matrix multiplication; another evaluates the programmability aspect. We used a small cluster computer environment that is made of two computers composed of AMD Opteron 2GHz with 2GB DDR memory where NVIDIA GeForce 7300 with 256MB DDR VRAM and ATI RADEON1950 with 512MB DDR VRAM are equipped, with Linux and connected by 10M Ethernet. The LAM [26] implementation of the MPI is adopted in both machines and the OpenGL runtime environment is used as the graphics runtime of the Caravela library.

In this experiment, the calculation corresponds to the multiplication of matrices with $N \times N$: $C_{N,N} = A_{N,N} \times B_{N,N}$. For parallelizing the computation, the workload is



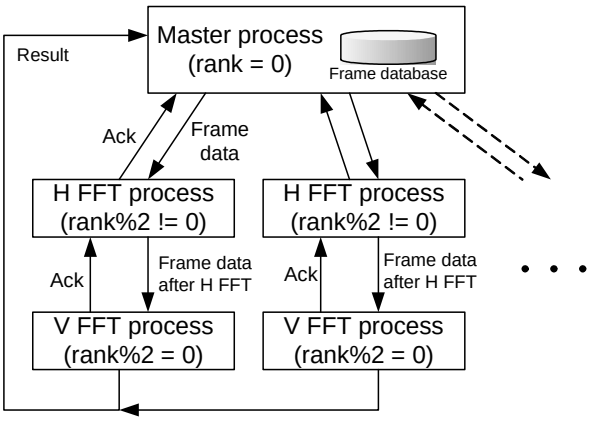
Execution time (sec)				
Matrix size	512^2	1024^2	2048^2	4096^2
CPU-based	0.84	6.05	44.66	345.85
GPU-based	0.65	1.15	3.19	12.49

Figure 13. Execution times and speedups with 4 processes.

equally distributed to each processor, by initially computing $n = N/nprocs$. Process with rank 0 distributes n rows of the matrix A and n columns of the matrix B to the processes that become responsible for them. After receiving the rows and columns, each process calculates $n \times N$ dot products for the elements it is responsible in the matrix C, and returns n rows of the matrix C to the process 0. Using `CaravelaMPI_Irecvsend()` and `CaravelaMPI_Wait()`, the parallel algorithm was implemented with the C code listed in Figure 12(a). The process 0 sends the parts of matrices A and B to other processes. Then, those processes call `CaravelaMPI_Irecvsend()` using a flow-model that calculates dot products for correspondent elements of matrix C. The flow-model is designed as illustrated in Figure 12(b), which has two input data streams (A and B) and generates an output data stream (C). Figure 12(c) shows the shader program generated by the Caravela library in GLSL, which is embedded in the flow-model. This shader program reads four elements A and B in each access to the VRAM, calculates dot products of those elements and outputs results for C by outputting the result to the VRAM.

To compare the performance of the GPU version, we also implemented a CPU version that the dot products were implemented by for-loops and the code was compiled by *mpicc* with *O3* option.

We experimentally measured elapsed time from the initial data distribution started by processor 0 to return of result to the processor, parallelizing with four processes, where two processes are assigned to each computer. Figure 13 shows the execution times and the speedup for different matrix sizes, for N equal to 512, 1024, 2048 and 4096. The speedup is computed as the ratio that the execution time of the CPU-based implementation is divided by the one of the GPU-based. The GPU-based implementation is 27 times faster than the GPU-based one when N is 4096. Thus, the cluster of GPUs achieves much higher performance than the CPU-based cluster. Therefore, we confirmed that the CaravelaMPI is very important factor of providing transparent and flexible API for GPU cluster.



(a) System organization

```
while(1){
  CaravelaMPI_Irecvsend( &inputStream,
    &outputStreams, source_rank, dest_rank,
    source_tags, dest_tags,
    MPI_COMM_WORLD, flowmodel,
    swap_pairs, 2, iterations, &RecvSendReq);
  do {
    MPI_Irecv((void *) buf,
      4, MPI_CHAR, my_rank+1,
      0, MPI_COMM_WORLD, &Irecv_req);
    CaravelaMPI_Test(&RecvSendReq);
    MPI_Test(&Irecv_req, &flag, &status);
    if(flag) break;
  } while(1);
  CaravelaMPI_Wait(&CReq, &stat);
  MPI_Send(ack_message,
    4, MPI_CHAR, 0, desttags[0],
    MPI_COMM_WORLD, &status);
}
```

(b) Code from H FFT part

Figure 14. Structure and application code for FFT system.

4.1.6. Evaluation of Programmability

We implemented a parallel image processing system as illustrated in Figure 14(a). This system performs two dimensional FFT (Fast Fourier Transform) using several image frames saved in a database. The master server picks up every image from the frame database and sends it to the processing pipeline while searching for an idle processing pipeline. The processing pipeline is structured in an H FFT and a V FFT processes. The H FFT process computes a one dimensional FFT for the horizontal direction. The V FFT does the same for the vertical direction. The H FFT process receives a frame from the master, computes FFT using a flow-model, and sends the result to the V FFT process. The V FFT process receives the results from the H FFT process, computes FFT with the flow-model, and sends the result back to the master. When the H or V FFT process finishes processing, it sends a notification to indicate that the process is idle. As soon as the FFT processes receive and compute the frames, they send the results to the next processes. Thus, the processing pipeline receives the input data and generates the output data asynchronously. Therefore, each process must manage dynamically sending/receiving messages and calculations on the FFT processes.

We applied a GPU-based one dimensional FFT algorithm reported in [14] for the FFT processes. Those are implemented into flow-models and passed to the CaravelaMPI functions on the processes.

Let us focus on the code for H FFT process as listed in Figure 14(b), comparing the coding style with the conventional one applied to programming on a GPU cluster. In the code of the figure, `CaravelaMPI_Irecvsend()` function manages the receiving/sending data and the GPU's computation implicitly. The GPU's computation is managed by `CaravelaMPI_Test()` function, and the communication is implicitly performed. Therefore, the code size can become small and the processing flow becomes transparent. If we use the MPI functions and the graphics runtime functions directly, we would have to consider the calling order of both functions arranging the timings of GPU's calculation and communications simultaneously. Thus, we can clearly say that CaravelaMPI achieves better

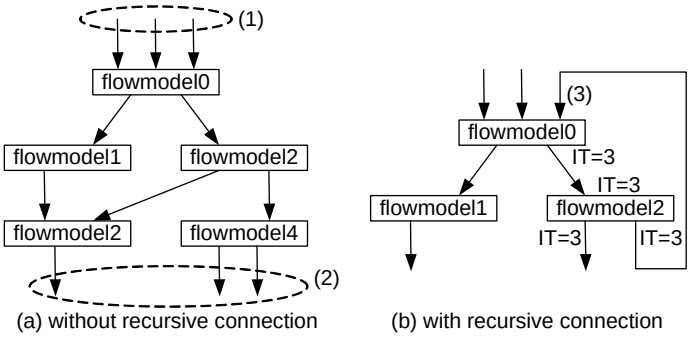


Figure 15. Examples of pipeline-models with and without recursive connection.

programmability on parallel application for GPU cluster than the conventional method that would require the hand-scratched scheduling of GPU's calculations and communications separately.

4.2. Meta-pipeline

Thus, using the flow-model framework, the Caravela platform has implemented a 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 section is focused on an extension of the execution mechanism to a pipelined processing mechanism, directly connecting flow-models assigned to multiple processing units distributed across the Caravela platform.

The mechanism in the Caravela platform that executes the multiple flow-models connected through I/O data streams is called *meta-pipeline*. The meta-pipeline applies the *pipeline-model* to execute the flow-model units.

4.2.1. Pipeline-model

As depicted in Figure 15, flow-models whose I/O data streams are connected can create a pipeline-model. The pipeline-model assumes that a meta-pipeline as a whole may have its own I/O *ports*: input data streams (1) in Figure 15 are received in *ENTRANCE* ports and output data streams (2) are provided in *EXIT* ports. Since it is a pipeline-model, and it has been defined to compute applications, 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 a data stream is provided at the output of the pipeline-model, otherwise the computation would be useless. The second and third conditions impose that a meta-pipeline can not include other independent meta-pipelines, otherwise it has to be split in multiple individual meta-pipelines, according to the pipeline-model. Note that no condition is imposed for the number of input ports, because the pipeline-model can have a self-generated feedback input stream data that results from the output data stream. This is a common situation for

algorithms that generate special data streams, for example the recursive generation of the Fibonacci number sequence.

When all input data streams of a flow-model unit are ready, this unit is executed. For example, in the pipeline model depicted in Figure 15(a), if data for the ENTRANCE ports (1) has become ready, 'flowmodel0' is executed and generates the output data stream needed to 'flowmodel1'. At this time, the readiness of the 'flowmodel1' input data stream triggers its execution, and the process is repeated for the subsequent flow-model units. In the example of Figure 15(b), at the beginning of the execution the input data streams of 'flowmodel0' never become all ready, unless the data stream in the feedback connection (3) is artificially initialized. Without this initialization, a deadlock can occur in the pipeline-model's execution. This particular type of input port, called *INITONCE* port, must be initialized with an input data stream to trigger the execution of the pipeline.

To increase the flexibility of the model, we also define a limit number of iterations (depicted in Figure 15 as IT) 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 Figure 15(b), the port must be initialized every three times the output data is generated by the 'flowmodel2'. The concept of iteration limits is not only applied to INITONCE ports but also to ENTRANCE and EXIT ports. In the case of ENTRANCE port, an iteration limit restricts input data initialization, which means the ENTRANCE port is only initialized when the number of executions is multiple of the IT. The same role is applied to the EXIT ports, also to efficiently get the generated output data streams. Moreover, we call *INTERMEDIATE* ports to the flow-model's I/O which are not in any one of the categories referred above (i.e. a port that connects an output stream of a flow-model to an input stream of another flow-model that does not require any assignment of initial data, on the contrary to the INITONCE port). The concept of iteration limit is also applied to the INTERMEDIATE ports: for instance, when the iteration limits illustrated in Figure 15(b) are set, the output data from 'flowmodel0' is generated every three executions and the input/output data of 'flowmodel2' is initialized/generated every three iterations of 'flowmodel2'. A merit of the iteration limit is to define an execution set; for example in Figure 15(b), '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 avoiding redundant data communication.

In summary, flow-model execution in a pipeline-model is repeated whenever sets of initial data are provided for ENTRANCE port(s) and INITONCE port(s). Moreover, different iterations of the flow-models can be executed in parallel in independent processing units, with the required communication reduced to the minimum by parameterizing the model with the limit numbers of iterations. Thus, the meta-pipeline execution mechanism will behave in a distributed environment as a suitable autonomous stream-based computing unit.

4.2.2. Runtime environment

Meta-pipeline requires a runtime execution mechanism, which is able to check if all input data streams of any flow-model are ready, maintaining the maximum number of processing

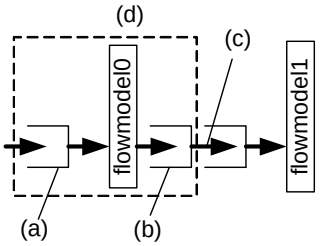


Figure 16. Buffers to avoid serial execution of multiple flow-models connected in a meta-pipeline.

units (GPUs) busy, executing the respective flow-models. Although a connection between flow-models' input and output data streams can be established through a shared buffer that receives the output data stream and allows it to be used as the next input data stream, this approach can serialize the execution of a meta-pipeline. It can prevent the execution of a flow-model that can not output data because the buffer is occupied with data required by the subsequent flow-model. To avoid this serialization, we define an execution model as depicted in Figure 16. Each flow-model has individual buffers for its input and output data streams (Figure 16(a)(b)). When the input data streams are initialized and 'flowmodel0' is invoked, its output data streams are stored in the output buffer once (Figure 16(b)). When the subsequent flow-model's input buffer becomes empty, the content of the output buffer will be moved to the input buffer (Figure 16(c)). This usage of independent buffers for input and output data streams avoids serial execution, in Figure 16 of 'flowmodel0' and 'flowmodel1'.

A single flow-model with input/output data stream(s) and its buffers, such as depicted in Figure 16(d) can be instantiated as a single flow-model execution using the Caravela library functions shown in Table 1. Therefore, it can be mapped and executed in any processing unit of the Caravela platform. When in a connection the output and input buffers have different sizes, data is copied by adopting the smaller buffer size. For example, when the buffer size of Figure 16(c) is smaller than the one of Figure 16(b), data is resized to fit in the buffer of Figure 16(c), and is copied from one buffer to the other. Therefore, even if different sized buffers are connected in a pipeline-model, the I/O data will be smoothly propagated.

4.2.3. Application Programming Interface

An Application Programming Interface for the meta-pipeline is defined and additional functions are included in the Caravela library for implementing it. To invoke a pipeline-model, an application follows the eight steps presented below.

1. **Looking for processing units**
   This step uses the conventional Caravela functions shown in Table 1 to acquire 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 Car-

avela platform. Flow-models that are used in a pipeline-model are reproduced from local or remote flow-model's XML files in this step.

3. **Creating a pipeline-model data structure**
   A pipeline-model is represented in a single data structure in the Caravela system. In this step, the 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 in the 1st step. A pair flow-model/processing unit is named *stage* of the pipeline. The application needs to create all the stages in a pipeline-model in this step.

5. **Creating connections among flow-models**
   This step defines connections between I/O data streams of flow-models registered to the different stages of the pipeline-model. To fulfill the conditions to be a valid pipeline-model, the 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 is provided to the first stage of a pipeline-model, its flow-model 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 the 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 be part of the pipeline-model through remote resources. Moreover,

the pipeline-model is implicitly executed by the meta-pipeline's runtime environment, as the application provides input data to the ENTRANCE/INITONCE ports. Thus, pipeline-model programmers do not need to explicitly schedule flow-model execution.

4.2.4. Library functions

The meta-pipeline is implemented as a set of C-based functions, which extends the original Caravela library with the basic functions. For example, machine creation and shader acquisition are performed by instantiating the basic functions as listed in Table 1.

After acquiring the shaders, a set of other functions has to be used. Firstly the `CARAVELA_CreatePipeline` function is instantiated to create a data structure for the pipeline-model. At the next step, `CARAVELA_AddShaderToPipeline` function adds a shader, acquired by `CARAVELA_QueryShader` function of the original Caravela library, to the pipeline-model data structure. To associate a shader to the pipeline-model, a flow-model previously created by the `CARAVELA_CreateFlowModelFromFile` is introduced by using the `CARAVELA_AttachFlowModelToShader` function. Thus, a pair of a shader and a flow-model has been registered in the pipeline-model, which means stages of the meta-pipeline have been individually defined.

The `CARAVELA_ConnectIO` function is used to setup connections between pipeline stages, receiving as arguments a flow-model previously registered in a pipeline-model and input/output data streams' indices. As a consequence, the I/O data streams are marked as INTERMEDIATE ports. If needed, `CARAVELA_SpecifyInitOncePort` is called after making a connection, to specify that an INTERMEDIATE port is an INITONCE port. The iteration limit (IT) parameter, whose meaning was defined in the previous section, is specified after creating the connections by instantiating the `CARAVELA_SpecifyEntrancePort`, the `CARAVELA_SpecifyExitPort`, and the `CARAVELA_SpecifyIntermediateInput/Output` functions. The setup of the pipeline-model is the last step before the application can start implementing the meta-pipeline.

The function `CARAVELA_ImplementPipelineModel` implements the pipeline-model. It assigns flow-model/shader pairs (i.e. stages) to machines equipped with shaders. This function returns the identifiers of the ENTRANCE and EXIT ports in an array. This function does not execute any stage of the pipeline-model. The execution is triggered by the `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 function for executing the stages whenever the established conditions are fulfilled. This execution mechanism can perform local execution or remote execution.

4.2.5. Implementation of the execution mechanism

Since the meta-pipeline execution is triggered by inputting data into the ENTRANCE ports, data must be provided to the ports periodically to maintain pipeline operation. Moreover, data streams generated on the EXIT ports must be read by the application as soon as they are generated in order to avoid stalls. Figure 17 shows, as an example, a piece of code for

```
// 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 17. Code for pipeline-model execution, which provides input data to ENTRANCE port(s) and receives output data from EXIT port(s) of a meta-pipeline.

executing a pipeline-model, where new data is sent in each iteration to ENTRANCE ports and new results are received from EXIT ports.

For executing stages in local shaders, `CARAVELA_SendInputDataToPipeline` and `CARAVELA_ReceiveOutputDataFromPipeline` functions have the chance to execute stages by repeatedly executing the code shown in Figure 17. 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 (see Figure 3). 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 the flow-model in a worker server is triggered by input data. Therefore, the application does not need to explicitly activate each flow-model's distributed remote shaders. To conclude, we can state that the meta-pipeline mechanism allows applications to define stages, which are implicitly executed in local or in remote machines.

4.2.6. Designing a modeling tool for meta-pipeline applications

The meta-pipeline mechanism allows the definition and execution of the pipeline-model with complex interconnection patterns, which can easily lead to deadlock situations. For example, the flow-model unit (1) in Figure 18 can never start execution, because not all the input data streams are available, and consequently the whole pipeline execution will be stalled. To avoid this situation, the application programmer needs to identify the edges

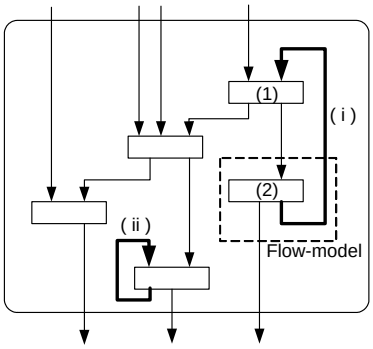


Figure 18. Flow-model and Pipeline-model in Caravela platform where a deadlock occurs in the loop (i) consisting of (1)→(2)→(1) and loop (ii).

in the feedback connections that require initialization. If the feedback connections in a pipeline-model are simple and in small number, such as the connection (i) or (ii) illustrated in Figure 18, it is an easy task for the programmer to identify them. However, when the pipeline-model is more complex, this task when performed by hand becomes hard and susceptible to introduce errors.

To program an application according to the pipeline-model one has to define the meta-pipeline and program the communication between flow-model units using a set of Caravela C functions. The programmer has to register flow-model units to a pipeline-model using `CARAVELA_AttachFlowModelToShader` function, and to establish the connections between the flow-models through the `CARAVELA_ConnectIO` function. A GUI-based modeling tool that abstracts the details needed to develop this code, which is quite useful in practice, is described in the next sections.

4.2.7. GUI-based modeling tool for the pipeline-model

The main purpose of the modeling tool is to provide: *i*) a GUI for defining the pipeline structure; *ii*) save and restore mechanisms for establishing meta-pipelines; and *iii*) tools to verify the pipeline-model and to generate the executable program. Component *ii*) encapsulates pipeline-model into a file, which enables to share it with other applications and programmers. The last component checks the validity of pipeline-models, by identifying connections on the computational loops that can cause execution deadlock. Moreover, this component generates the required source code for implementing the defined meta-pipeline application. This last task is accomplished by instantiating the required Caravela C functions in the proper order. A typical design flow with these three components is: programmer graphically creates a pipeline-model by establishing the flow-models and connections between them; whenever a programmer wants to save or interrupt the design, the defined pipeline-model is stored into a file, which can be reused by other programmer or application; finally, the tool verifies the design and generates the C code required to implement the specified meta-pipeline. The component *i*) can be implemented with conventional graphics libraries, such as *Windows control* and *Tcl/Tk*. The GUI shows a pictorial diagram of the flow associated to the pipeline-model. To save and restore flow-models to/from files,

Algorithm 1 Algorithm for identifying computational loops in pipeline graphs.

1: Identification of all cyclic paths in the pipeline graph
2: Sorting cyclic paths in ascending order regarding the number of nodes
3: Reducing the sorted cyclic path list to the minimum cyclic paths
4: Composing a list exclusively with edges to be initialized

standard languages such as XML can be used. However, the component requires the development of a method to detect computational loops in the pipeline. Therefore, a new general and efficient algorithm was developed to detect computational loops in pipeline graphs, in order to avoid deadlock situations.

4.2.8. Algorithm to detect execution deadlock

Loop identification is an essential step in high-level programming languages for performing loop transformation and optimization in computers. Among the algorithms that have been proposed for loop identification, the Tarjan's algorithm [42] is the most well known for identifying loops for the case of reducible graphs, while the Havlak's algorithm [18] corresponds to an extension of the Tarjan's algorithm also to handle irreducible graphs. However, both algorithms are mainly targeted to reducible graphs that only contain loops with a single entry, in opposition to the irreducible graphs that accommodate loops with more than one entry. Therefore a new efficient algorithm is required to solve the deadlock problem by identifying the uninitialized inputs in any pipeline-model structure with a general topology and no restrictions.

Servers/programs and the data streams in the pipeline-model can be represented by a directed graph (DG). In such DG, a node represents a flow-model while input and output data streams are represented by directed edges. These DGs are designated in this particular application *pipeline graph*s. Conditions to compute a node are the ones required by the flow-model: the node is computed *iff* data in all the input edges is ready, and data is only generated in output edge(s) after computing the respective node.

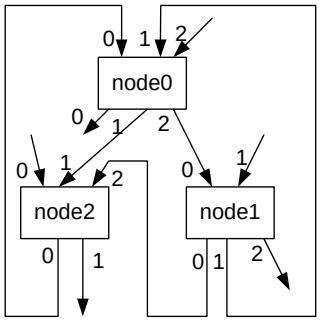
The problem to be solved here is to find the minimum set of edges that needs to be initialized to start computing the pipeline graph, in order to avoid a deadlock situation. To find such edges, the algorithm has to identify loops like (i) and (ii) in Figure 18. These loops are called *cyclic paths* because they are cyclically executed, such as node (2)→node (1)→node (2) in Figure 18.

4.2.9. Proposed algorithm

Since the pipeline graph accepts that a single node may have multiple input and output edges, without imposing any restrictions, multiple cyclic paths can exist in such a graph. For example, a pipeline graph might contain two or more cyclic paths between two given nodes, including the case when these paths are composed by the same nodes. Supposing the cyclic path node0→node1→node2→node1, the connection between node2 and node1 can be presented in two or more cyclic paths (e.g. node1→node2→node1 and node2→node1→node2), and therefore cyclic paths have to be identified not only by the nodes but also by the edges.

```
struct output{
  struct node *next_node;
}
struct node {
  int num_output;
  struct output[ ]:
  struct node *next;
};
struct cyclic_path{
  struct node *node:
}
struct cyclic_path_list {
  struct node *node;
};

global struct cyclic_path current_cyclic_path;
global struct cyclic_path_list current_node;

(a)
function create_cyclic_path_list(node_array){
  for(i=0;i<num_nodes;i++)
    create_cyclic_path_list_recurse(node_array[i]);
  }
}

function create_cyclic_path_list_recurse(node){
  for(i=0;i<all the outputs;i++){
    if(the next input of the node's output[i]
        is not connected to the other node) continue;
    if(node is included in current_cyclic_path){
      add current_cyclic_path to cyclic_path_list.
        return;
      }
      else{
        add node to current_cyclic_path.
        create_cyclic_path_list_recurse(node->output[i]->node);
      }
  }
}

(b)
function sort(){
  sort cyclic paths in cyclic path_list by the number of nodes
}

(c)
function reduct_paths(){
  for(i=0;i<num_paths-1;i++){
    for(j=i+1;j<num_paths;j++){
      compare if the cyclic_path[j] includes cyclic_path[i].
      if(cyclic_path[i] is included)
        delete cyclic_path[j] from cyclic_path_list
    }
  }
}

(d)
function main(){
  read node information from file to node_array.
  create_cyclic_path_list(node_array)
  sort();
  reduct_paths()
}
```

Figure 19. The algorithm phases represented in pseudo-code: each number in the figure indicates the corresponding phase number in the loop detection algorithm.



[0] 2-0 0-1 2-0
[1] 1-1 0-2 1-1
[2] 0-2 1-1 0-2
[3] 0-1 2-0 0-1
[4] 2-0 0-2 1-1 0-2
[5] 2-0 0-2 1-0 2-0
[6] 1-1 0-1 2-0 0-1
[7] 1-0 2-0 0-2 1-0
[8] 1-0 2-0 0-1 2-0
[9] 0-2 1-0 2-0 0-2

b)

[0] 2-0 0-1 2-0
[1] 1-1 0-2 1-1
[5] 2-0 0-2 1-0 2-0

c)

Figure 20. Application example of the algorithm (x-y denotes node x-edge$y$): a) pipeline graph; b) list generated after phase 1 and phase 2; c) list after phase 3.

Four main phases can be considered in the proposed Algorithm 1. Phase 1 of Algorithm 1 generates a *cyclic path list* with all existent cyclic paths, each one represented by the nodes that compose it and the corresponding output edges. Phase 2 of Algorithm 1 sorts the cyclic paths identified in phase 1 in ascending order according to the number of nodes. After the sorting process, longer cyclic paths may include other shorter cyclic paths existent in the list. Phase 3 of Algorithm 1 finds a minimum set of cyclic paths that are exclusive, in the sense that they do not accommodate common shorter cyclic paths. This means that when a cyclic path exactly matches a part of another cyclic path in the list, the larger one is removed from this list. From this phase it results a set of exclusive cyclic paths, herein designated by *true cyclic paths*. In the last phase, phase 4 of Algorithm 1, a list of true cyclic paths without common edges is produced, and then a list with the edges that need to be initialized is built. For example, when the following two true cyclic paths exist, node1(edge1) $\rightarrow$ node2(edge3) $\rightarrow$ node1 and node0(edge1) $\rightarrow$ node1(edge1) $\rightarrow$ node2(edge3) $\rightarrow$ node1,

and the edge node1(edge1) $\rightarrow$ node2 is selected to be initialized, the later cyclic path is removed because the required initialization was already performed in the former.

Figure 19 presents pseudo-code of the proposed algorithm as well as related data structures required to implement it. The naif implementation of phase 1 in Figure 19 directly identifies cyclic paths in a recursive way. For example, a fast mergesort algorithm can be used in Figure 19 to sort the cyclic paths. In Figure 19, the third phase reduces the cyclic path list by considering the already initialized edges. Finally, Figure 19 represents the main functions that execute the phases in order to produce the final list of edges not initialized. A lemma, and the respective proof, is provided in appendix to prove that phases 3 and 4 of the proposed algorithm reduce the cyclic path list and compose a list exclusively composed by edges not initialized.

Figure 20 illustrates an application of the proposed algorithm to a pipeline graph with only three nodes. After identifying loops in phase 1 and phase 2, the cyclic path list shown in Figure 20 b) is achieved. It includes all the cyclic paths ranked by the number of nodes (ascending order). From the top of the list to the base, the cyclic path list is reduced in phase 3 of the algorithm by comparing all cyclic paths. The "true" list of cyclic paths is presented in Figure 20 c). The paths in the list are the ones that really have to be initialized in order to avoid deadlock. For example, when we choose the input0 of node0 and the input0 of node1 as the edges to be initialized, the condition to avoid deadlock is satisfied, because all the cyclic paths listed in Figure 20 includes one of these edges.

4.2.10. Complexity analysis and optimization techniques

The complexity of the proposed algorithm depends on the number of the cyclic paths, and in particular on the number of true cyclic paths. The complexity of phase 1, by considering V vertices and E edges, is O($V + E$). By assuming that C cyclic paths are identified, the complexity of fast sorting algorithm for the phase 2 is O($C \log C$), and the complexity of phase 3 is, in the worst case, O(CV). O(CV) can also be considered the worst case complexity for the fourth and last phase. In practice, the optimization technique decreases, from the second to the last phase, the number of cyclic paths that has to be analyzed to C. By considering the number of cycles proportional to the number of vertices O(V), the total complexity is O(V^2), even for a pipeline graph with a high number of edges E, in the order of O(V^2). The complexity of the Havlak's algorithm is $\theta(V^2)$, even for graphs with a number of edges proportional to the number of vertices O(V).

Some aspects can be optimized for achieving a more efficient implementation of the proposed algorithm. Although phase 1 of the proposed algorithm is based on the Dijkstra's algorithm, we can simplify it for our particular problem. One of the main changes was to dismiss all cycle paths that include other cyclic paths inside, such as, for example, the case with node0$\rightarrow$node1$\rightarrow$node2$\rightarrow$node1$\rightarrow$node0, which does have to be considered. This optimization reduces not only the memory required to store the list of cyclic paths but also the number of comparisons required in phase 2 for sorting the cyclic paths.

Together with the phase 1, phase 3 is the most computational demanding part of the algorithm, when the number of cyclic paths is reduced based on an all-to-all comparison. This computational burden can be lightened by taking advantage of the fact that the cyclic paths are ordered according to the number of nodes, and that with the previous optimization

technique it can be assured that a cyclic path A can not match any other paths with a larger number of nodes than the path A. Therefore, for any path A, the comparisons in phase 3 can stop when the number of nodes in a cyclic path is larger than the one of path A. This last optimization significantly reduces the number of comparisons required for performing phase 3 of the algorithm.

4.2.11. Implementation and experimental results

Let us show a realistic application example programmed by applying the proposed meta-pipeline execution mechanism. The Discrete Wavelet Transform (DWT) [7] was chosen as the application to show the effectiveness of the meta-pipeline execution mechanism. We have also programmed, in C# language, a GUI-based *PipelineModelCreator* entry tool for programming meta-pipeline applications. To assess this tool and the deadlock detection function, we will evaluate the efficiency of the proposed algorithm to extract the cyclic paths in the pipeline-model and compare it with the algorithm without the optimization techniques discussed in the previous section.

1. 2D Discrete Wavelet Transform
   Discrete Wavelet Transform (DWT) [7] is a powerful tool for image processing applications, such as compression used in JPEG2000 standard, denoising, edge detection and feature extraction. The 1D 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 applying a linear low-pass and high-pass filter 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:

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

   where K is the number of filter taps. The decimation process is already embedded in 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 the DWT, two dimensional DWT (2D-DWT) can be performed by sequentially applying the equations above across the horizontal and vertical image directions. It generates 4 sub-bands (i.e. LL, HL, LH and HH as shown in Figure 21(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 LL sub-band previously computed. This recursive calculation is iterated until the given number of decomposition levels is achieved (typically 3 to 5 levels). Figure 21(c) shows a result of 2D-DWT with 2 decomposition levels calculated from the input image in Figure 21(b).

2. Pipeline-model for 2D-DWT

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

Figure 21. 2D-DWT example with 2 decomposition levels.

The recursive nature of N-level DWT decomposition suggests a pipeline-model organization, with each pipeline stage corresponding 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 compose the other output stream.

The program included in the flow-model is shown in Figure 22(a). The program is written in GLSL. It receives an LL sub-band as the input data stream and generates two output data streams. Using this flow-model, the pipeline-model illustrated in Figure 22(b) can be defined. Each stage in the pipeline-model consists in a flow-model (kernel program in Figure 22(a)). From one level to the next, the size of the input data streams are reduced to 1/4. The input data stream in the ENTRANCE port triggers the flow-model in level 1. One of the output data streams of this flow-model is connected to the next stage (i.e. INTERMEDIATE port). The other output data streams carry the sub-bands at the corresponding decomposition levels, through the EXIT ports.

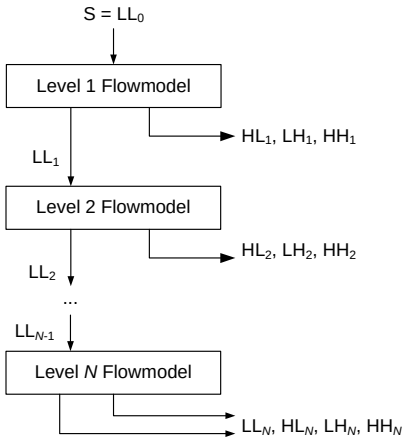
After implementing the pipeline-model and by feeding the image data to the ENTRANCE port, the Caravela runtime executes the flow-model in each stage when the input data becomes available. If the pipeline-model is computed in a single local shader, each stage is assigned to the shader and automatically replaced by the next stage. When it is executed in remote worker servers, each flow-model is assigned to a worker, and waits for the input data that is propagated from the previous flow-model in the pipeline.

4.2.12. GUI-based tool with automatic deadlock detection

The time to compute phase 1 to phase 3 of Algorithm 1 is measured by considering randomly generated pipeline graphs. The pipeline graphs are generated considering a number

```
uniform sampler2D CaravelaTex0;
uniform sampler2D CaravelaTex1;
uniform vec4 const0;
uniform vec4 const1;

void main()
{
  float delta = 1/NUMDATA;
  int i;
  vec4 tmp, tmp0, tmp1;
  vec2 coord = gl_TexCoord[0].xy;
  vec2 caux;
  coord += coord;  caux = coord;
  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.x = dot(tmp, const0); tmp1.x = dot(tmp, const1);

  coord.x = caux.x; 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.y = dot(tmp, const0); tmp1.y = dot(tmp, const1);

  coord.x = caux.x; 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.z = dot(tmp, const0); tmp1.z = dot(tmp, const1);

  coord.x = caux.x; 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.w = dot(tmp, const0); tmp1.w = dot(tmp, const1);

  result.x = dot(tmp0, const0);
  result.y = dot(tmp0, const1);
  result.z = dot(tmp1, const0);
  result.w = dot(tmp1, const1);

  gl_FragData[0] = result;
  gl_FragData[1] = result;
}
```

(a) kernel program of flow-model



(b) pipeline-model for DWT

Figure 22. Flow-model and pipeline-model for the 2D-DWT.

of nodes that varies from 2 to 10. For each node, a number of output edges, that also varies from 2 to the number of nodes, and a number of input edges equals to the number of nodes. For evaluating the algorithm and the optimization techniques proposed in section 4.2.10., the corresponding program was compiled with GCC 3.3.5 and executed on the Linux operating system. A personal computer with a 3.2GHz Pentium4 with 1GB DDR400 memory was used for obtaining the experimental results.

Figure 23 shows the obtained results for different numbers of true cyclic paths. Without applying any optimization technique, the execution time varies with $C \times logC$. However, when the number of valid cyclic paths grows the execution time significantly increases, mainly due to the memory accesses. Therefore, the reduction of the number of memory accessed is important to execute the algorithm with a good performance. The execution time is drastically reduced (see Figure 23), mainly by applying the optimization technique in phase 3. Moreover, the execution time has become predictable due to the decreasing of

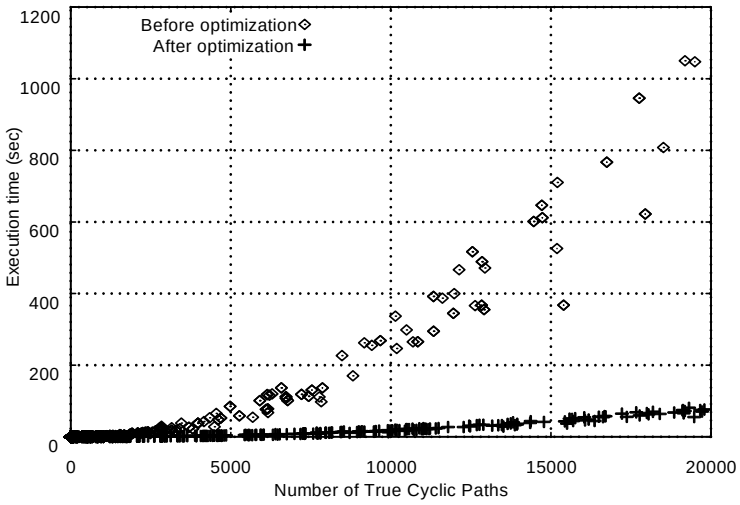


Figure 23. Experimental results with and without the optimization techniques.

the number of memory accesses in phase 1.

Figure 24 shows a design verification example with the pipeline-model illustrated in Figure 20. A screen-shot of the tools with the automatically detected computational loops is presented in Figure 20 a); the dotted lines represent the connections that have to be initialized in order to avoid deadlock. We implemented an additional function to specify initialization dataset for each marked input edge. After specifying the initialization dataset for all the marked inputs (repeating the steps as performed in Figure 20), the tool results show the connections with just solid lines. Then the programmer can give order to the tool to generate the C code for the designed pipeline-model and to execute it on the Caravela platform.

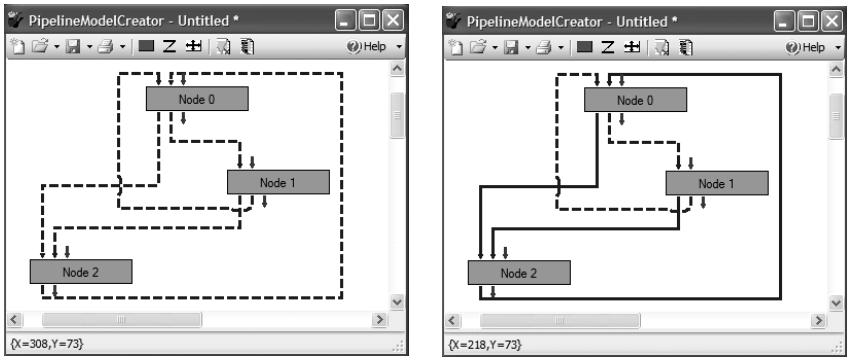
With the PipelineModelCreator tool, programmer is able to graphically design a meta-pipeline application through simple steps and to automatically check the existence of computational loops in order to avoid deadlocks.

4.3. Automatic Parallelization of a Pipeline-flow Based on Flow-models

4.3.1. GUI-based stream-based computing

We have developed a GUI of Caravela platform as shown in Figure 25. The programmer just defines the flow-models and connection among those flow-models graphically. It generates executables and batches needed for CarSh. The GUI is very helpful for automatic programming of the accelerators. However, it is very hard to generate an execution scenario for the processing pipeline. For example, given a processing pipeline as depicted in Figure 26 (a), it is easy for us to identify the execution order. While the input data for *flowmodel1* is given successively, the overall calculation is invoked in the order from *flowmodel1* to *flowmodel4*.

When we consider the concurrency of the execution of multiple flow-models, it is also conceivable for us by writing the execution order assuming the following considerations: 1)



(a) After verifying the pipeline-model. Three loops are detected.

(b) After giving initialization dataset to the connection from node2 to node0. Two computational loops are solved.

Figure 24. Example of design and verification with the PipelineModelCreator.

flowmodel1 and *flowmodel2* are not executable in parallel since the connected I/O are interfered with each other because it is assigned to the same physical buffer. 2) *flowmodel2* must be executed after *flowmodel1* because the input data must be prepared (we call this *initialized*) before the flow-model execution. As the right side of Figure 26 (a) suggests, after the executions of *flowmodel1* and *flowmodel2*, we can think intuitively that the combinations of *flowmodel2 & 4* and *flowmodel1 & 3* are executed concurrently.

However, given the processing flow of Figure 26 (b), how do you specify which is the flow-model initially executed when the first input data is given to *flowmodel1*? How is the parallelism if multiple flow-models could be invoked concurrently? It is impossible to implement the perfect GUI-based programming method before solving these problems. As an objective of this section, we propose an algorithm that mechanically exploits the deterministic execution order and the parallelism from any kind of pipeline execution flow.

4.3.2. Spanning Tree

Challenges for execution ordering and exploiting parallelism from a program are studied in the decades by researchers of compilers. The High Performance Fortran (HPF) is a well-known solution to exploit potential parallelism from a numerical program description. *Spanning Tree* introduced in [45] is a technique to determine the control flow of a program description.

Given a directed graph $G(V, E)$, where V is a set of vertices (nodes) of G, and E is the set of edges of G, $S(V, T)$ is called the $G(V, E)$'s *spanning tree*, when a subset of edges T satisfies $T \subseteq E$ and the graph $S(V, T)$ forms a tree. Here, $S(V, T)$ does not include loops. When spanning tree is defined, the edges in G is categorized into four types: 1) *Tree edges* form the spanning tree. 2) *Advancing edges* are a set of edges of $X \rightarrow Y$ that are not the tree edges. Y is a descendant of X. The edges jump to vertices in the lower structure of the tree. 3) *Retreating edges* are also a set of edges of $X \rightarrow Y$ that are not the tree edges. Y is an ancestor of X. That is, the edges jump to vertices in the upper structure of the tree. Finally, 4) *Cross edges* are the rest of the edges which do not belong to 1) - 3). Those are a set of edges of $X \rightarrow Y$, where Y is neither ancestor nor a descendant of

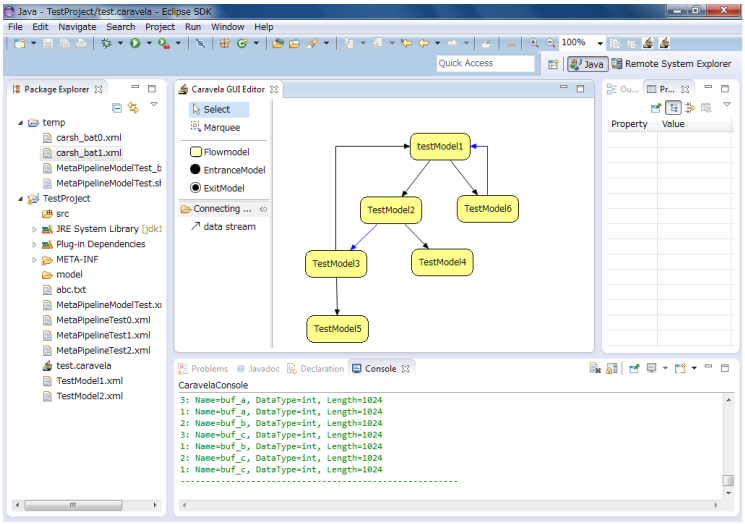


Figure 25. The Caravela GUI implemented as an Eclipse plugin that generates XML files of the flow-model, CarSh executables and batches.

Algorithm 2 Depth-first spanning tree (DFST) algorithm.

```
void DFST(int x){
  pre_num ++;
  NPre[x] = pre_num;
  for(int y=0; y<N; y++){
    if(edges[x][y].connect){
      if(NPre[y] == 0){
        //x → y is a tree edge
        DFST(y);
      }
      else if(Npre[x] < Npre[y]) { // x→y is an advancing edge }
      else if(NRPost[x] == 0){  //x → y is an retreating edge }
      else { // x → y is an cross edge }
    }
  }
  NRPost[x] = rpost_num;
  rpost_num --;
}
```

X. Here, there exists a condition regarding the root vertex of the spanning tree. From a root vertex, the graph must have a reachable path to all other vertices. If the path does not include all vertices of G, it does not have a spanning tree. However, note that a spanning tree generated from a node is uniquely found in G. Therefore, a different root constructs a different spanning tree from the same graph.

Algorithm 2 lists the steps to categorize the edges of a directed graph into four categories above, when a root vertex is given to the `DFST` function. This algorithm is developed based on the Tarjan's depth-first search algorithm [42]. In this meaning, we call it *Depth-First Spanning Tree* (DFST) algorithm. Regarding the tree edges generated by the algorithm, we can find the spanning tree.

The algorithm is a combination of the preorder numbering $NPre()$ and the reverse post numbering $NRPost()$. In the former numbering, if $NPre(X) < NPre(Y)$, X is either a preorder ancestor of Y in the tree, or the left of Y. In the latter numbering, if

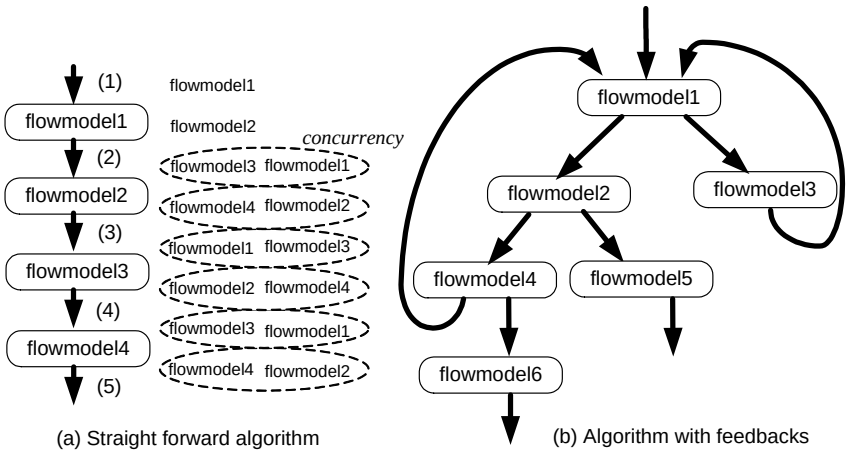


Figure 26. Pipeline flow examples of a) straight forward algorithm and b) algorithm with feedbacks.

$NRPost(X) < NRPost(Y)$, X is either a preorder ancestor of Y in the tree, or the right of Y. First, in the step performing the preorder numbering , the DFST is checking the preorder number of the next connected node after the current node. If it is zero, it is detected as a tree edge. When the search reaches a leaf of the tree, it performs the reverse post numbering, returning to the tree edges. In the backward searching, it marks one of retreating, advancing and cross edges.

Figure 27 shows an example of a spanning tree generated by the DFST. The pairs of numbers in the figure are $(NPre, NRPost)$. We select A as the root vertex. First the preorder numbering is performed in the order of $A \rightarrow B \rightarrow C \rightarrow I \rightarrow J \rightarrow D \rightarrow E \rightarrow G \rightarrow H \rightarrow F$. During the numbering of the backward searching, the reverse post numbering is performed in the order of $J \rightarrow I \rightarrow H \rightarrow G \rightarrow E \rightarrow F \rightarrow D \rightarrow C \rightarrow B \rightarrow A$. In the former step, the tree edges are found like the path marked with the thick arrows. The retreating edges and the crossing edges are also found during the backward search.

4.3.3. Discussion

Spanning tree method is also applied in the communication network field. The Spanning Tree Protocol (STP) is a network protocol that ensures a loop-free topology for any bridged Ethernet-based local area network. The basic function of STP is to prevent bridge loops and the broadcast radiation that results from them. The spanning tree also allows a network design to include spare (redundant) links to provide automatic backup paths if an active link fails, without the danger of bridge loops, or the need for manual enabling/disabling of these backup links. Spanning Tree Protocol (STP) is standardized as IEEE 802.1D. As the name suggests, it creates a spanning tree within a network of connected layer-2 bridges (typically Ethernet switches) and disables those links that are not part of the spanning tree, leaving a single active path between any two network nodes [37]. Perlman applied STP to a routing algorithm in a communication path, considering the network connections among switches and communication nodes as edges and nodes of a directed graph [38]. The algorithm finds the shortest network path with the smallest number of links that eliminates loops.

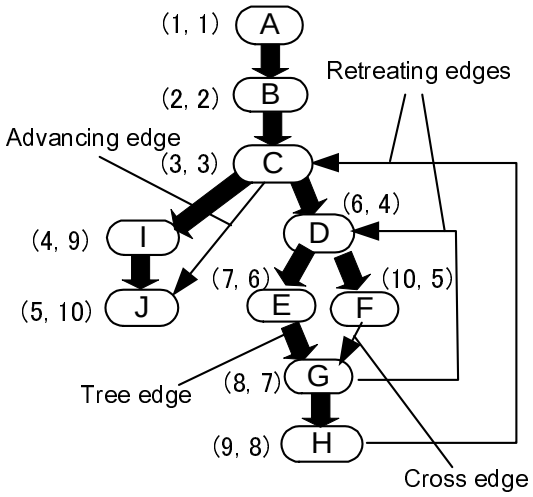


Figure 27. Spanning tree examples.

To break loops in the LAN while maintaining access to all LAN segments, the program in each bridge that allows it to determine how to use the protocol is known as the spanning tree algorithm. The algorithm is specifically constructed to avoid bridge loops (multiple paths linking one segment to another, resulting in an infinite loop situation). The algorithm is responsible for a bridge using only the most efficient path when faced with multiple paths. If the best path fails, the algorithm recalculates the network and finds the next best route.

The Dijkstra's algorithm [8] is also another well-known graph search algorithm that solves the single-source shortest path problem for a graph with non-negative edge path costs, producing a shortest path tree [28]. For a given source node in the graph, the algorithm finds the shortest path and the reaching cost between a vertex and another vertex. For example, if the vertices of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, the Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. As a result, the shortest path first is widely used in network routing protocols, most notably IS-IS [15] and OSPF (Open Shortest Path First) [30].

In the Dijkstra's algorithm, the node at which it starts is called the initial node, and the distance of node Y is the one from the initial node to the node Y. The algorithm will assign some initial distance values and will try to improve them step by step. The steps of the algorithm are summarized below; 1) assigning every node a temporary distance value, 2) marking all nodes unvisited, for the current node, 3) considering all of its unvisited neighbors and calculating their temporary distances, 4) marking the current node as visited and removing it from the unvisited set, and 5) finishing the algorithm if the destination node has been marked as visited or if the smallest temporary distance among the nodes in the unvisited set is infinity, 6) selecting the unvisited node that is marked with the smallest temporary distance, setting it as the new current node and then go back to step 3).

The Dijkstra's algorithm is usually the working principle behind link-state routing protocols, OSPF and IS-IS being the most common ones. The process that underlies the Dijkstra's algorithm is similar to the greedy process used in Prim algorithm [39]. Prim's purpose

is to find a minimum spanning tree [10] that connects all nodes in the graph. Meanwhile, the Dijkstra's is concerned only between two nodes. The Prim's does not evaluate the total weight of the path from the starting node, only the individual path.

The spanning tree algorithm is to discover a loop-free subset of the topology dynamically. For example, if a network topology does not change, the network is not partitioned temporarily while nodes switch over to other routes. If the backup root is very near the old root, the topology will not change significantly even when the old root dies. On the other hand, in the case of the Dijkstra's algorithm, when any cost is changed, it inevitably needs to recalculate it. Therefore, the search result of Dijkstra's algorithm is not adaptive to the dynamic route path.

According to the discussion above, the spanning tree algorithm is powerful for finding a route path between the nodes in a tree. When we consider a pipeline with processing tasks (i.e. flow-models) connected by the I/O data streams, it can be treated as a tree. Therefore, the spanning tree algorithm can be applied to define a unique processing order. Additionally it also finds the feedback I/Os. In the spanning tree, the nodes with the same depth from the root node do not have edges among them. This means that those nodes (flow-models) can be independently executed. The groups of the nodes induce a definition of stages in the processing pipeline because we can define an execution order of the groups. Thus, the spanning tree algorithm can define an effective pipeline order with all tasks included in the processing flow. This section will focus on the characteristics of the pipeline flow exploited by the spanning tree algorithm and proposes a novel algorithm using the spanning tree that extracts the best parallelism.

4.3.4. Parallelism extraction algorithm with spanning tree

In order to address the execution order and finds concurrency, we need to develop an algorithm for 1) finding a flow-model executed first, 2) finding a deterministic execution order without I/O buffer collisions and 3) exploiting an available concurrency from the processing flow. We also define the CarSh batch scenario from the algorithm. Let us map the processing flow to a directed graph. The flow-models correspond to the nodes. The I/O data streams correspond to the edges.

1. Finding the first execution flow-model
   First, we define an *executable node* and a *root node*. When all edges point to a node that is given, we call the node *executable*. The edges that come out from the node are *initialized* after the execution of the node. Here, these nodes are found by the algorithm explained in section 4.2.9.. Selecting one of the nodes in the minimum cyclic paths as the root node, we can find a spanning tree. It includes a unique path of the tree when the execution steps follow the tree edges from the root node to downward. This path becomes the execution flow of the directed graph of the flow-models.

2. Extracting parallelism and determining execution order
   The number of nodes in a cyclic path is not only one in general. Any node in a minimum cyclic path can be the root node of the spanning tree. Therefore, it is

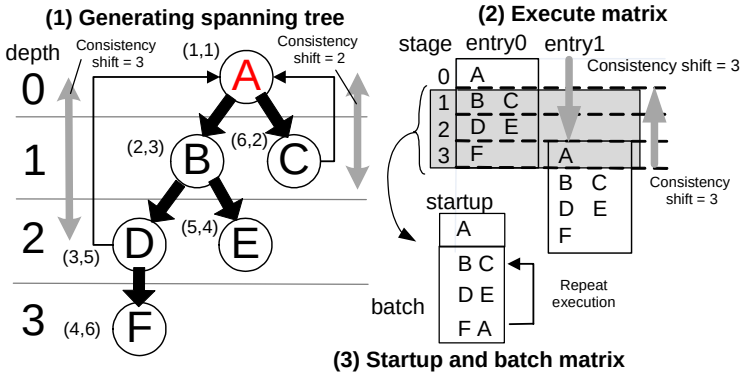


Figure 28. Processing steps of the PEA-ST.

available for all nodes to be selected as the root node in the spanning tree. Here, we assume that the processing flow builds a strongly connected directed graph.

The goal of this section is to exploit parallelism of the processing flow. With the parallelism, we perform a processing order of concurrent execution of multiple flow-models in a pipeline manner by using the parallelism extraction algorithm with spanning tree (PEA-ST). Here, let us consider the case of Figure 28. We assume that the edges $D \rightarrow A$ and $C \rightarrow A$ are initialized. Then we select A as the root node. The spanning tree of the graph becomes like Figure 28(1). The initialized edges are categorized as the retreating edges. Others are the tree edges.

To exploit the parallelism from the spanning tree, we define *depth* and *stage* of a processing pipeline. The depth of a node is defined as the number of tree edges from the node to the root node.The stage is defined as a set of nodes (kernel programs of flow-models) which can be invoked concurrently without the I/O buffer conflicts. Let us generate the depths and stages of the spanning tree in Figure 28. Here, we define that a tree edge has a single depth of the pipeline. For instance, the depth of E is 2 because there are two tree edges in the path of $A \rightarrow B \rightarrow E$. As another example, the depth of F is 3 because there are three tree edges in the path of $A \rightarrow B \rightarrow D \rightarrow F$. Thus, the total depth of the example graph becomes four.

According to the definitions of the stage and the depth, it is obviously to find that the nodes with the same depth can be added to the same stage. For example, the nodes B and C or the ones D and E can be invoked concurrently because there is no I/O conflict among those nodes in the same stage. Therefore, we can build the pipeline as $A \rightarrow (B, C) \rightarrow (D, E) \rightarrow F$. Shifting a stage, we can organize a pipeline like $A \rightarrow (B, C, A) \rightarrow (D, E, B, C) \rightarrow (F, D, E) \rightarrow (A, F) \rightarrow (B, C, A)$...

In the explanation above, we ignore the retreating edges. Here, we consider the effect of the retreating edges (i.e. loops) in the graph. If we completely ignore a retreating edge, the pipeline works incorrectly due to I/O conflicts caused by the loops. For example, the stage (B, C, A) can not be executed correctly because A must be invoked after C. Moreover, A must be invoked after D in the next execution. To resolve the problems we define *consistency shift*. First we find a re-

treating edge. In the loop where the two nodes connected by that retreating edge, we calculate the number of nodes connected by tree edges. In the example case, the edge $D \rightarrow A$ includes three nodes (i.e. A, B and D). The edge $C \rightarrow A$ includes two nodes (i.e. A and C). The consistency shift is that number we just calculated. For example, if the consistency shift is two, the pipeline becomes $A \rightarrow (B, C) \rightarrow (D, E, A) \rightarrow (F, B, C) \rightarrow (A, D, E)...$ by shifting two stages. However A be invoked before D which causes I/O conflict. Therefore, it must be maximized. Taking the largest number of consistency shift among all retreating edges. Thus, if the stages are shifted by the max consistency shift, all nodes related to retreating edges are invoked correctly. The correct pipeline should become $A \rightarrow (B, C) \rightarrow (D, E) \rightarrow (F, A) \rightarrow (B, C) \rightarrow (D, E) \rightarrow (F, A)...$

Although the condition that a graph has no retreating edge, we need to consider the consistency shift. For example, we consider a straightforward processing flow $A \rightarrow B \rightarrow C$. The pipeline would become $A \rightarrow (B, A) \rightarrow (C, B) \rightarrow (A, B)...$ However, this is wrong because may occur I/O conflict. To resolve this case, we assume that nodes connected by an edge also have a retreating edge. Therefore, the default number of the consistency shift must be two.

In the correct pipeline, we can find two parts. One is the initial stage(s) executed once called *startup*. The other is the contiguous repeating stage(s) called *repeat batch*. In the example case, the stage of (A) is the former. In the case of Figure 28, a set of stages of $(B, C) \rightarrow (D, E) \rightarrow (F, A)$ is the latter. The startup stages include the beginning ones of the pipeline after the consistency shift is performed. Therefore, the number of stages of the startup is calculated by $depth - max\_consistency\_shift$. In the example case, it is $4 - 3 = 1$. Therefore the first stage with A is included in the startup. Other three stages organize the repeat batch. Thus, while the correct pipeline is configured, the startup is invoked once. Then the repeat batch is repeated.

Regarding the cross edges, PEA-ST ignores them because the edge can be treated as tree edge. Cross edge has two or more ancestors. An edge from one of the ancestors should become tree edge.

Let us summarize the processing steps of the PEA-ST explained above. First, one of the root nodes is selected. This makes a graph executable. Second, a spanning tree is created from the root node. Only a tree is generated. Third, the consistency shift is calculated from the retreating edges. Finally, the startup and the repeat batch are generated.

4.3.5. Implementation

Algorithm 3 shows the PEA-ST written by a C-like code. The *main* function processes 1) spanning tree creation, 2) consistency shift calculation, 3) startup and batch creation and finally 4) updating the startup and the repeat batch according to the user-defined requirements. The function checks all available spanning trees by selecting available root nodes. The process 4) will break the loop of selecting the root nodes and return the best startup and repeat batch.

Algorithm 3 Parallelism extraction algorithm applying spanning tree (PEA-ST).

```
struct edge {
    bool connect;
} edges[N][N];

struct node {
  char name;
  int depth;
  bool update;
} nodes[N];

void DFST_Modified(int x){
  pre_num ++;
  NPre[x] = pre_num;
  for(int y=0; y<N; y++){
    if(edges[x][y].connect){          <-- Tree edge
      if(NPre[y] == 0){
        nodes[y].depth=nodes[x].depth+1;
        DFST_Modified(y);
      }
      else if(NPre[x] < NPre[y])      <-- Advancing edge
        edges[x][y].connect = false;
      else if(NRPost[x] == 0){        <-- Retreating edge
        If(max_retreat_offset<nodes[x].depth -  nodes[y].depth)
          max_retreat_offset=nodes[x].depth - nodes[y].depth +1;
        edges[x][y].connect = false;
      }
      else  Edges[x][y].connect = false;   <-- Cross edge
    }
  }
  NRPost[x] = rpost_num;
  rpost_num --;
}

void main(){
  for(int result=0; result<N; result++){
    node_init();
    edge_init();
    root_test(result);
    if(node can reach all other nodes){
      node[result].depth=0;
       DFST_Modified(result);
    }
    else  continue;
    if(there is no retreating edges)
      consistency_shift=2;
    else
      consistency_shift = max_retreat_offset;
    for(i=0;i*move<max_stage;i++){
      for(int j=0; j<N; j++){
        stage= nodes[j].depth+i*move;
        if(stage < max_stage){
          excute[stage][count[stage]] = nodes[j].name;
          count[stage]++;
        }
      }
    }
    if(Parallelism is better than the previous one)
      update(startup and batch);
    else continue;
  }
}
```

The spanning tree creation is performed by *DFST_Modified* function. In the function all edges are categorized to the ones defined by the spanning tree. When an edge is categorized, it updates the *edges* matrix that obtains the tree edge. If an edge is categorized as a retreating edge, it updates the *max_retreat_offset* that obtains the largest offset of stages in the retreating edge (i.e. consistency shift).

After the spanning tree creation, the *main* function calculates the consistency shift. If any consistency shift exists, the *max_retreat_offset* is used. If not, it is two because of the case of straightforward processing flow. Then finally the execute matrix is created. It obtains the stages and the node combinations at each stage. Using the example of Figure 28, let us explain how to calculate the startup and the repeat batch. At the first iteration of *execute* matrix creation, it fills each stage with the nodes of the same depth illustrated by *entry0* in the figure. In the second iteration, it shifts the column index by the consistency shift and fills the nodes of stages again like *entry1*. If the shifted index is larger than the depth of the graph, the iteration ends. The final *execute* matrix is the startup and the repeat batch. The repeat batch is the last n rows of the matrix where n is the consistency shift. The remaining row(s) are the startup.

Finally, user-defined conditions are checked. The conditions depend on the maximum/minimum parallelism, the average parallelism, the number of stages and the smallest consistency shift, etc. In our implementation, we apply the condition to find the largest parallelism of $MIN\_AVR < parallelism < MAX\_AVR$ and $parallelism < MAX$, where MIN_AVR and MAX_AVR are the minimum and maximum parallelisms respectively, and where the MAX is the maximum parallelism. MAX is given by the program-

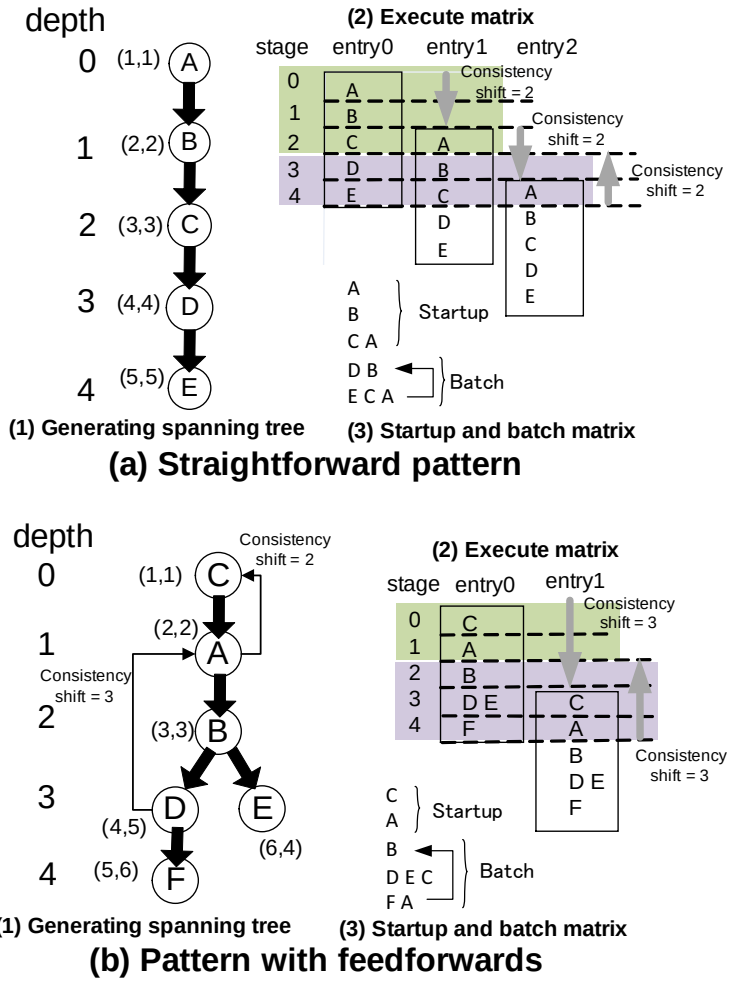


Figure 29. Examples of batch generation using the PEA-ST. A straight forward pattern (a) and the pattern with feedforward edges of Figure 28 when C is selected to the root vertex (b).

mer because of the limitation of the resources (the number of accelerators).

Regarding the complexity of the PEA-ST, it is similar to the spanning tree algorithm. The preorder and the reverse post order numberings takes $O(NE)$ time respectively, where N is the number of nodes and E is the number of edges in the graph. However, the PEA-ST needs to try all available spanning tree creations of the root nodes. Therefore, it becomes $O(NEM)$ where M is the number of available root nodes.

4.3.6. Examples

Here, let us introduce two additional examples as depicted in Figure 29. The straightforward pattern shown in Figure 29(a) needs three iterations to generate an *execute* matrix. The maximum parallelism becomes three. Figure 29(b) shows another example of the same

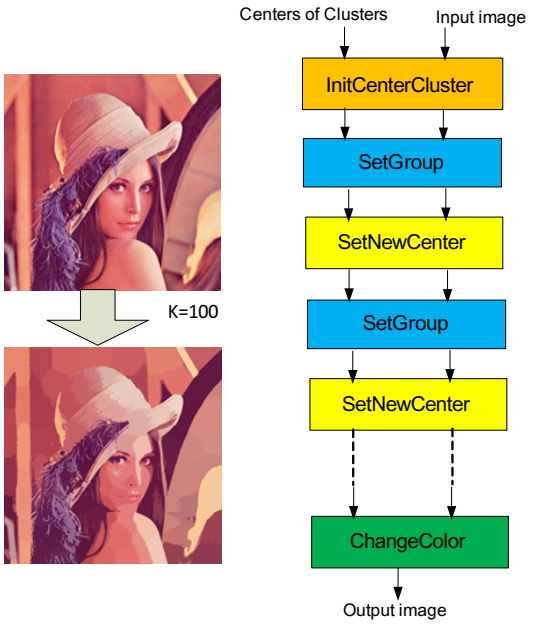


Figure 30. k-means example applying PEA-ST.

processing flow as Figure 28. But the root node is C. This case shows that the maximum parallelism becomes three. It is two when we selected A as the root node. Thus, PEA-ST is very flexible to select an ideal parallelism defined by a programmer according to the limited number of accelerator resources.

4.3.7. Performance evaluation

To validate the PEA-ST, we have made two applications related to image processing. The first one shows the validity of the PEA-ST. The second one shows the impact on the performance aspect. Both applications have the potential ability of achieving small latency to output the final results when the PEA-ST is applied and the processing flows are modified to pipeline flows with parallel executions of multiple tasks (flow-models).

1. Color image quantization

   The first example is a color image quantization. We will show an realistic example when the PEA-ST is applied to a processing flow graph. Color quantization is a task of reducing the color palette of an image to a fixed number of colors k. The k-means algorithm can easily be used for this task and produces competitive results. Figure 30 shows the processing flow. It consists of four kinds of flow-models. The `InitCenterCluster` performs the initialization of the clusters' centers which are randomly selected. The `SetGroup` divides all pixels into k clusters. It calculates the distances from the pixels (RGB) to each center and assigns each pixel to the nearest cluster. The `SetNewCenter` calculates the new centers. Then it compares the old and the new center. If they are different, it continues iterating the loop of

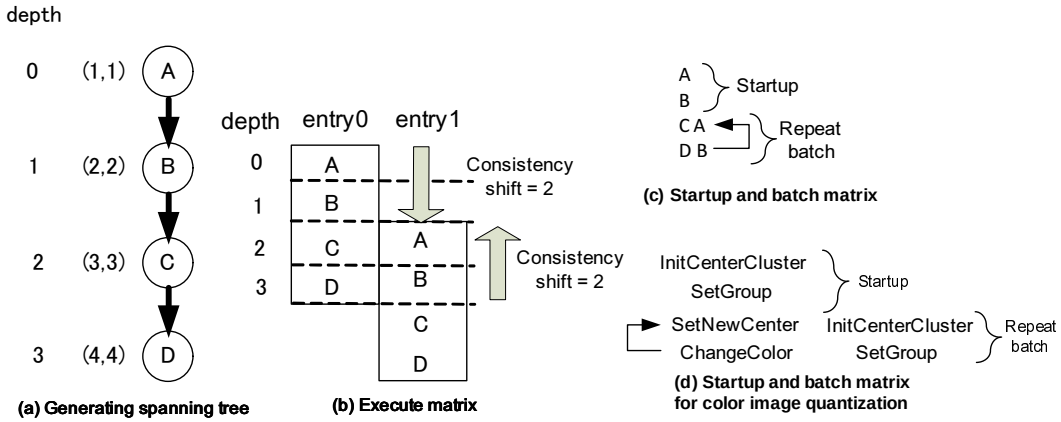


Figure 31. The startup and the repeat batch of the color image quantization application.

the processing steps with the `SetGroup` and the `SetNewCenter` flow-models. Contrarily, if they are the same, it stops the loop and executes the subsequent flow-models. The `ChangeColor` performs a transformation of the color (RGB) of the original image.

Figure 31 shows the pipelined processing flow generated by PEA-ST. Using the figure, let us explain the steps of how to generate pipelined processing flow of k-means example by PEA-ST in detail. Here, we can simplify the structure of the four kinds of flow-models in Figure 30 into the graph of Figure 31(a). Because only the node A can become the root node, only one spanning tree is derived. In other words, the k-means example has just one result of pipeline processing flow according to the PEA-ST. First, the spanning tree generated by the DFST Modified function is shown in Algorithm 3. In the resulting spanning tree, there exists a retreating edge $C \rightarrow B$.Also, we can see that the `max_retreat_offset` of this retreating edge is 2. The `main` function calculates the consistency shift, which is equal to the `max_retreat_offset`. Then the PEA-ST algorithm generates the *execute matrix* with consistency shift. There are four stages and two entries in the execute matrix in Figure 31(b). And at the first iteration of the creation process for the execute matrix, the algorithm assigns the corresponding offset incremented from 0 for the depth to each stage of the nodes from A to D illustrated by *entry0*. In the second iteration, it shifts the depth by the consistency shift of 2, and assigns the nodes of stages again from the shifted depth number as shown in *entry1*. Because the maximum depth is 3 (the original depth of the spanning tree is 3), the nodes is placed in the depth which is larger than the maximum depth. Therefore, the nodes C and D in the *entry1* is canceled. Finally, the startup and the repeat batch are generated from the execute matrix. As shown in Figure 31(c), the repeat batch is the last 2 rows of the matrix because of the consistency shift and it contains all the flow-models of k-means example. And the remaining first two rows are regarded as the startup. The final pipelined result of k-means example is shown in the Figure 31(d). The startup that contains the `SetGroup` and the `SetNewCenter` flow-models is executed once at first. Then

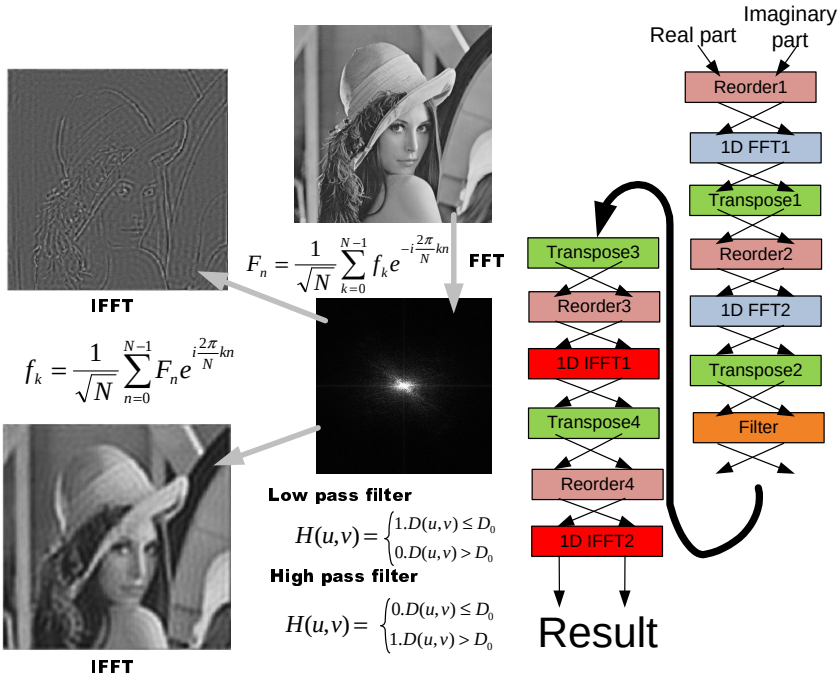


Figure 32. FFT example using PEA-ST.

the repeat batch is repeated.

In this example, the `ChangeColor` outputs the result regarding a single input image. Therefore, if the flow is executed in serial, each flow-model is executed by blocking after the execution because of the I/O data dependencies. Therefore, the total execution time takes the elapsed time of all four flow-models in the execution flow. However, after it is pipelined, the processing flow has been parallelized with two flow-models in a stage. If multiple accelerators are available, the execution time will be improved at most to the one of a single flow-model (`ChangeColor`). Therefore, the PEA-ST approach is effective modification of the processing flow achieving the consistency of the I/O data dependency. Let us see the performance impact on the optimization by the PEA-ST algorithm in the next example.

2. 2D FFT

To evaluate the performance of the PEA-ST, we compare between the serialized and the pipelined processing flows of a common image filtering. Figure 32 shows the processing flow. It consists of four kinds of flow-models. The reorder performs butterfly operation, the FFT and IFFT calculate 1-dimentional FFT and IFFT respectively. The transpose performs a transpose of the 2-dimentional matrix data. The filter is a high-pass or low-pass filter. A CarSh executable of the flow-model is downloaded to accelerator via Caravela runtime. And finally, it is executed by the OpenCL runtime. Figure 33 shows the pipelined processing flow generated by PEA-ST which is packed in two CarSh batches; one is the startup and another is the repeat batch. Totally 13

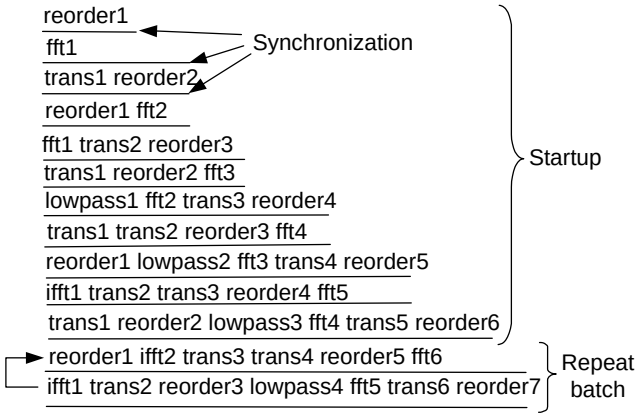


Figure 33. The startup and the repeat batch of the image filtering application resolved by the PEA-ST in the case when the maximum parallelism is seven.

flow-models are executed in a pipeline manner. The maximum parallelism of the processing flow exploited by the PEA-ST becomes seven as shown in Figure 33.

This application also includes the potential performance improvement function after applying PEA-ST algorithm to the original flow with 13 flow-models. The final IFFT2 output the result of the input single image. Therefore, the original flow graph shows 13 steps to execute all flow-models and it need the latency to execute all the flow-models to calculate a single result. However, after pipelining, the latency to calculate the result from IFFT2 becomes small because IFFT2 is executed at every execution of a stage.

The execution time until the final IFFT is measured with/without parallelization. In the case of the serialized version, the 13 flow-models are executed in serial. On the other hand, the pipelined version generates the final image output after the execution of the stage with IFFT2. Therefore, the actual execution time per final result equals to the elapsed time of the repeat batch. The experimental platform is a PC with a Core i7 2.8GHz with 12 GByte DDR2 memory in which a Tesla C2050 is connected via PCI Express bus. Varying the input image size from 128^2 to 1024^2 using OpenCL runtimes on GPU and CPU. We apply the width of input image to the number of the threads at each kernel program. The execution times until the IFFT2 generates the final image are shown in Table 5. Due to the OpenCL runtime overhead, the speedup (serialized/parallelized) of the GPU case is about 25%. The execution time of each kernel program is small. Therefore, the parallelized version promotes the intensive utilization of GPU resource. On the other hand, in the case of CPU, the parallelized version achieves about 4 times higher performance because the parallel threads for different kernels are working concurrently. Therefore, in any accelerator we have confirmed that the parallelized version generated by PEA-ST achieves better performance than the serialized version.

Table 5. Execution times by resulting IFFT comparison among the serialized and the pipelined versions invoked on CPU and GPU. The startup time shows the elapsed times of the startup part of the batch. The repeat time shows the ones of the repeat batch.

Serialized	128^2	256^2	512^2	1024^2
GPU	1.64 sec	1.67 sec	1.77 sec	2.11 sec
CPU	4.19 sec	4.25 sec	4.38 sec	4.77 sec
Pipelined (startup)	128^2	256^2	512^2	1024^2
GPU	1.79 sec	2.18 sec	2.15 sec	1.85 sec
CPU	2.38 sec	2.40 sec	2.46 sec	2.46 sec
Pipelined (repeat)	128^2	256^2	512^2	1024^2
GPU	1.39 sec	1.40 sec	1.41 sec	1.41 sec
CPU	1.12 sec	1.12 sec	1.13 sec	1.14 sec
Speedup	128^2	256^2	512^2	1024^2
GPU	1.18	1.19	1.26	1.50
CPU	3.74	3.79	3.87	4.18

5. Exploiting Potential Performance of Manycore Accelerators

5.1. Swap Execution Method on Manycore Accelerator

5.1.1. Architecture of GPUs

GPU is the most popular example of the platform for executing the stream-based programs. According to the significant performance growth of the GPU, it is expected for a breakthrough for the recent ceiling Moore's low due to applying the multicore/manycore architecture. For example, the NVIDIA's GPU [32] as depicted in Figure 34 integrates *stream processors* that read/write memories where the data streams are stored. The processor has an individual index that corresponds to the one of the data stream. Therefore, each data unit is assigned to individual stream processor. The data stream will access two kinds of memories; *shared* and *global memories*. The shared memory works as a cache for the data in the global memory. Thus, using the data parallelism in the data stream, the GPU processes data units contained in the data streams by assigning the program to the stream processors.

GPU is equipped in the system connected via a peripheral bus like the PCI Express. This means the CPU needs to download the stream-based program to the GPU side, and also it needs to send/receive the input/output data streams to/from the global memory. This causes overhead during execution of the program because the execution must be setup via the peripheral bus. Especially, the recursive program accumulates the overhead at every iteration. Therefore, it is very important to eliminate or to hide the overhead caused when the CPU accesses the resources in the GPU side.

The memories that equip on the GPU and the host sides are separated. Therefore, the recursive application needs to send/receive the required data for the kernel program via the peripheral bus. This data copies take large overhead and occupy large percentage of the total execution time [51] increasing the number of iteration. Thus, this section is focused on this

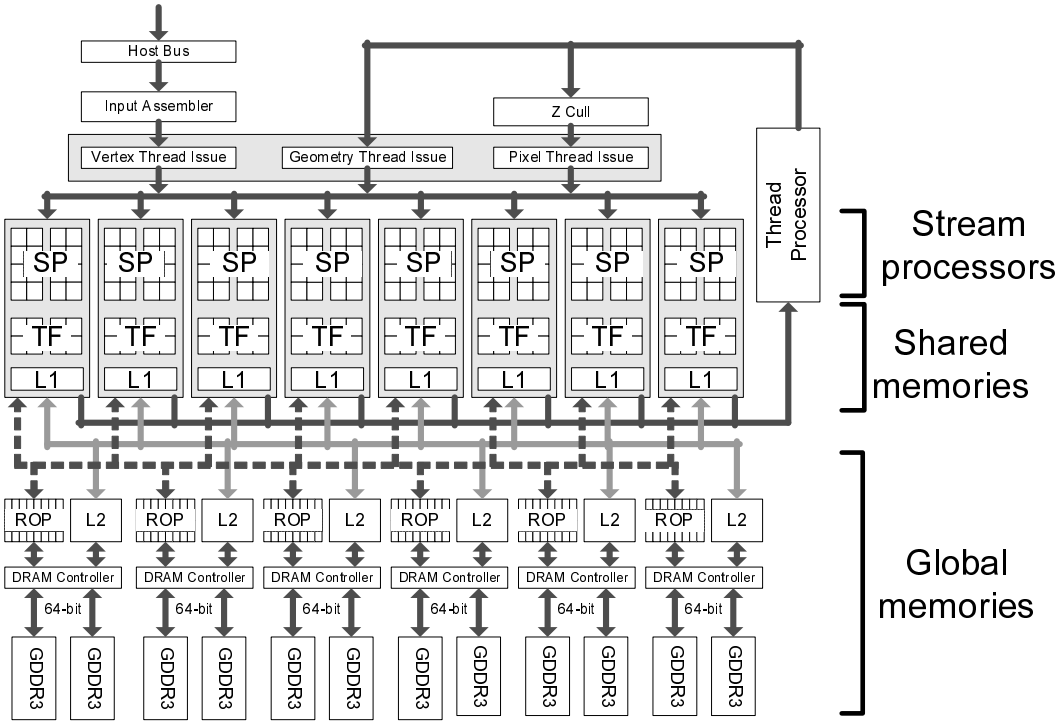


Figure 34. Recent typical architecture of GPU: GeForce Series.

problem and proposes the implementation studies investigating the performance aspects of the recursive applications. The next section will illustrate techniques to eliminate the copy operations.

5.2. Techniques to Eliminate Copy Operations on GPGPU

Let us consider the methods to implement the recursive I/O operations in the stream-based computing on GPU. We can categorize the methods depending on where the copy operation occurs or how the buffer pointers are exchanged as illustrated in Figure 35. We assume that a kernel function receives an input data stream ”a” and produces an output data stream ”b”, and those are exchanged every execution due to the recursive characteristics. We assume that each side of CPU and GPU owns separated buffer for the ”a” or the ”b”. Let us consider how to exchange the I/O buffers after the first execution of the kernel function.

Figure 35 (a) and (b) include copy operation(s). Figure 35 (a), called *Copy host*, copies eagerly the output data stream (1) from the GPU side to the CPU side, (2) between the buffers and (3) from the CPU side to the GPU side. This case is the default method for the recursive kernel function. We need to address this case to achieve better performance. Figure 35 (b) also performs copy operation. However, the copy operation is performed in the device side. Figure 35 (b), called *Copy device*, is another method with copy operation. It copies data on GPU side. This method achieves better performance than the Copy host because it does not need copy operations to/from the CPU side.

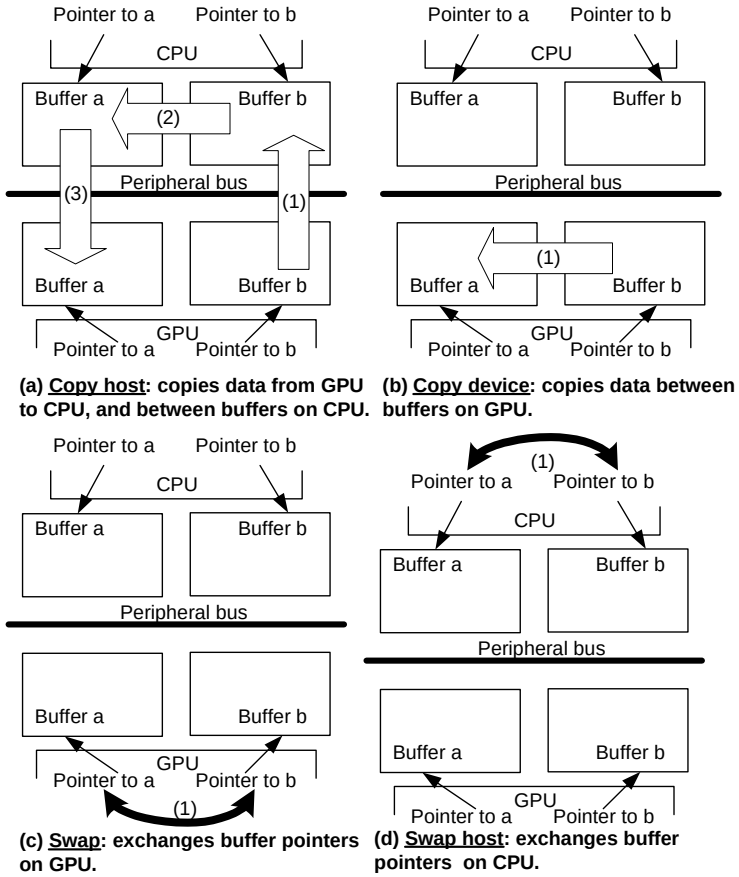


Figure 35. Categorization of methods regarding copy between buffers, and pointer exchange.

To eliminate the copy operation in any part of the system, Figure 35 (c) and (d) exchanges the pointers to the buffers. Figure 35 (c), called *Swap*, exchanges the buffer pointers in the GPU side. This method should achieve optimal performance at every iteration of the recursive kernel because all copy operations are eliminated. This pointer exchange is controlled by the CPU. On the other hand, Figure 35 (d) exchanges the pointers in the CPU side. However, this mechanism needs data transfers from/to GPUs that are performed in Figure 35 (a)-(1) and (3). Therefore, in this section, we do not focus on this method because this method is in terms of performance equivalent to Figure 35 (a).

5.2.1. Case studies of implementations

This section shows how to implement the copy host, the copy device and the swap methods on the CUDA and the OpenCL platforms.

- Case study in CUDA

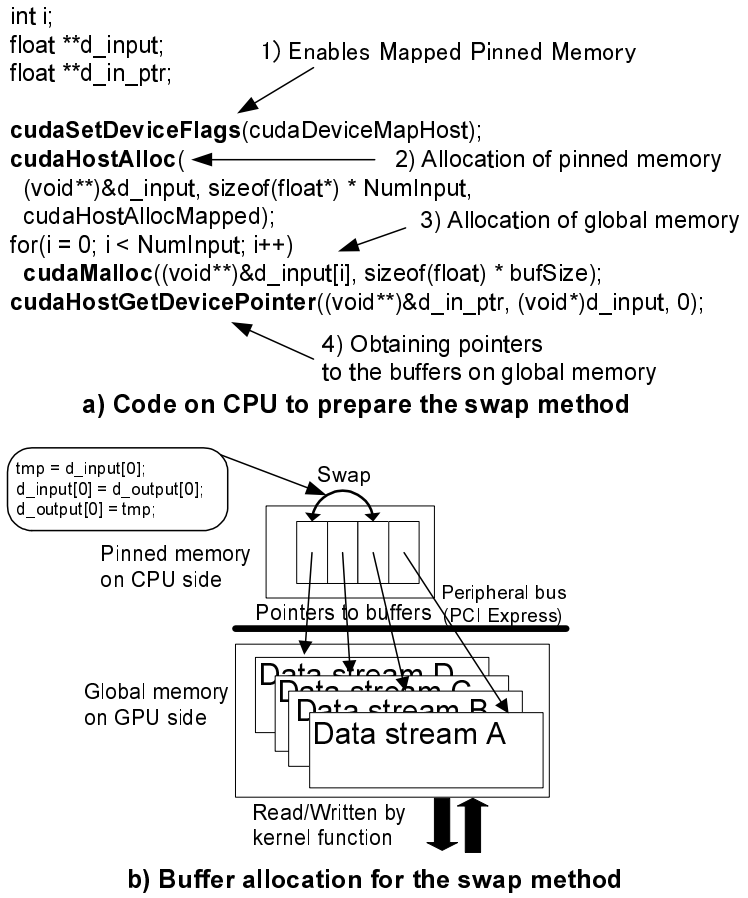


Figure 36. The Swap method on CUDA environment.

The CUDA provides the `cudaMemcpy` function that performs data transfer between the CPU and the GPU sides. Using this function, the Copy host method is implemented with additional copy operation between the I/O buffers in the CPU side. The Copy device method is also available on the CUDA environment using the `cudaMemcpy` function with `cudaMemcpyDeviceToDevice` at the fourth argument. The function performs a copy operation between the buffers allocated on the GPU side. Finally, the CUDA environment enables the Swap method applying a tricky mechanism using the *pinned* memory that is allocated on the CPU side, but that is accessed from the GPU via the peripheral bus as shown in Figure 36. Preparation for the swap method follows the steps as shown in Figure 36a); 1) the `cudaSetDeviceFlags` enables the allocation in the pinned memory, 2) allocation in the pinned memory that obtains the buffer pointers to the data streams placed on GPU side, 3) allocation for the buffers on GPU side and finally 4) registering the pointers to the streams to the buffer of 2). From the 2nd execution of the kernel function, the pointers maintained in the buffer allocated by 2) will be exchanged by CPU side as explained in Figure 36b). This code does not include any copy operation. Therefore, the optimal performance will be expected.

- Case study in OpenCL

  First, the OpenCL implements the copy method using `clEnqueueWriteBuffer` and `clEnqueueReadBuffer`. Therefore, here explains how to implement the copy device and the swap methods. Second, the copy device method is mainly implemented using `clEnqueueCopyBuffer` function that is called by the CPU and performs a copy operation between two buffers in GPU side via the GPU bus. Finally, the swap method is implemented by using `clSetKernelArg` function. This function is used to exchange the parameters of the I/O buffers at each iteration. For example, assume that a kernel has two arguments; `float *inbuf` and `float *outbuf` and two buffers *src* and *dst* are allocated in the GPU side. At the first execution, the `clSetKernelArg` function associates the *src* buffer to the `inbuf` and the *dst* to the `outbuf`. After the execution of the kernel, the function exchanges the `inbuf` and `outbuf` pointers using this function. Thus, the data stored in *dst* buffer is accessed via the `inbuf` without explicit copy operation between buffers.

As explained above, on both the CUDA and the OpenCL environments, the three methods shown in Figure 35 are available to be implemented. According to our experintal measurements, comparing to the data transfer performances on a recent personal computer of PCI Express bus where 10 GByte/sec is achieved at most, the copy method but on the GPU achieves higher bandwidth on which the GeForce GTX285 and the Tesla C2050 obtain about 150 GByte/sec are orbital. According to the static large performance difference, it is easy to expect that the Copy device method achieves very higher performance than the Copy host one. However, the point is how higher performance the swap method can achieve against the copy device method due to elimination of copy operation. Let us discuss this point in the next section assessing dynamic behaviors of kernel functions.

5.2.2. Experimental Performance Evaluations

Let us discuss performance of the three methods explained in the previous sections using dynamic kernel functions: the IIR filter, the Jacobi method and the LU decomposition. Although these applications work recursively, each application has a unique characteristic.

The experimental environment is a PC that consists of an Intel's Core i7 930 processor at 2.80GHz with 12GB DDR3 memory, and an NVIDIA Tesla C2050 with 3GB Memory. The OS in the PC is the Cent OS of the Linux Kernel 2.6.18. The driver version of the GPU is 3.0.

We measure the execution times of each application using the methods when we apply both double precision and single one to the calculation. Regarding the Copy host/device methods, we measure the data transfer time consumed by the copy operation. The time used by the swap operation is also measured. We will show the results on graphs with bars where two parts are included: a gray one shows the time consumed by the copy/the swap operation, and the black one shows the time used for the calculation performed in the kernel.

- IIR filter kernel

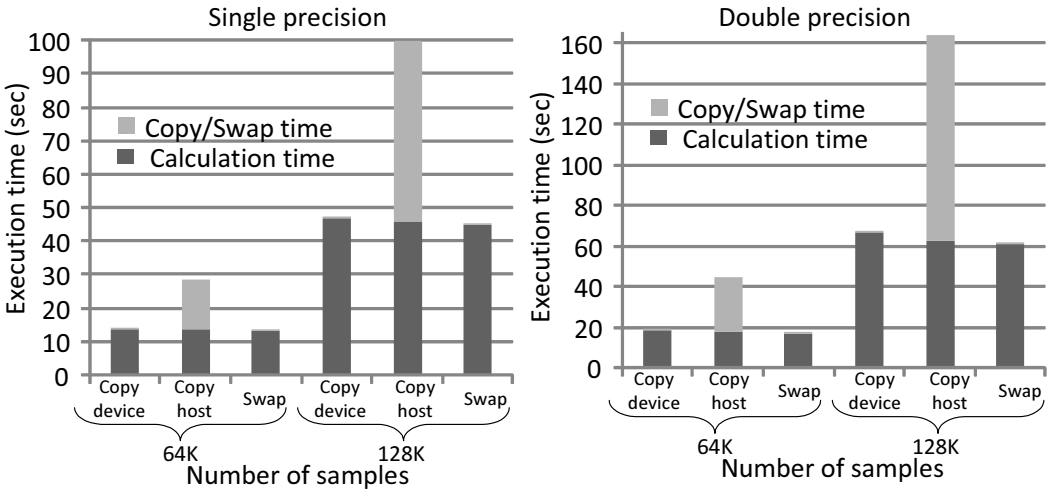


Figure 37. Performance of IIR filter kernel on CUDA.

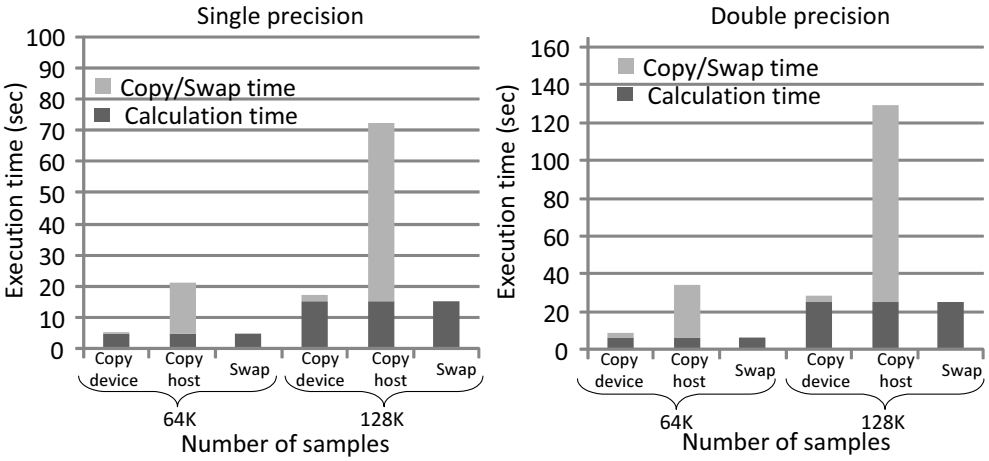


Figure 38. Performance of IIR filter kernel on OpenCL.

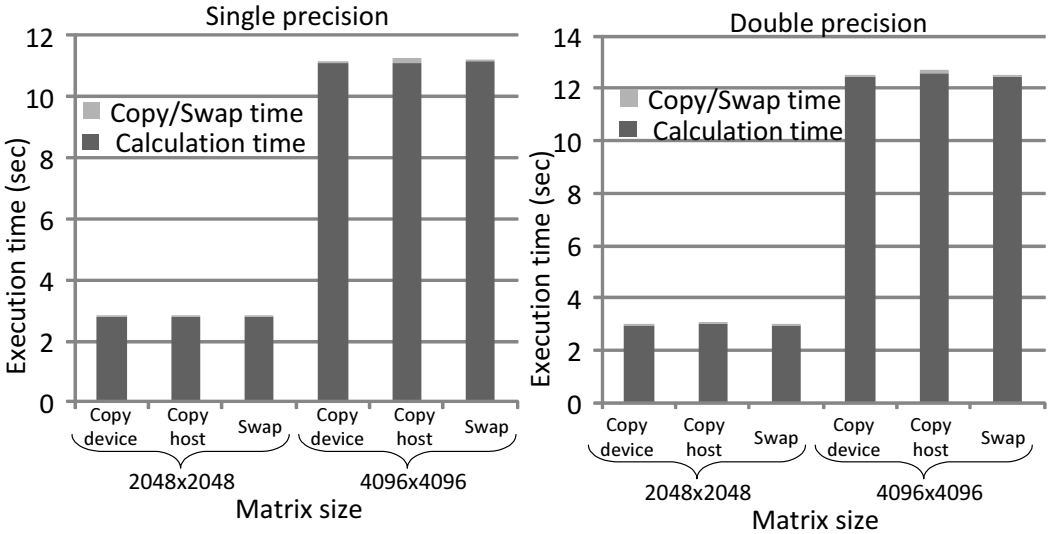


Figure 39. Performance of Jacobi method on CUDA.

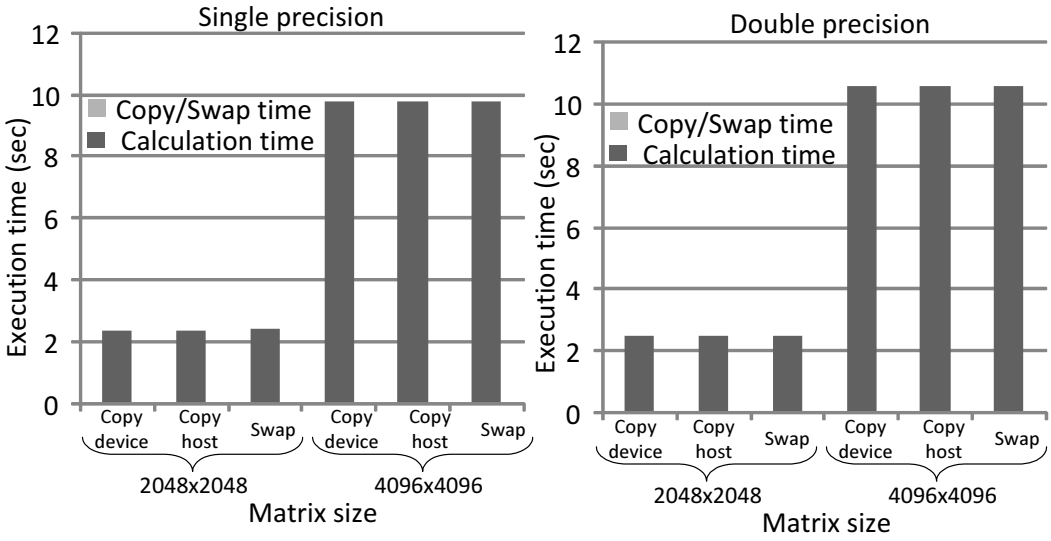


Figure 40. Performance of Jacobi method on OpenCL.

$$\begin{array}{c} \begin{array}{cc} k & n-k \end{array} \\ \begin{array}{c} k \\ n-k \end{array} \left(\begin{array}{c|c} U_{11} & U_{12} \\ \hline L_{21} & \begin{array}{|c} U_{22} \\ \hline L_{22} \end{array} \end{array} \right) \end{array} \qquad \begin{aligned} U_{11} &= A_{11} \\ U_{12} &= A_{12} \\ L_{21} &= \frac{1}{A_{11}} A_{21} \\ L_{22}U_{22} &= A_{22} - \frac{1}{A_{11}} A_{21}A_{12} \end{aligned}$$

Figure 41. Algorithm of LU decomposition.

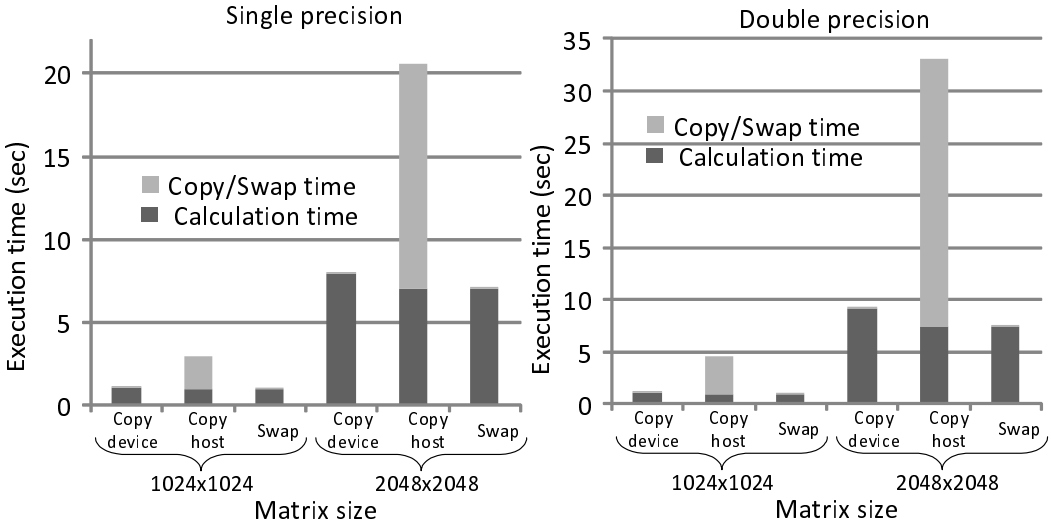


Figure 42. Performance of LU decomposition on CUDA.

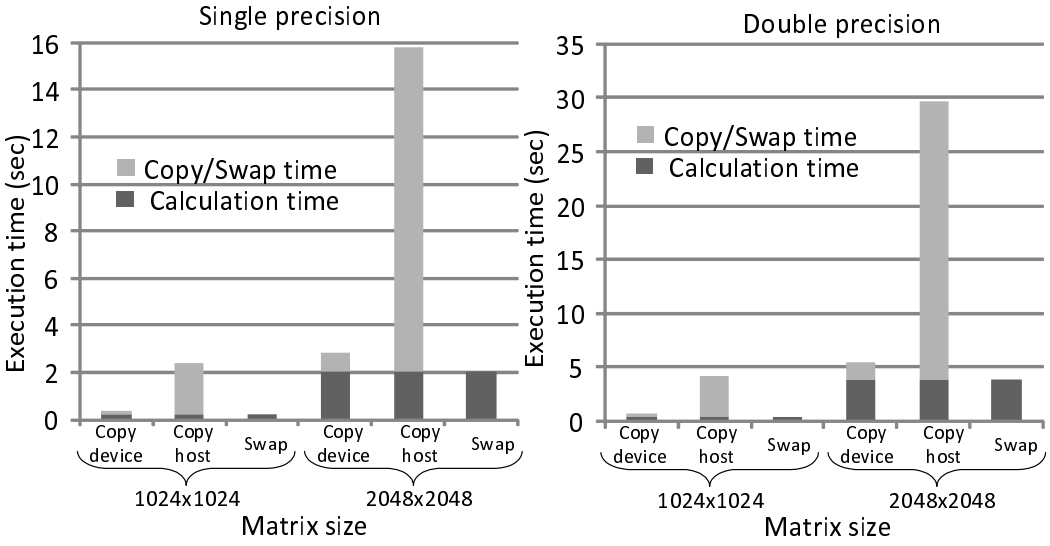


Figure 43. Performance of LU decomposition on OpenCL.

The IIR (Infinite Impulse Response) filter is a well-known kernel to be applied to image processing for emphasizing the edges of objects. We use the following equation for the filter with 16 coefficients:

$$y_n = \sum_{i=0}^{15} a_i x_{n-1} - \sum_{j=0}^{15} b_j y_{n-j-1}$$

Here, the y is used in the right side of the equation. Herein we explain, this is with a typical recursive kernel, which exchanges the input data stream of y and the output data stream of y. This exchange is performed by the Copy host/device and the Swap method. Regarding the parallelization, each y_i calculation is assigned to a stream processor (i.e. a thread in the CUDA and a work item in the OpenCL). The number of total input samples of x and y equivalents to the number of iterations.

Figure 37 and Figure 38 show the execution times for the methods on the CUDA and the OpenCL respectively. During the iteration of the kernel, the size of y does not change. Moreover, the calculation is not heavy. Therefore, the percentage of the copy time in the Copy host method is very large. The copy device and the swap method achieve almost the same time on both environment. Thus, this kernel function has benefit from the elimination mechanism of the swap or the copy device.

- Jacobi Method

The Jacobi method is applied to a system of linear equations where the coefficient matrix is sparse and to be solved approximately with recursive iterations using the following equation:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{j \neq i}^{n} a_{ij} x_j^{(k)})$$

where the $x_i^{(k+1)}$ approximates x_i after kth iteration.

In the kernel function assigned to the thread or the work item, the input data stream for x obtains the approximation after the kth iteration. The output data stream will become the one of the $(k+1)$th iteration. Therefore, the I/O buffers are exchanged among the recursive iterations.

Figure 39 and Figure 40 show the execution times among the methods on the CUDA and the OpenCL respectively. The number of iterations is normalized to the same as the length of x vector. As the opposite case of the kernel function such as the IIR filter, this kernel function does not require a large amount of I/O data for the recursive calculation in comparison to the amount of calculations. Therefore, three methods achieve almost the same performances without copy/swap overheads.

- LU decomposition

  Finally, the LU decomposition is used with the same objective as the Jacobi method to solve the linear equations. However, it is applied to the case when the coefficient matrix is dense. Figure 41 shows the shape of the matrix after the kth decomposition calculating the equations listed in the left side of the figure. The $L_{22}U_{22}$ is generated as the coefficient matrix for the next iteration. This means that the input data stream is the matrix and also generates the matrix as the output data stream for the next iteration. Thus, the output data streams is recursively used in the input every iteration as the size is decreased.

  Figure 42 and Figure 43 show the execution times for the considered methods on the CUDA and the OpenCL respectively. The sizes of the I/O data streams are controlled by a constant argument to pass the iteration number. Even if the I/O data streams are reduced every iteration, this kernel needs to use $k \times k$ matrix at the kth iteration. Therefore, copy overhead is observed in the cases of the Copy host/device methods. Here we confirmed that the Swap method achieves the best performance again.

Any targeted applications that we have focused in this section achieve the best performance when the swap method is applied. The Copy device method shows also drastically better performances than the ones of the Copy host. However, it also causes some overhead to use the GPU's memory bus during the data copy between the I/O buffers. Thus, we have confirmed that the swap operation should be implemented to avoid the overhead imposed when the recursive data transfer between the I/O buffers is included in the algorithm because it does not have any overhead even if the transferred data size is small as we have observed in the case of Jacobi method. When the data size becomes larger, it is clear that the Swap method obtain the best performance.

5.3. Scenario-based Execution Method on Manycore Accelerators

The CarSh has addressed the double programming problem on the CPU and the accelerator. It provides an interface to invoke only the accelerator program. This promotes that the programmer concentrates to develop the applications only focusing on the stream computing paradigm.

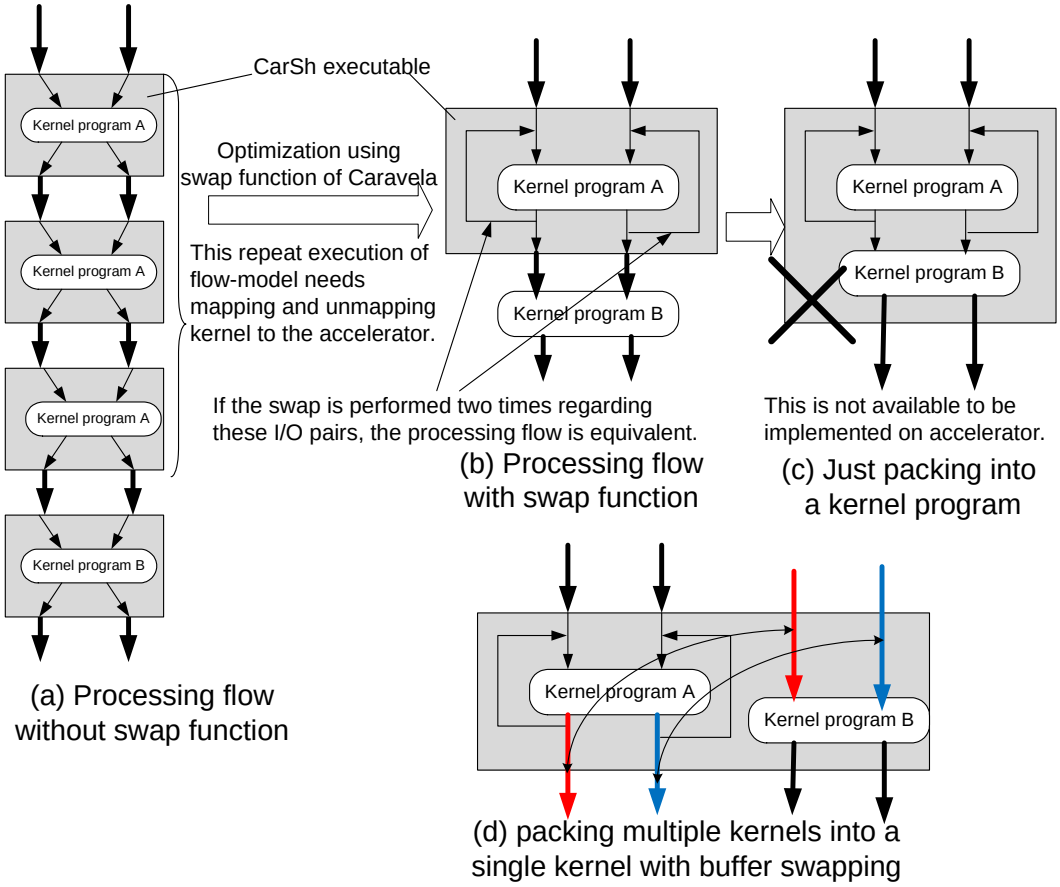


Figure 44. Optimizing repeat execution of a flow-model with the swap function of Caravela.

The buffer management has been also addressed by Caravela and CarSh employing the swap function. For example, in the case when a GPU is employed to an accelerator of a processing flow with multiple kernel programs, Figure 44(a) illustrates a processing flow with a repeat execution of a kernel program packed into a CarSh executable. The *Kernel program A* is packed into an executable file of CarSh and executed for three times by passing the data at every execution. The data propagations among the *Kernel program A* can be optimized by the virtual buffer function that the buffers are provided by memory implementation not by files. This achieves the higher performance of the total execution than the conventional one. However, the flow-model execution needs mapping/unmapping the flow-model to/from GPU at every execution. This causes a large overhead because the control performed by the host CPU is returned at every execution of the flow-model. Moreover, the I/O buffers must be configured for the next kernel execution. This causes the data transfer overhead via the peripheral bus.

If we apply the swap function supported by Caravela platform, we can implement the repeat execution on the recursive structure of the flow-model if the I/O buffers are straightly connected as shown in Figure 44(b). This program execution brings an optimization without unmapping the flow-model and keeps the continuous execution on the accelerator without

returning the control to the CPU side. However, the *Kernel program B* must be mapped to the accelerator after the swap execution of the *Kernel program A*. This also causes the data transfer overhead via the peripheral bus. The kernel merging techniques such as [17] has been proposed. However, it merges multiple kernels and invokes it at a single mapping to a GPU. It is targeted to utilize the resource on the accelerator without considering the processing flow that conveys the results among multiple kernel programs. Therefore we need to invent a novel technique to eliminate the mapping/unmapping overhead.

To address the overhead problem as discussed above, it is indispensable to develop a new technique that different kernel programs are able to run without unmapping it from the accelerator. For example, Figure 44(c) shows the packing situation of the *Kernel program A* and *B* into a *single kernel program*. The I/O connection in a kernel program is not available in the flow-model because the kernel program is not able to randomly read/write the output data due to the assigned processing unit numbers. In this case, the output data stream from the *Kernel program A* must be transferred to the CPU side once, and then must be unmmaped from the accelerator. Then the *Kernel program B* is mapped to the accelerator. The output data stream is set to the input one again. And finally calculation of the Kernel program B can be started. Therefore, the kernel execution must be reset after the execution of the *Kernel program A* and the output must be swapped with the corresponding input data stream to the *Kernel program B* such as Figure 44(d). This strategy eliminates the overhead for the mapping/unmapping a kernel program and buffer configuration. However, this case implies problems of the execution order and how to select the correct kernel program. To address this, the single kernel must have a special input that receives the execution order of the kernel programs in the desired order specified by the programmer. We call this order a *scenario* of a single kernel program with multiple functions. The main objective of this research is to develop a modeling technique to pack the multiple kernels resolving the I/O swapping timings and the one to insert the additional code that merges multiple kernel programs. From the next section, we propose a novel technique called *scenario-based execution method* that provides how to pack multiple kernel programs. This strategy eliminates the mapping/unmapping a kernel program to/from an accelerator and the buffer configuration overheads.

5.3.1. Design of scenario-based execution method

In order to implement the scenario-based execution of kernel program, we need to consider two matters; one is the packing technique of multiple kernel programs and another is the I/O buffer definition for the swap function.

First, let us begin to design the I/O buffer definition for the swap function. Assume that we have a processing flow as illustrated in Figure 45(a). The flow consists of three kernel programs named A, B and C. Each kernel program has four input streams named a, b, c and p. It also has four output streams named d, e, f and q. We assume that, of course, the programmer knows the relations among the input and the output streams. The output streams of A are connected to the next kernel program B and the calculated data are propagated to the output of C. Now let us pack the I/Os into swap pairs. Regarding the input stream p, because it is not connected to the previous kernel program, we do not need to care it as the swap pair. The output stream q is also considered as the same. The other

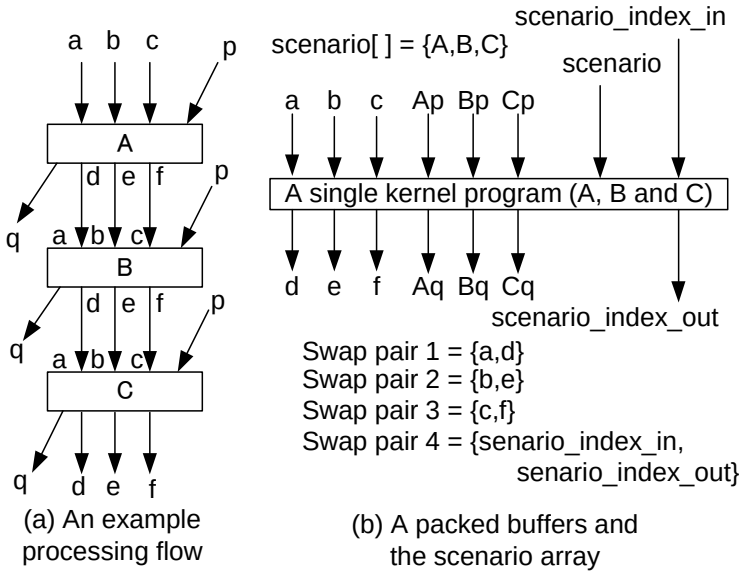


Figure 45. The buffer packing technique for the scenario-based execution.

I/O streams are connected among the kernel programs one by one. If we define the same length of the I/O buffers of a, b, c, d, e and f, we can pack those into three swap pairs. Therefore, we can define the *swap pair 1* with a and d, the *swap pair 2* with b and e, and the *swap pair 3* with c and f. Thus, we can pack the kernel program A, B and C into a single one with three swap pairs, and also the single kernel program just enumerates the input streams p and the output streams q of A, B and C as illustrated in Figure 45(b). Here, we need to introduce additional input data streams to the single kernel. One is the scenario, which has a set of sequence that implements the processing flow. In the example case, it is an array with three integer elements of A, B and C. Another is the scenario index, which is also an array with one element that represents the index number for the scenario array. The index is incremented at every execution of the single kernel. Therefore, its output stream is indispensable and also those input and output streams must be defined as another swap pair for controlling the behavior of the kernel program.

According to the buffer packing technique explained above, at every execution the I/O buffers defined as the swap pair will be exchanged and also the input data is automatically propagated from one function to another in the processing flow following the scenario. The scenario is also pointed by the scenario index that the redundant output buffer will be exchanged with the input buffer after incrementing the index number. Thus, the buffers can be packed completely to a single kernel program and the single kernel program implements the processing flow.

Next, let us consider the kernel merging into a single program. In order to select a corresponding calculations of each kernel program packed in the single kernel program we can merge all kernel programs using a selection statement of the language used by kernel program such as *if else* or *switch*. Because it is easy to shape the selection flow we use *switch* statement. The scenario is selected by the index and also will become the selection

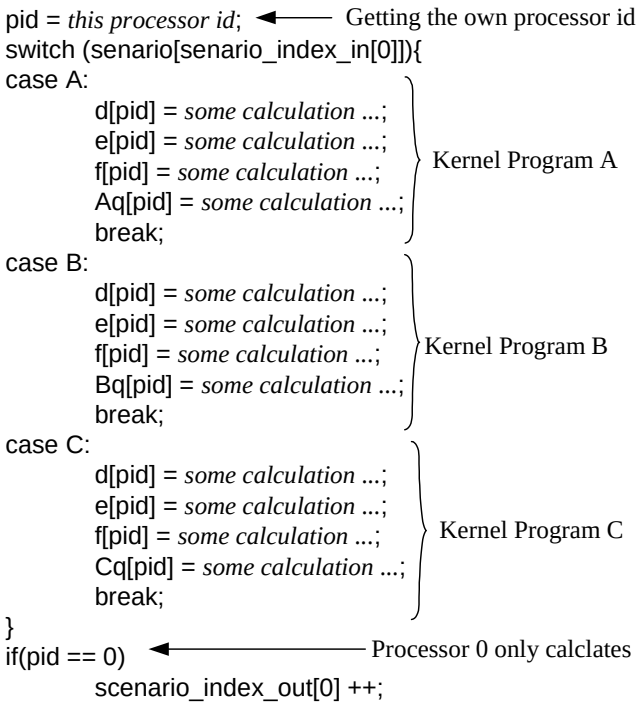


Figure 46. The kernel merging technique for the scenario-based execution.

key for the corresponding function for the kernel program.

Figure 46 shows the perspective of the single kernel program written in a C-like code, which packs the processing flow illustrated into Figure 45(a). The `pid` is the processor ID given by the accelerator's runtime. Each scenario is selected by the scenario index, and jumps to the corresponding calculation. In each case, it performs the calculations of the corresponding `case` statement and generates all output data streams that will be swapped after the kernel execution. Besides, the output data stream not related to the swap pair is generated at the corresponding calculations in the `case` statement. For example, the output stream of `Cq` is generated at the `case` statement of "C". However, it does not calculate the one of `Aq`. Because the output data steam is not volatile during swapping the I/O buffers and iterative execution of the single kernel, the output data will be picked up by the host CPU. Finally, the first processor specified by the `pid` increments the `scenario_index_out` that will be used as the next scenario index.

According to the buffer packing and the kernel merging techniques explained above, a processing flow can be packed into a single kernel program with the scenario input. Every execution of the single kernel program, it is not unmapped from the accelerator, and then, the buffers related to the swap pairs are exchanged without copy operation. Because the host CPU just commands to the accelerator the pointer swap operation of the related buffers and re-execution of the same kernel program, the data transfer via the peripheral bus is reduced to the minimal. Thus, the scenario-based execution is expected to exploit the potential performance of the accelerator eliminating the inevitable overheads conventionally caused in the general execution by mapping/unmapping kernel programs.

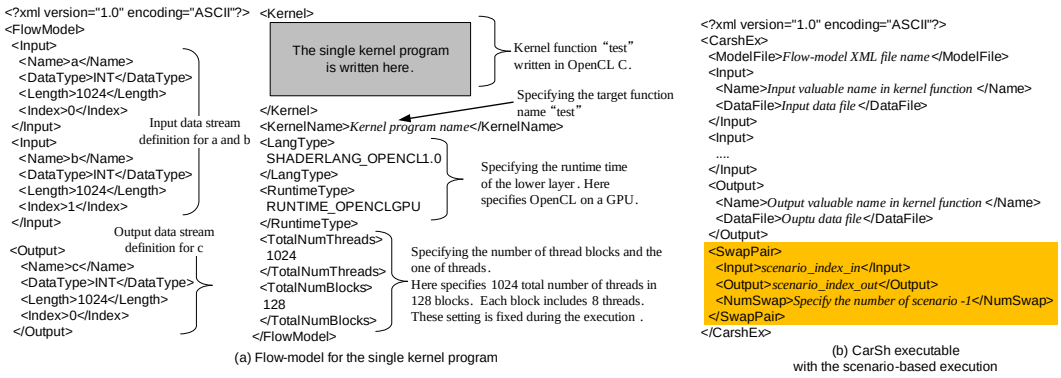


Figure 47. Implementation of the scenario-based execution on CarSh framework.

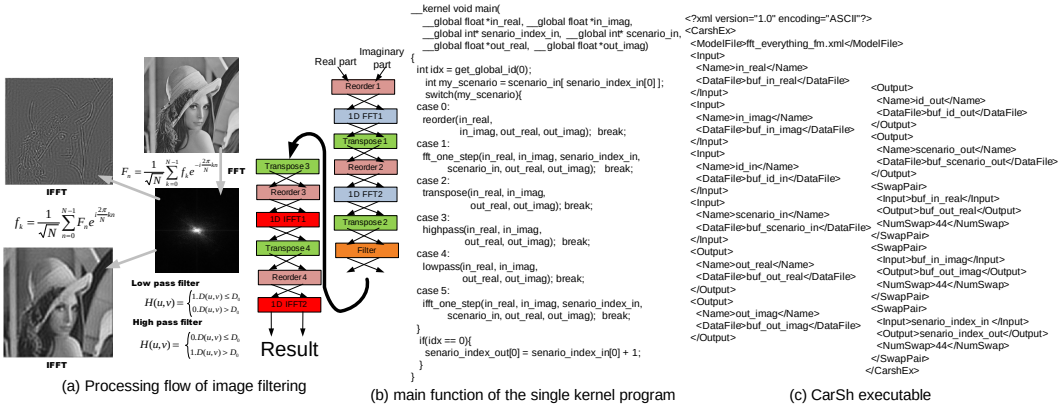


Figure 48. Image filtering application and the CarSh executable.

5.3.2. Implementation of scenario-based execution method

For the implementation of the scenario-based execution, we employed CarSh environment. As listed in Figure 47(a), first the programmer needs to define a flow-model and packs the single kernel program. In the flow-model, there is no definition of the swap pair. Next, the CarSh executable is defined as listed in (b) of the figure. The CarSh executable includes the relations among the I/O data streams defined in the flow-model and the I/O files that are read/written in order to execute the kernel program. It also includes the swap pair definitions of the data I/Os and the scenario index. The `<SwapPair>` tag includes the number of swaps. It equals to $(N-1)$ where the N is the number of recursive iterations in the scenario stream. Finally, the programmer needs to setup the input data streams preparing CSV files and also the scenario in the same type of the file.

The CarSh executable invokes the single kernel program embedded in a flow-model reading the swap pair definitions and the number of swap exchanges. It reads the input data streams and the scenario, and thus the execution will be iterated for the specified number of swap exchanges without unmapping the kernel program from the target accelerator.

5.3.3. Restrictions of scenario-based execution

The scenario-based execution has the limitations of the freedoms of parallelization and the buffer utilization as explained below:

- Limitation to the freedom of the number of threads
  When the parallelism would change among kernel programs in a processing flow, the scenario-based execution is not able to run those kernel programs equivalently. The parallelism must be the same among all kernel programs because the single kernel program is not able to change the number of threads by itself after the invocation. Therefore, the number of threads of the single kernel program must be defined as equal to the maximum one in the kernel programs. This means that the programmer needs to program any kernel program in the processing flow with the same number of threads (of course, the kernel program should support any number of threads due to the extensibility).

- Limitation to the I/O buffer size
  The buffer swapping needs the buffers which sizes are the same among the input and the output buffers. If one of the sizes of buffers in a pair does not match to another, the amount of data elements generated by the kernel program will overflow. This means that the buffer sizes of the swap pair must be equal. Therefore, the size of the amount of data outputted from any kernel program in a processing flow is set to the largest number of elements in the I/O buffers of the swap pair.

In some cases such as the application that changes the parallelism at every execution of the kernel like LU decomposition, the programmer would desire to write the kernel program that can change the parallelism and the buffer sizes with the input parameter such as the iteration index. However, in spite of this kind of complex case, the scenario-based execution will achieve the best performance because the copy operations are eliminated during the execution of the single kernel program, and thus it becomes a novel method of the high performance execution to exploit potential performance of any kinds of manycore accelerators.

5.3.4. Experimental evaluations

Let us evaluate the scenario-based execution using a realistic example. Here we use a typical image filtering with 2D FFT performed often in the image operations. We assume to perform the filtering an $N \times N$ pixel image with a high or a low pass filter. Figure 48(a) shows the processing steps used in the evaluation. An image (*Lena*), which is transformed by the FFT, is passed to a high or low pass filter, and finally transposed by IFFT. This simple process is composed by five flow-models: *reorder* performs butterfly exchanges, *transpose* inverses the rows and the columns, *filter, FFT* and *IFFT*. Totally it executes 13 functional steps in a processing flow as shown in Figure 48(a). The FFT and IFFT needs iteration to perform the butterfly operations. The number of iteration is calculated by $log_2 N$ when the input image size is $N \times N$ pixels.

We pack the processing pipeline to a single kernel with the scenario. Figure 48(b) shows the main function of the single kernel program. The scenario changes depending

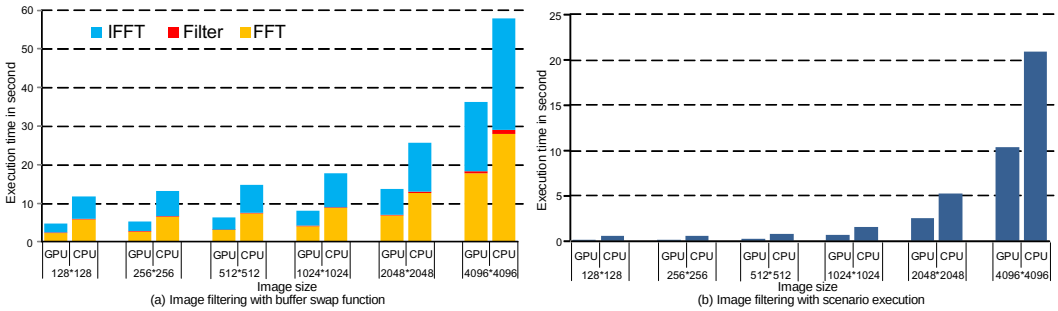


Figure 49. Comparisons of execution times of the image filtering application (a) with and (b) without scenario-based execution.

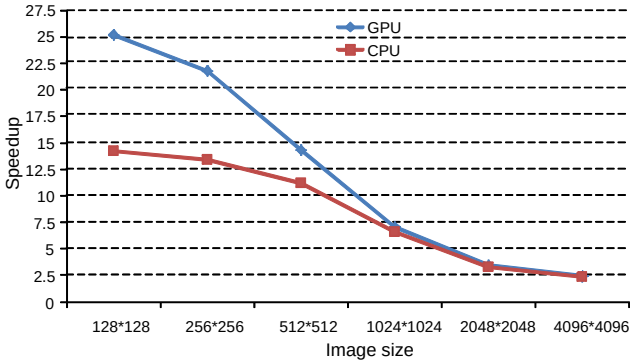


Figure 50. Comparison of speedups between with/without scenario-based execution.

on the input image size because the numbers of iterations of the FFT and the IFTT are varied by the input image size. We have assigned the scenario numbers from 0 to 5 to the $reorder$, the single step of *FFT*, the *transpose*, the *high pass filter*, the *low pass* filter and the single step of $IFFT$ respectively. The scenario input stream obtains the numbers in the processing order specified in Figure 48(a). For example, when the N is 512, the numbers of iteration of FFT and $IFFT$ becomes $log_2 512 = 9$ respectively. Therefore the scenario becomes a sequence of 0, repeating 1 for nine times, 2, 0, repeating 1 for nine times, 2, 3 or 4, 2, 0, repeating 5 for nine times, 2, 0, repeating 5 for nine times. Total number of the swap iterations becomes 45. The `scenario_index_in` is used to point the current scenario and the corresponding *case* statement is selected by the number.

Regarding the buffer packing, the I/Os of the real part and the imaginary part are defined as the swap pairs declared in `<SwapPair>` tag combined with `buf_in_real` and `buf_out_real`, and with `buf_in_imag` and `buf_out_imag` respectively as listed in Figure 48(c). The CarSh maps the single kernel program to the accelerator following the specification of the executable, and performs the buffer swaps. At every execution of the program, the scenario index is incremented and it points the case number from the scenario. Thus, all processing steps of the flow are executed in the accelerator side only.

Let us evaluate the performance using execution times with/without applying the scenario-based execution. Our platform of the performance test is a PC with a Core i7

930 at 2.80GHz with an Nvidia Tesla C2050 GPU. CarSh uses the OpenCL runtime to access both CPU-based and the GPU-based accelerators. We measure the performances on both accelerators.

The first experiment analyzes the baseline performance of the processing flow applying the image filtering. This experiment does not use the swap function in Caravela. Therefore, each flow-model in the processing flow is mapped/unmapped to the accelerator. The execution flow of the multiple kernel programs is given by a CarSh batch using the virtual buffer function passing the real and the imaginary data streams are propagated from the first $reorder$ to the final $IFFT$ as shown in Figure 48(a). The performance is shown in the graph of Figure 49(a). Because each kernel program in the flow will be unmapped once and the control will be returned to the host CPU side, it is available to measure the execution time of each kernel program as the graph consists of different colors. As we can see the performance of each part of the flow, we confirm that almost all total execution time is owned by FFT and $IFFT$ due to the copy operations performed by the butterfly operations.

Next, we measured the execution times with applying the scenario-based execution. Figure 49(b) shows the execution times of the performance. During the small sizes of the input image, it drastically achieves the performance than the one without the scenario-based execution. The performance is linearly increasing when the image size becomes double. Moreover, the execution time scales to four times. This means that the accelerator is intensively working to calculate the kernel program. When CPU is used for the accelerator, we also confirmed the performance becomes improved by the swap operation due to the elimination of copy operation because the swap operation on the CPU-based OpenCL accelerator is implemented by just exchanging pointers of buffers.

Figure 50 shows the speedup (with/without scenario). When the input image size for the filter application is small, the speedup shows in both cases very high such as 25 times in the case of GPU and 15 times in the one of CPU. This represents that the overheads are eliminated, which are caused by the mapping/unmapping of the kernel program and the data copy operation from a kernel program to another.

According to the performance evaluation in this section, we have confirmed that the scenario-based execution achieves very reasonably high performance in any processing flow. When we apply two simple techniques discussed in this section, which are the buffer packing and the kernel merging techniques, the processing flow has become an iterative execution of a single kernel program without unmapping it from the accelerator. Therefore, the overhead caused by the data transfer between the host CPU and the accelerator via the peripheral bus is completely eliminated. Thus the scenario-based execution is an indispensable technique to provide the performance goal of the application designer.

6. Remarks

This chapter showed an advanced programming environment of manycore accelerators and techniques to exploit the potential performance of these accelerators. The Caravela platform was therefore introduced. It simplifies the programming environment of the accelerators using the flow-model concept. The flow-model was employed to represent algorithms based on the stream computing approach. The Caravela platform is capable of parallelizing the flow automatically and exploiting the potential performance of the accelerators. Even if the

flow is concurrently separated into several physical accelerators as in a GPU cluster, the timings of message exchange and kernel execution are automatically adjusted by the Caravela runtime. Moreover, Caravela resolves the overhead of data copy operations among the host CPU and the accelerator using the swap method, after which the execution mechanism is extended to the scenario-based execution using the swap operation. Thus, we can conclude that the Caravela platform completes the fundamental techniques necessary to use the latest manycore accelerators.

For the future, we plan to extend the techniques presented in this chapter to the hardware implementation. The hardware is an intelligent manycore accelerator that analyzes the processing flow automatically and creates the execution order in a pipeline manner using the PEA-ST algorithm. Then the flow can be optimized by using the scenario-based execution mechanism. In this case, if the CPU side provides an algorithm definition of the pipeline model to the intelligent accelerator, the accelerator will work autonomously without further communicating with the host side.

References

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

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

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

[4] Barracuda: An OpenCL Library for Ruby. http://gnuu.org/2009/08/30/barracuda-an-opencl-library-for-ruby/.

[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] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

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

[8] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *NUMERISCHE MATHEMATIK*, 1(1):269–271, 1959.

[9] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne 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.

[10] Michael L. Fredman and Dan E. Willard. Trans-dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths. *J. Comput. Syst. Sci.*, 48(3):533–551, June 1994.

[11] Alan O. Freier, Philip Karlton, and Paul C. Kocher. *The SSL Protocol Version 3.0*. Netscape communications corporation, 1996.

[12] James Frey, Todd Tannenbaum, Ian Foster, Miron Livny, and Steven 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.

[13] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[14] Naga K. Govindaraju, Scott Larsen, Jim Gray, and Dinesh Manocha. A memory model for scientific algorithms on graphics processors. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 89, New York, NY, USA, 2006. ACM Press.

[15] Hannes Gredler and Walter Goralski. *The Complete IS-IS Routing Protocol*. SpringerVerlag, 2004.

[16] William Grosso. *Java RMI*. O'Reilly Media, 2001.

[17] Marisabel Guevara, Chris Gregg, Kim Hazelwood, and Kevin Skadron. Enabling task parallelism in the cuda scheduler. In *Workshop on Programming Models for Emerging Architectures*, PMEA, pages 69–76, Raleigh, NC, September 2009.

[18] Paul Havlak. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst.*, 19(4):557–567, 1997.

[19] OpenMP homepage. http://www.openmp.org/.

[20] 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.

[21] Vijay Karamcheti and Andrew A. Chien. Software overhead in messaging layers: where does the time go? In *ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 51–60, New York, NY, USA, 1994. ACM Press.

[22] 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.

[23] John Kessenich, Dave Baldwin, and Randi Rost. The OpenGL Shading Language. *3Dlabs, Inc. Ltd.*, 2006.

[24] Polina Kondratieva, Jens Krüger, and Rüdiger Westermann. The Application of GPU Particle Tracing to Diffusion Tensor Field Visualization. In *IEEE Visualization*, page 10, 2005.

[25] Pablo Lamilla, Shinichi Yamagiwa, Masahiro Arai, and Koichi Wada. Elimination Techniques of Redundant Data Transfers among GPUs and CPU on Recursive Stream-Based Applications. In *IPDPS/APDCM11 Anchorage USA*, May 2011.

[26] LAM/MPI Parallel Computing. http://www.lam-mpi.org/.

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

[28] Thomas J. Misa and Philip L. Frana. An Interview with Edsger W. Dijkstra. *Commun. ACM*, 53(8):41–47, August 2010.

[29] Kenneth Moreland and Edward Angel. The FFT on a GPU. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 112–119, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[30] J. Moy. OSPF Version 2. Internet RFC 2328, April 1998.

[31] Aaftab Munshi, Benedict R. Gaster, Timothy G. Mattson, James Fung, and Dan Ginsburg. *OpenCL Programming Guide*. Addison Wesley, 2011.

[32] Hubert Nguyen. *GPU Gems 3*. Addison-Wesley Professional, first edition, 2007.

[33] NVIDIA Corporation. CUDA: Compute Unified Device Architecture programming guide, http://developer.nvidia.com/cuda.

[34] OpenCL. http://www.khronos.org/opencl/.

[35] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Kruger, Aaron E. Lefohn, and Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005.

[36] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Kruger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005.

[37] Radia Perlman. An Algorithm for Distributed Computation of a SpanningTree in an extended LAN. In *Proceedings of the ninth symposium on Data communications*, SIGCOMM '85, pages 44–53. ACM, 1985.

[38] Radia Perlman. *Interconnections (2nd Ed.): Bridges, Routers, Switches, and Internetworking Protocols*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[39] R. C. Prim. Shortest Connection Networks and some Generalizations. *The Bell Systems Technical Journal*, 36(6):1389–1401, 1957.

[40] Daniel A Reed, Celso L Mendes, Chang da Lu, Ian Foster, and Carl Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure - Application Tuning and Adaptation*. Morgan Kaufman, 2003.

[41] Leonel Sousa and Shinichi Yamagiwa. aravela: A Distributed Stream-Based Computing Platform. In *Proceedings of 3rd HiPEAC Industrial Workshop*, 2007.

[42] Tarjan R. E. Testing flow graph reducibility. *J. Comput. Syst. Sci. 9*, pages 355–365, 1974.

[43] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196. Springer-Verlag, 2002.

[44] N. Doss William Gropp, E. Lusk and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. In *MPI Developers Conference*, 1995.

[45] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison Wesley, 1996.

[46] S. Yamagiwa and Shixun Zhang. Scenario-based execution method for massively parallel accelerators. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on*, pages 1039–1048, July 2013.

[47] Shinichi Yamagiwa. Invitation to a Standard Programming Interface for Massively Parallel Computing Environment: OpenCL. *International Journal of Networking and Computing*, 2(2):188–205, 2012.

[48] Shinichi Yamagiwa and Leonel Sousa. Caravela: A novel stream-based distributed computing environment. *Computer*, 40(5):70–77, 2007.

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

[50] Shinichi Yamagiwa and Leonel Sousa. Modelling and Programming Stream-based Distributed Computing based on the Meta-pipeline Approach. *Int. J. Parallel Emerg. Distrib. Syst.*, 24(4):311–330, 2009.

[51] Shinichi Yamagiwa, Leonel Sousa, and Diogo Antao. Data buffering optimization methods toward a uniform programming interface for gpu-based applications. In *CF '07: Proceedings of the 4th international conference on Computing frontiers*, pages 205–212, New York, NY, USA, 2007. ACM Press.

[52] Shinichi Yamagiwa and Shixun Zhang. CarSh: A Commandline Execution Support for Stream-based Acceleration Environment. In *Procedia Computer Science, Proceedings of the International Conference on Computational Science, ICCS 2013*. ELSEVIER, 2013.