

Design and implementation of a high-performance stream-based computing platform on multigenerational GPUs.

By Pablo Lamilla Álvarez

September 27, 2010

Supervised by:

Professor Shinichi Yamagiwa
Kochi University of Technology, Japan

Professor Francisco José Abad Cerdá
Universidad Politécnica de Valencia, Spain

Table of Contents

Chapter 1 Introduction.....	4
1.1 Overview of the Project.....	4
1.2 Objective of research.....	6
1.3 Expected Outcomes.....	7
1.4 Disposition of the document.....	7
Chapter 2 Background and Definitions.....	9
2.1 Stream computing.....	9
2.2 GPGPU.....	10
2.3 GPGPU computing platforms.....	13
2.3.1 Caravela Platform.....	14
2.3.2 OpenCL.....	17
Chapter 3 Stream-based computing platform on multigenerational GPUs.....	27
3.1 Caravela implementation.....	27
3.2 Caravela low-level functions.....	28
3.3 Flow Model.....	33
3.3.1 Constant values structure.....	35
3.4 Flow Model Creator.....	36
3.5 Kernel Structure.....	38
3.6 Swap Mechanism.....	39
3.7 Implementation conclusions.....	41
Chapter 4 Performance Evaluation.....	42
4.1 Straightforward application: Matrix multiply.....	42
4.2 Recursive application: IIR filter.....	46
Chapter 5 Conclusions and future work.....	50
Bibliography (references).....	52
Appendix I User guide of Flow Model Creator GUI.....	55
1 Introduction.....	55
1.1 FlowModelCreator.....	55
1.2 Authors.....	55
2 Functionality.....	56
2.1 Basics concepts.....	56
2.2 Top Window.....	57
2.3 Set constant values window.....	61

List of Figures

Figure 1. A typical system organization with legacy GPU	10
Figure 2. Graphics processing steps	11
Figure 3 Recent GPU architecture.....	12
Figure 4. Structure of the flow-model	14
Figure 5. Resource hierachy in a processing unit.....	15
Figure 6. Swap mechanism.....	17
Figure 7. Companies that support OpenCL.....	18
Figure 8. Platform Model of OpenCL	18
Figure 9. Example of 2-dimensional NDRange	20
Figure 10. Conceptual OpenCL architecture	21
Figure 11. VectorAdd example in OpenCL kernel language	23
Figure 12. OpenCL host code VectorAdd example. Part one	24
Figure 13. OpenCL host code VectorAdd example. Part two	25
Figure 14. Structure of the Caravela functions.....	28
Figure 15. Flow Model of VectorAdd example	35
Figure 16. Changes in the top window of FlowModelCreator.....	37
Figure 17. New interface for introducing the constant values.....	38
Figure 18. Example of kernel header for Caravela.....	39
Figure 19. Execution times using matrix multiply	43
Figure 20 MatrixMultiply Kernels	44
Figure 21 Execution times using IIR filter	46
Figure 22 Kernel code of IIR program	47
Figure 23. VectorAdd execution times with 5000 swaps.....	48
Figure 24. VectorAdd execution times with 10000 swaps.....	49

List of tables

Table 1. Main functions of Caravela library.....	16
Table 2. Environment for the evaluation	42

Chapter 1 Introduction

1.1 Overview of the Project

During this decade, high performance computation demand has been increasing more and more, for example in the field of humanities [1]. Scientists and investigators are in need of high speed and performance environments for their research, which need to perform millions of floating points operations per second [2].

One way to achieve this goal is to increase the power of the hardware. Multi-core CPUs and Supercomputers [3] are an example of the evolution of this path. However, even this type of hardware (Supercomputers) has their limits. Supercomputers, besides being very expensive and complicate to build, have a very high electricity consumption [22].

In the meanwhile, another kind of processors, the GPUs, have experimented a great improvement within this decade, and a single GPU IC chip is able to achieve 10 TFLOPS. This performance attracts HPC researchers to the potential computing power of GPUs, so in the last decade the field called GPGPU [17] (General-Purpose computation on Graphics Processing Units) has gained more importance. However programming on the GPU entails a great difficulty, due to its architecture, that is categorized into two types nowadays.

One type of GPU architecture has three kinds of processors called, *vertex processors*, *rasterizer* and *fragment processors* that are dedicated respectively to transform the vertices that define the graphics primitives, to transform a primitive into a set of pixels and finally to compute each pixel color, taking into account an illumination equation and maybe a texture. This graphics pipeline can be programmed using graphics libraries such as DirectX 9[7] and OpenGL 2.0 [8]. Also, this architecture is generally called *legacy* architecture, to distinguish from the newest architectures that have a different structure.

These newest architectures integrate tens or hundreds of a standardized general purpose stream processor that can execute any type of *shader*. When the computational units are unified like this, the architecture is called Unified Shading Architecture. Each

processor invokes a program to generate element(s) of output data stream(s) from element(s) of input one(s). Emulating the three-step graphics processes performed by the former architecture, this recent architecture implements the similar operations of the graphics runtimes. In addition, it releases the stream processor resources to general purpose computing via special runtimes such as CUDA [11] and OpenCL [12].

During the era of the legacy GPU architecture the main problem of GPGPU applications was the complete different style of programming compared to CPU programming, due to the fact that legacy GPUs environments were originally designed for graphics processing. Therefore some applications that tried to cover and abstract those differences for the programmers were developed such as Brook [9] and Sh [10]. Brook introduced a programming model for GPUs called *stream computing*.

Stream Computing [4] is a programming paradigm in which the data is processed in a continuous way, as a unique piece of information. This piece is called an element of *stream*. The point in the stream computing is to manipulate and to operate each element of the stream in a parallel pipeline, i.e. all at the same time. The operations applied to the stream are packed into a function called *kernel*. For instance, if we want to sum the elements of two vectors ($\text{output}[i] = \text{inputA}[i] + \text{inputB}[i]$), using the old-sequential style the elements have to be summed one after another inside of a loop. However, using stream computing, each vector is treated as a stream and the sum operation is programmed inside a kernel. Then, this kernel will receive each stream and will apply the sum operation to all the elements of these streams in a parallel way, so the elements of the resulting vector are obtained all at once.

Another GPGPU platform, the Caravela [13], was developed to form the true stream-based computing based on the *flow-model* [14]. This model, used by the Caravela in its execution, is defined by the number of I/O streams, constant values and a program (kernel) invoked on a targeted GPU .

The problem with these architectures is that their programming style is not compatible because for programming in the recent architecture one must follow a stream-based computing style, which is not the same style that is followed when programming in the legacy architecture. Therefore algorithms or optimization techniques cannot be shared between architectures. However the Caravela platform has a unified framework to implement stream-based computing using the flow-model, just

by defining the number of I/O streams and the target program. This flow-model also can be defined easily with a GUI called FlowModelCreator, which is included in the Caravela package. Thus, the Caravela platform can provide a unified programming style between different GPU generations, because the stream-based computing concept is standardized.

This project is focused on the migration of the Caravela platform to OpenCL. Now the Caravela is implemented on the DirectX9 and the OpenGL environments, but the OpenCL support, as a standard and platform-independent language, will allow the Calavera to run into any modern GPU environment. Therefore with this migration process the following problems will be solved:

- 1) Impossibility to run the Caravela into any GPU environment.
- 2) Disparities in the programming style of the runtimes of the legacy and recent GPU architectures.

This project will be developed in C/C++. C# will also be used in the Graphical User Interface (GUI) that allows the creation of the “Flow Model”, the structure that packs all the information used by Calavera.

1.2 Objective of research

There are three main objectives of this research project as listed below:

1) OpenCL support in Caravela platform.

This objective will cover the necessity of Caravela platform to run on any GPU environment. To achieve this objective, the relationship between the OpenCL functions and the Caravela functions will be studied and established.

2) OpenCL support in FlowModelCreator GUI.

This objective will allow the FlowModelCreator GUI to automatically generate a flow-model XML file with all the new parameters that OpenCL implementation needs.

3) **Exploiting maximum performance from GPUs in any generation.**

This objective will focus on preserving the performance of Caravela execution on GPUs of any generation.

1.3 Expected Outcomes

There are two outcomes expected by this research project.

1) **A software package that includes the new Caravela software with OpenCL support.**

This package will include the Caravela platform with the new functionalities to support OpenCL environment, as well as the support of the previous environments (OpenGL and DirectX).

2) **Multi-Platform “FlowModelCreator” GUI with OpenCL option.**

This GUI application will generate the flow-model structure for Caravela. With the release of the new version, the changes that allow the support of OpenCL will be implemented.

3) **Application examples for the Caravela.**

Programs samples used for the performance evaluation are also added. This includes the matrix-multiplication and the IIR filter kernel programs, both written in OpenCL, and the main programs using the Caravela API functions to execute the kernels.

Thus, with the package of these outcomes, a high-performance stream-based platform on OpenCL will be developed.

1.4 Structure of the document

Chapter 2 explains the background of this project’s field, specifically “Stream computing” and GPGPU, as well as a description of the two main environments used in this project, the “Caravela” and the OpenCL.

Chapter 3 illustrates the implementation of the new version of Caravela. Technical problems encountered along the migration process and their solutions are explained in detail.

Chapter 4 presents the results obtained with the new software. Various performance evaluations are carried out, and their results are commented.

The last chapter describes the conclusions for this project and proposes future directions.

Finally in the appendix, the User's Guide of the application and the GUI "FlowModelCreator" are attached at the end of this document.

Chapter 2 Background and Definitions

2.1 Stream computing

Stream computing (or stream processing) is a computer programming paradigm that allows some applications to easily exploit a limited form of parallel processing [4]. These applications are able to use multiple computational units without the necessity of explicitly managing allocation, synchronization, or communication among those units.

Stream computing takes advantage of a SIMD [21] (Single Instruction, Multiple Data) architecture, where the same instruction can be applied to various instances of different data.

The main point of stream computing is to use a continuous flow of data called *stream* as the input and output of the program. A stream is a collection of data which can be operated on in parallel. Each element of a stream is a record of data requiring a similar computation; however it is, in general, independent of the other elements. A series of operations will be applied to each element in the stream all at once. These operations are packed in a function called *kernel*. Typically, the same kernel function is applied to all elements in the stream (Uniform streaming), but it is not the only option.

The benefit of stream computing stems from the highly parallel architecture of GPUs, whereby tens to hundreds of parallel floating points operations are performed with each clock cycle.

For this reason, nowadays stream computing is primarily used in the realm of the GPUs, where stream computing can easily take advantage of the large number of parallel processors of GPUs. Stream computing on the GPU was mainly used for graphics purposes, but in the recent years, there has been an increasing interest to use it for general purpose applications [5]. This recently new high performance computing field is called GPGPU [17] and is explained in the next section.

2.2 GPGPU

As mentioned above, GPGPU is the technique of using a GPU to perform general purpose, i.e. not only for computer graphics, computation in applications that traditionally have been executed in the CPU. This field has become popular in recent years due to the high demand of high performance computation platforms. Also the results achieved in fields like Bioinformatics [15] are good enough to continue this path in the future.

The evolution of this field is directly related to the evolution of GPUs, specially their architecture and the communication with the host memory and the CPU. Figure 1 illustrates the typical organization around a legacy GPU.

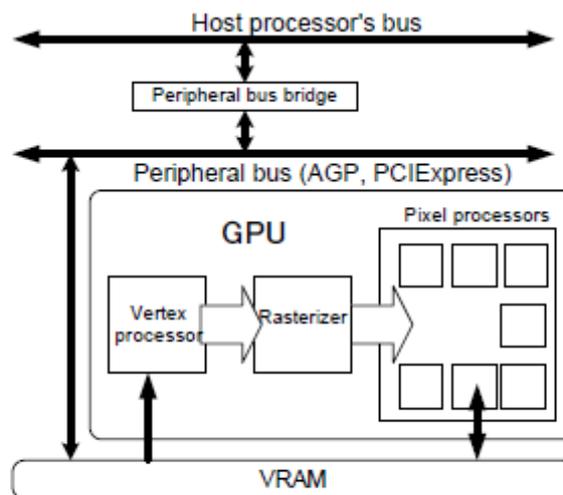


Figure 1. A typical system organization with legacy GPU

In this figure, a video adaptor that includes a GPU and a Video RAM (VRAM) is connected to a peripheral bus of a CPU and the main memory, linked to the “host processor’s bus”. The GPU inside the video adaptor is controlled by the CPU to execute a part of the rendering tasks in the systems.

In order to use the GPU as a computing resource for GPGPU applications, the CPU downloads the application program to the GPU's instruction memory and also has to prepare the input data for the program. This data is copied into the VRAM by the CPU and then the GPU reads/writes the VRAM directly to execute the calculations. Finally

the CPU has to read again the VRAM to copy the results written by the GPU to the main memory.

In recent GPUs, it is possible to access the main memory directly. The area is *pinned* [16] not to be moved by a paging function of the operating system. With this method, there is no need to perform copy operations to transfer data from the main memory to the device memory.

Next, let's see in detail the two different architectures of a GPU mentioned in the introduction: the legacy architecture and the new architecture.

The legacy architecture is based on the graphics pipeline showed in figure 2.

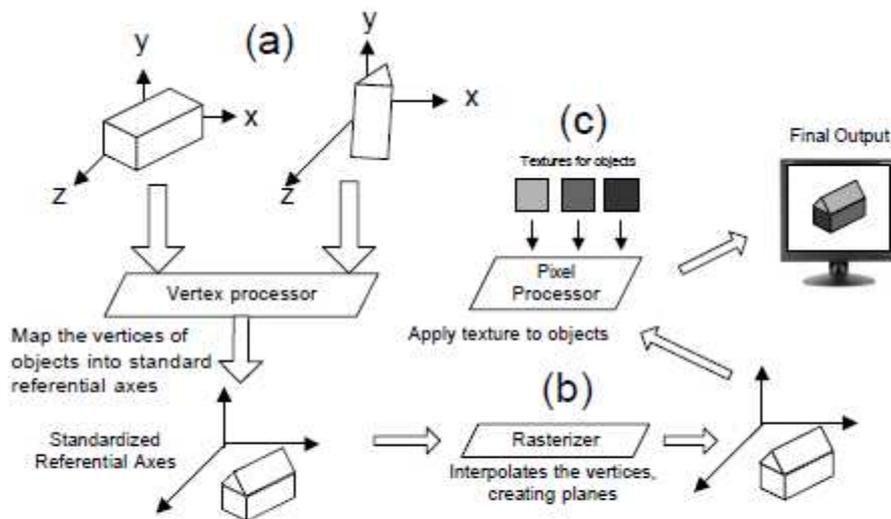


Figure 2. Graphics processing steps

This figure shows the processing steps done by the GPU to create a graphical image to be later displayed in a screen. First, the graphics data is prepared as a set of normalized vertices of objects on a local coordinate system defined by the graphics designer (Figure 2(a)). The vertices are sent to a vertex processor to apply different transformations, such as translations, scales and rotations. In this step all the objects will be mapped to a standardized referential axis (the Camera Coordinate System). Further operations will apply a perspective transformation and will project the 3D vertices of the objects into a 2D plane. In the next step, a rasterizer interpolates the coordinates and defines fragments that represent the graphics objects (Figure 2(b)). Finally, a pixel processor receives those fragments from the rasterizer and creates color data to send to the frame buffer, after calculating the composed RGB colors from the textures of the

objects (Figure 2(c)). The color data is written into the frame buffer, which is connected with the pixels of the screen.

The GPGPU with this architecture used graphics runtimes such as OpenGL and DirectX that operate in the CPU domain in order to control the GPU. The problem was that, because these environments had been firstly designed for graphics processing, the environments had an interface that non-graphics programmers were not familiar to.

For instance, when an application wants to output an NxN matrix, the programmer must setup a frame buffer and must understand that the output will be written to an NxN pixel plane in the frame buffer. Despite this method being correct for graphics applications, for programmers that are not familiar with these graphic programming details it is hard to understand and to write non-graphics related programs. For this reason, there was a necessity of APIs that hides this graphical legacy of GPU.

Nowadays there is another type of architecture for GPUs, as shown in figure 3.

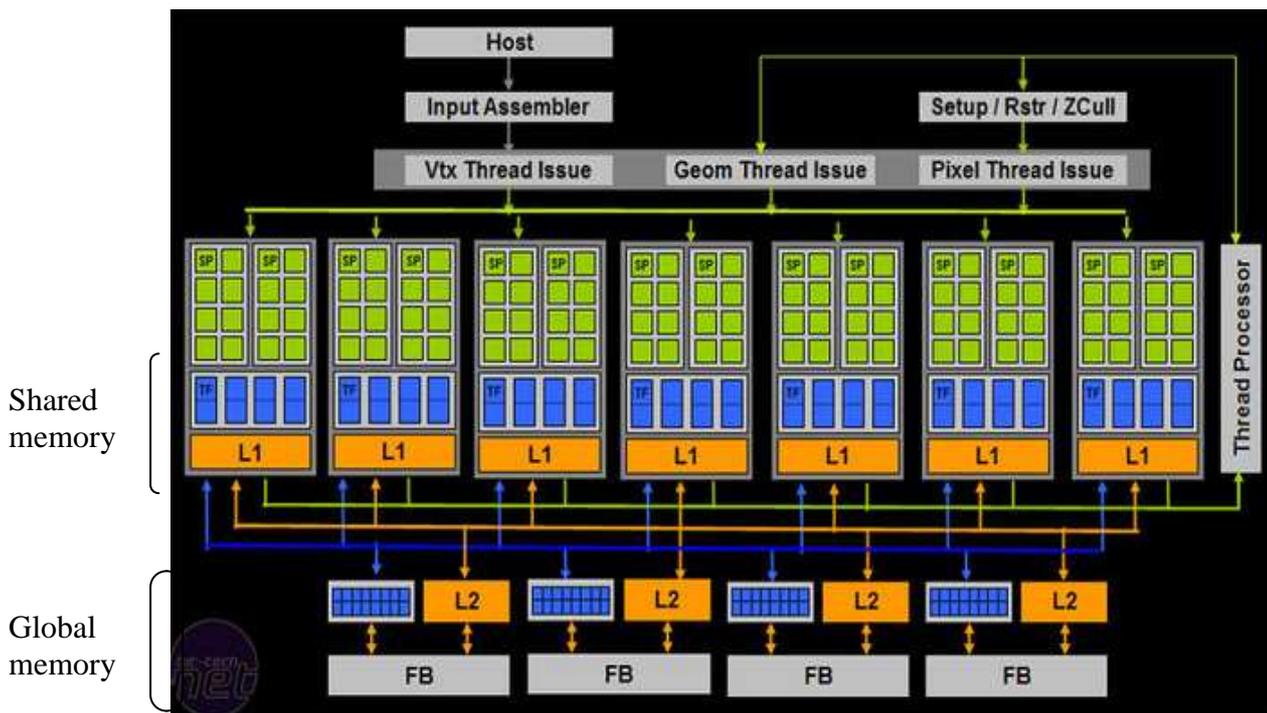


Figure 3. Recent GPU architecture

The recent GPU architecture has only a kind of processor called *stream processor* or *thread processor*. This type of processor can be configured to perform any stage in the graphics pipeline. However, the computing style follows to stream-based one with distributing elements of streams into multiple stream processors. Employing a dedicated

stream-based programming interface (CUDA, OpenCL, DirectCompute [23]), the graphics processing on the new architecture is emulated by the OpenGL and the DirectX providing the equivalent interfaces to the vertex and the fragment processors.

Furthermore, in this architecture, memory has been divided into two types called *global* and *shared* memory, as shown in figure 3. Global memory is provided by the memory placed outside the GPU such as DDR3 VRAM. On the other hand, shared memory is placed besides of the stream processor to work as a cache.

This new architecture is being supported by many runtimes for GPGPU (CUDA, OpenCL or DirectCompute) that hides the problems of disparities in the programming style between GPU and CPU. Moreover, the new distribution of the architecture is more suitable for GPGPU than the legacy one. Thus, the application treated in this project, Caravela, must give support to this new architecture to take advantage of the new features that helps GPGPU.

2.3 GPGPU computing platforms.

In the recent years, researches in the HPC (high performance computing) field have been giving efforts to use GPUs for a supercomputer platform as tried in the web site [17]. As explained in the previous section, during the era of legacy architecture, there was the problem of disparities between graphics runtime environments and general purpose processing on GPGPU applications. To solve this problem some solutions have been proposed, such as Sh, Scout, Brook and Caravela. Sh [10] is a graphics processing interface with an object oriented interface for C++. Scout [18] is another wrapper for graphics which uses a language based on C* [24]. Although details of graphics runtimes are hidden by these two systems, they are still targeted for visual applications. Thus, graphical dependent issues cannot be completely eliminated. Brook [9] was a compiler-oriented interface for GPU-based applications in which the programmer just needs to identify functions to be transposed to programs on the fragment processor specified with a special keyword *kernel*. This computer style is inherited by the newest GPGPU environments (CUDA, OpenCL, etc). Finally the Caravela provides a stream-based computing platform in which the programmer can just concentrate in the design of a

flow-model data structure that follows a stream computing manner. Thus, implementation details such as the calculation of the GPUs are hidden.

This project is focused on Caravela and its porting to OpenCL language in order to be compatible with any GPU environment. In the next two sections, Caravela and OpenCL are explained in detail.

2.3.1 Caravela Platform

The Caravela platform is an interface for stream-based computing that uses the concept of flow-model, shown in Figure 4.

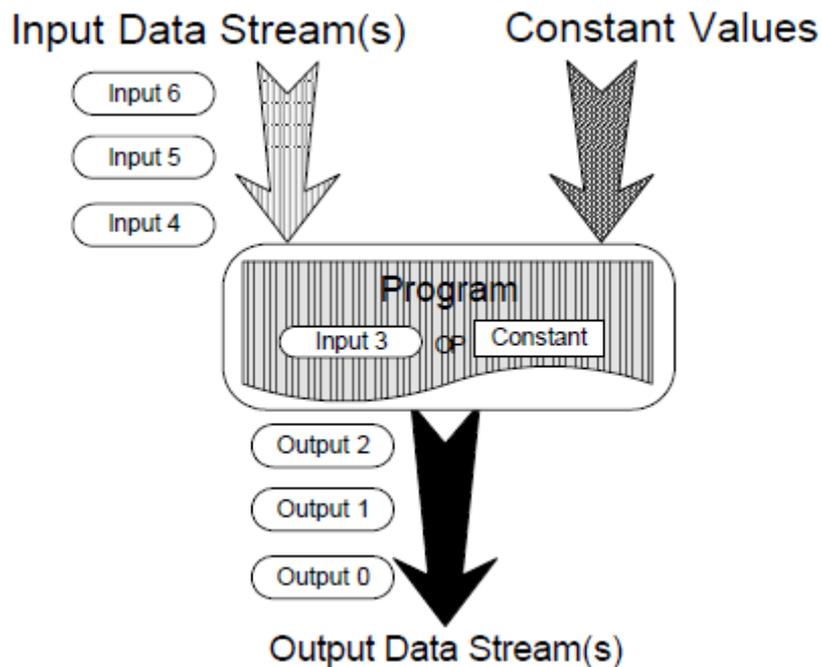


Figure 4. Structure of the flow-model

The flow-model is composed of input/output data streams, constant parameter inputs and a program which processes the input data streams and generates the output data streams. Therefore, it can be applied to any kind of stream processor such as the ones in the new architecture of GPUs. The program part of the flow-model can include multiple stream-based programs suitable for supported environments. Currently the Caravela supports OpenGL and DirectX, so the program can be written in GLSL or HLSL.

To create the flow-model, the Caravela environment provides a GUI to help in the creation process. This GUI is called “FlowModelCreator” and, a user guide is provided in Appendix A.

The Caravela is composed by a library that supports an API for GPGPU. Figure 5 shows the resource hierarchy in this library. The Caravela library has been adapted to the definitions for the processing units represented in Figure 5

- Machine:** host machine of a video adapter.
- Adapter:** video adapter that includes one or multiple GPUs.
- Shader:** a GPU.

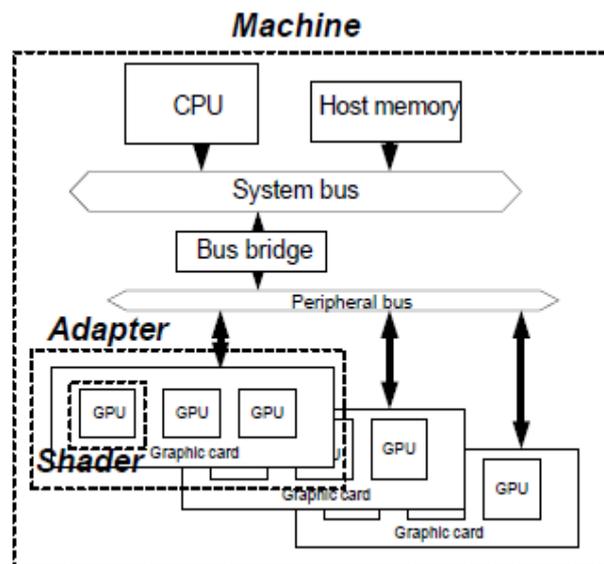


Figure 5. Resource hierarchy in a processing unit.

An application just needs to map a flow-model into a shader to execute it.

The main functions of Caravela library are shown in Table 1.

Table 1. Main functions of Caravela library

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

With these functions, the programmer can implement target applications in the framework of flow-models just mapping flow-models into one or more shaders. Thus, the programmer does not have the necessity of knowing about graphics runtime environment details, so Caravela can become to a solution to relieve the problem of disparities between graphical environments mentioned in Section 2.2.

Moreover, the Caravela platform incorporates optimization functions of the flow-model execution itself, called *swap mechanism* [19], which allows executing recursive iterations of a flow-model exchanging the input and the output buffers in the GPU side without copy operations between the host memory and the GPU memory. Avoiding this data transfer between the host memory and the GPU memory, we reduce the execution time of the program considerably. Figure 6 shows this mechanism.

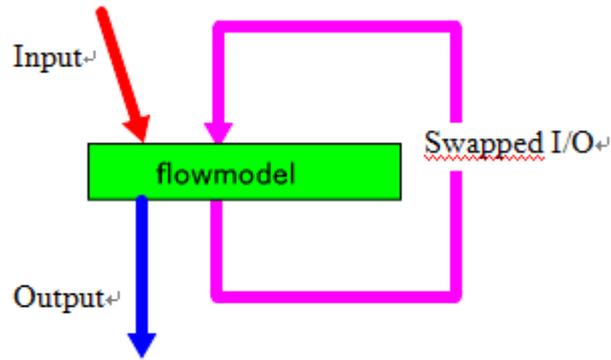


Figure 6. Swap mechanism

Currently, as mentioned before, the Caravela platform supports the legacy architecture of GPUs, so graphics runtime functions must be used to perform the stream-based computation. Also, recently Caravela has been ported to CUDA, so support for the recent architecture of GPUs has been added. However, because CUDA was developed by NVIDIA, it can be used only on NVIDIA GPUs. Thus, we must give support to Caravela for a runtime that can be used with any GPU regardless of the manufacturer, and the chosen language is OpenCL.

The next section gives an overview of this API.

2.3.2 OpenCL

OpenCL (Open Computing Language) is an open royalty-free standard, initially proposed by Apple and finally developed by The Khronos Group [20], for general purpose parallel programming across CPUs, GPUs and other processors. Moreover, its most important characteristic compared for example to CUDA, is that OpenCL is cross-platform. It also supports a wide range of applications, from embedded and consumer software to HPC solutions, through a low-level, high-performance, portable abstraction.

Besides Khronos Group, many industry-leading companies and institutes have participated and have supported the development of OpenCL. Figure 7 shows many of them.



Figure 7. Companies that support OpenCL

The objective of OpenCL is to help expert programmers, such as library writers and middleware vendors, to write portable yet efficient code. Therefore OpenCL provides a low-level hardware abstraction as well as a framework to support programming.

To explain the main ideas behind this language, we will describe these four models for OpenCL: i) platform, ii) memory, iii) execution and iv) programming models.

i) Platform Model

The platform model for OpenCL is shown in Figure 8. A host connected to one or more *OpenCL devices*, such as GPUs, CPUs... Each device is divided into one or more *compute units*. These compute units are further divided into one or more *processing elements* which is where the computation occurs.

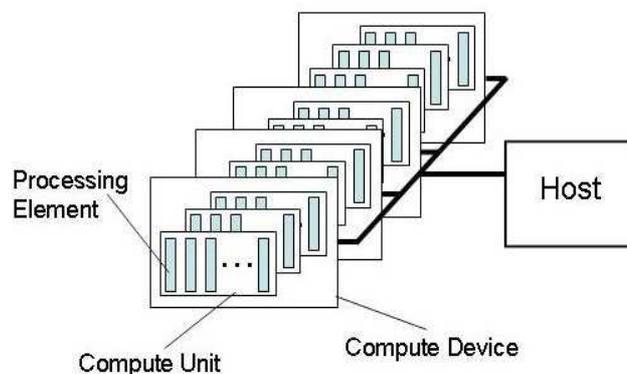


Figure 8. Platform Model of OpenCL

The program written in OpenCL runs on the host side submitting commands to execute computations on the processing elements within a device. The processing elements execute a single stream of instructions.

ii) Execution Model

OpenCL execution can be separated in two parts.

First a host program, which defines a *context* (environment where the kernels execute and the domain in which synchronization and memory management is defined). The context manages the execution of the kernels. Then the kernel functions are executed on one or more OpenCL devices.

When the host program submits a kernel for execution, an index space, called *NDRange*, is defined. For each point of this space an instance of the kernel is executed. This kernel instance is called in OpenCL *work-item*, which is identified by a global ID in the index space. Each work-item executes the same code but the execution pathway and the data operated can vary per work-item.

Moreover, work-items can be organized into *work-groups*, which provide a more coarse-grained decomposition of the index space. For each work-group a work-group ID is assigned and for each work-item within a work-group a local ID is assigned also. So in summary, each work-item has two IDs (global and local) and each work-group has one ID.

The NDRange can be one, two or three dimensional, so each work-item ID will be N-dimensional tuples, where N is the dimension of the NDRange. Figure 9 shows an example of 2-dimensional NDRange.

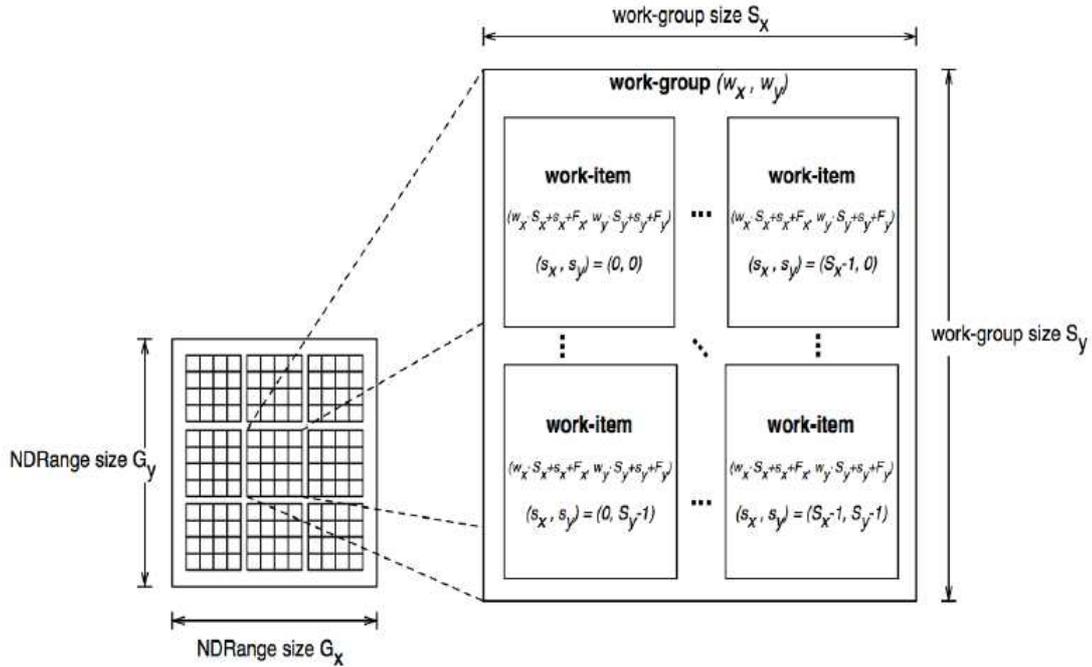


Figure 9. Example of 2-dimensional NDRange

In this example each little square is a work-item and each of the nine bigger squares is a work-group. The NDRange is used to identify and to have control of all the work-items that are executed. So for example, if we want to execute a program that does the matrix operation $A+B=C$, each work-item, when we launch the kernel a NDRange like the above will be created. Each work-item of the NDRange will perform a sum operation to calculate one element of the matrix C, so for example the work-item with the ID (0,0) will perform the operation $A[0][0]+B[0][0]=C[0][0]$. Because we have the control of all the work-items using the NDRange and the IDs, the sum operation inside the kernel function is just $A[x][y]+B[x][y]=C[x][y]$ where x and y are the IDs of each work-item.

As mentioned before, the host program creates a context to manage the executions of the kernel. The context includes the devices, the kernels, the program objects (source and executable that implement the kernels) and memory objects visible to the host and the device.

The method to submit orders from the host to the kernel is through a data structure called *command-queue*, created on the host side. The host program places commands into this structure which are then scheduled onto the device within the context.

Examples of commands are the one that executes a kernel, synchronization commands or memory commands which transfer data between memory objects (host->device, device->host or device->device).

iii) Memory Model

OpenCL divides the memory into four distinct regions:

- Global memory. Any work-item has read/write permission to this region.
- Constant memory. Is a region of the global memory that remains constant during the execution of a kernel. The host must allocate and initialize the memory objects placed in this region.
- Local memory. This region is local to a work-group, which means that is shared only by all the work-items in that work-group.
- Private memory. Memory private to a work-item. Any variable defined there is only accessible by this work-item.

Figure 10 shows these four regions of memory and how they relate to the platform model.

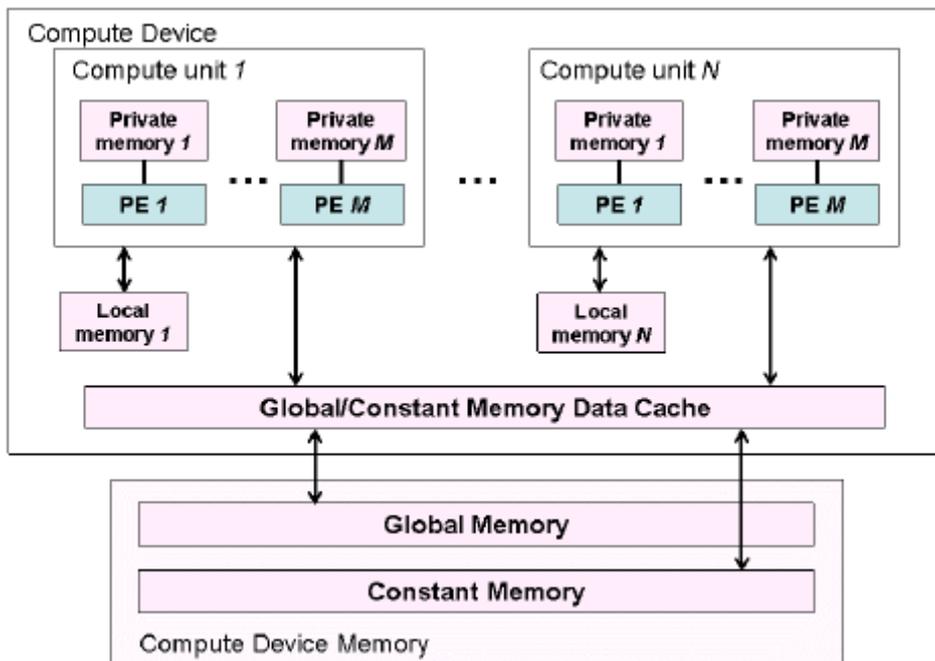


Figure 10. Conceptual OpenCL architecture

The host application uses the OpenCL API to create memory objects in global memory, and to enqueue memory commands, to write/read to/from these memory objects.

The host and the device memory are almost completely independent of each other, because the host side is defined outside of OpenCL, i.e. everything that is used by OpenCL during the execution (NDRange, memory regions ...) is inside the OpenCL device. The host can only communicate with the OpenCL device through the OpenCL API functions. However they need to interact in order to pass data between memory objects. This occurs in one of these two ways: by copying data or by mapping and unmapping regions of a memory object.

To copy data explicitly, the host has to submit commands to transfer data between a memory object and host memory, which can be blocking or non-blocking. On the other hand, to map/unmap regions of a memory object, the host can map a region from this memory object into its address space. Once a region from the memory object has been mapped, the host is allowed to write/read to this region. When the host finishes all the write/read accesses, the region is unmapped.

iv) Programming Model

Two types of programming models are supported by OpenCL: data parallel and task parallel models, as well as hybrids of these two.

In the data parallel programming model a sequence of instructions (kernel) is applied to multiple elements of a memory object. The index space associated with the OpenCL execution model defines the work-items and how the data maps onto the work-items. Strict one-to-one mapping between the work-item and the element in a memory object is not a requirement.

On the other hand, the task parallel programming model defines a model in which a single instance of a kernel is executed independent of any index space. It is equivalent to executing a kernel on a computing unit with an NDRange of 1 work-item and 1 work-group.

The programming language of OpenCL kernels is based on C99. Figure 11 shows an example of the vector addition problem written in this language.

```

__kernel void VectorAdd(__global const float* a, __global const float* b, __global
float* c, int iNumElements)
{
    // get index into global data array
    int iGID = get_global_id(0);

    // bound check (equivalent to the limit on a 'for' loop for standard/serial C
code
    if (iGID >= iNumElements)
    {
        return;
    }

    // add the vector elements
    c[iGID] = a[iGID] + b[iGID];
}

```

Figure 11. VectorAdd example in OpenCL kernel language

A kernel function must be defined with the **__kernel** directive. Memory objects created and initialized on the host side are passed to the kernel function as parameters. These parameters can have the directives **__global**, **__local**, etc. depending on the memory region that we want to use. In this example we have 3 arrays of floats (2 inputs and 1 output) declared as **__global** and a single int. The two inputs have also the keyword “const” so they are placed into the Constant Memory region.

Inside the kernel function the sum operation is executed for each element of the arrays. Before the operation, we must get the ID to identify the work-item inside the NDRange, stored in the variable **iGID**, so each work-item will perform one sum operation.

This code would be executed on the GPU side, so previously the host side must build and execute this code using the OpenCL API. The following code (Figure 12 and 13) shows the basic steps to prepare the OpenCL environment, execute the kernel function and retrieve the results from it for this VectorAdd example.

```

const unsigned int cnBlockSize = 512;
const unsigned int cnBlocks = 3;
const unsigned int cnDimension = cnBlocks * cnBlockSize;

// create OpenCL device & context
cl_context hContext;
hContext = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU,
0, 0, 0);

// query all devices available to the context
size_t nContextDescriptorSize;
clGetContextInfo(hContext, CL_CONTEXT_DEVICES,
0, 0, &nContextDescriptorSize);
cl_device_id * aDevices = malloc(nContextDescriptorSize);
clGetContextInfo(hContext, CL_CONTEXT_DEVICES,
nContextDescriptorSize, aDevices, 0);

// create a command queue for first device the context reported
cl_command_queue hCmdQueue;
hCmdQueue = clCreateCommandQueue(hContext, aDevices[0], 0, 0);

// create & compile program
cl_program hProgram;
hProgram = clCreateProgramWithSource(hContext, 1,
sProgramSource, 0, 0);
clBuildProgram(hProgram, 0, 0, 0, 0, 0);

// create kernel
cl_kernel hKernel;
hKernel = clCreateKernel(hProgram, "vectorAdd", 0);

// allocate host vectors
float * pA = new float[cnDimension];
float * pB = new float[cnDimension];
float * pC = new float[cnDimension];

// initialize host memory
randomInit(pA, cnDimension);
randomInit(pB, cnDimension);

// allocate device memory
cl_mem hDeviceMemA, hDeviceMemB, hDeviceMemC;
hDeviceMemA = clCreateBuffer(hContext,
CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
cnDimension * sizeof(cl_float),
pA,
0);
hDeviceMemB = clCreateBuffer(hContext,
CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
cnDimension * sizeof(cl_float),
pA,
0);
hDeviceMemC = clCreateBuffer(hContext,
CL_MEM_WRITE_ONLY,
cnDimension * sizeof(cl_float),
0, 0);

```

Figure 12. OpenCL host code VectorAdd example. Part one

```

// setup parameter values
clSetKernelArg(hKernel, 0, sizeof(cl_mem), (void
*)&hDeviceMemA);
clSetKernelArg(hKernel, 1, sizeof(cl_mem), (void
*)&hDeviceMemB);
clSetKernelArg(hKernel, 2, sizeof(cl_mem), (void
*)&hDeviceMemC);

// execute kernel
clEnqueueNDRangeKernel(hCmdQueue, hKernel, 1, 0,
&cnDimension, 0, 0, 0, 0);

// copy results from device back to host
clEnqueueReadBuffer(hContext, hDeviceMemC, CL_TRUE, 0,
cnDimension * sizeof(cl_float),
pC, 0, 0, 0);
delete[] pA;
delete[] pB;
delete[] pC;
clReleaseMemObj(hDeviceMemA);
clReleaseMemObj(hDeviceMemB);
clReleaseMemObj(hDeviceMemC);

```

Figure 13. OpenCL host code VectorAdd example. Part two

These API functions would be written on the host side to manage the execution of the kernel function.

Summarizing, OpenCL provides an API and a C-based kernel language that make an easy and transparent interface for GPGPU. It can be used in most modern GPU due to its platform-independent design. Therefore, the Caravela platform, which has potentially a stream-based computing style, can take advantage of OpenCL characteristics to become almost 100% compatible with any GPU.

In order to port the Caravela to OpenCL, we must take care of these issues:

- Relationship between the functionality of the Caravela functions and the OpenCL API functions.
- Modifications in the flow-model structure in order to pack all the OpenCL information needed.
- Modifications in the GUI “FlowModelCreator” in order to accept the new modifications done in the flow-model structure.

- Design of a header pattern that an OpenCL kernel function must follow to be accepted by Caravela Platform.
- Find a method in OpenCL for the implementation of the swap mechanism provided by Caravela platform.

The next chapter will explain all the process followed to achieve the previous objectives.

Chapter 3 Stream-based computing platform on multigenerational GPUs

After explaining all the background related to this project, in this chapter the process followed to migrate the Caravela to OpenCL is explained.

First of all we will describe the organization of the Caravela, i.e. the files which is composed of, the structure of the code, etc.

3.1 Caravela implementation

The Caravela application is mainly composed of 3 project files:

-“**CaravelaFlowModelClass**”. This project contains the definition of the “FlowModel” class, which is used in the rest of the files.

-“**CaravelaFlowModelCreator**”. This project contains the GUI application used to create a flow-model XML file, which is read by Caravela to fill the flow-model class.

-“**Caravela**”. This .DLL project contains the main code of Caravela (Caravela API functions and mid-level functions). When compiled it creates a dynamic library file with the Caravela API.

The structure of the code of the last project file, “Caravela”, is a 3-level stratum, showed in Figure 14.

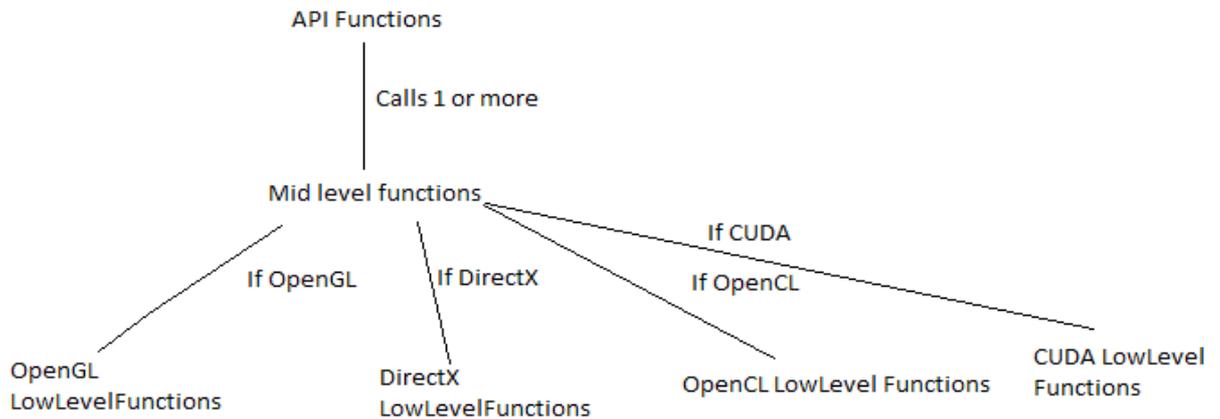


Figure 14. Structure of the Caravela functions

In the highest level we have the API functions of Caravela (named in Table 1). Each function can call 1 or more mid level functions. The functionality of these functions is basically to select the appropriate low-level function depending on the environment we are using. Finally the low-level functions have the functionality of each environment supported by Caravela. These functions are defined inside a .DLL project, one for each supported environment. When compiled, each project produces a DLL library with the functionality of the environment.

In summary, to give the OpenCL support to Caravela, we will only have to create a new .DLL project with the low-level functions programmed with the OpenCL API, as well as modify the mid-level functions just by adding a new condition for the OpenCL support. With this, the rest of the environments will not be affected by the porting process.

3.2 Caravela low-level functions

The next step is to analyze the functionality of the Caravela low-level functions in order to establish a relationship between these functions and the OpenCL functions. Next, a list of the low-level functions and a brief explanation of its functionality is provided:

1) `_InitializeLowlevel`

Initializes the environment and creates the structures that will be used in the rest of the functions.

2) _GetAllShaderInLocalMachine

This functions searches all the shader devices in the local machine and returns the first one available. Also, it obtains and returns technical information of the device in order to set maximum values such as “Maximum number of input streams allowed”, “Maximum data size”, etc.

3) _CompileShaderProgram

Read and compiles the shader/kernel program written by the user.

4) _AllocateOutputBufferToShader

This function creates one buffer for each output data stream. These buffers are stored in an array and the function returns a pointer to this array.

5) _MapProgramToShader

Links the shader program previously compiled to the current shader.

6) _SetConstants

Creates the necessary memory for the constant values and allocates these values into the buffers.

7) _GetInputBuffer

Creates and returns the buffers that will be used to store the input data.

8) _MapInputBufferToShader

Writes the input data initialized in the host into the buffers in the device side.

9) _FireShader

This function executes the program shader.

10) _GetOutputBuffer

Obtains the buffer that contains the output data after the execution.

11) `_FreeOutputBufferFromShader`

Free the memory used for the output data.

12) `_UnmapInputBufferFromShader`

Free the memory used for the input data.

13) `_FinalizeLowlevel`

Finalizes the environment and free all the structures and memory used in the whole process.

The previous functions are listed in the execution order of a program that uses the Caravela API.

Now that we know the functionality of each low-level function, we can decide the OpenCL code for each function. The following list explains the implementation of each function and the OpenCL API functions used in each one.

1) `_InitializeLowLevel`

Although OpenCL does not need an explicit initialization, this function will get the OpenCL platform with `clGetPlatformIDs` API function. This function was chosen to be placed here because if there is no platform for OpenCL there is no reason to continue the execution. Also this function will create a structure called `__GPGPUOCL_Device_Info` in which all the information needed by OpenCL (context, kernel, buffer pointers, etc.) will be stored.

2) `_GetAllShaderInLocalMachine`

This function will use the OpenCL API function `clGetDeviceIDs` to get an available device. Also to get the information of this device, the function `clGetDeviceInfo` will be used too.

3) `_CompileShaderProgram`

Here we must create and build the kernel program, so previously we must create a context to manage the program. Therefore we will use `clCreateContext`, `clCreateProgramWithSource`, `clBuildProgram` and `clCreateKernel`. Also, although is not used in this function, a command queue with `clCreateCommandQueue` is created after the context creation. This is placed here because this function is always executed only once, but the next functions may be executed more than one time, due to the possibility of using the swap mechanism.

4) `_AllocateOutputBufferToShader`

This function must create the output buffers so the function `clCreateBuffer` is used one time for each output stream. These buffers are stored in an array of “number of output streams” length. Because these buffers are in the device side, we must create also buffers for the host side, so that the data can go back from the device to the host. The reason why these host-side buffers are created is explained in point 7) of this list. These host buffers will be created using dynamic memory (`malloc`).

5) `_MapProgramToShader`

OpenCL does not need to explicitly link the program to the device because this is already managed by the context. Therefore, there is nothing to do in this function.

6) `_SetConstants`

In this function the buffers for the constant values are created with `clCreateBuffer`. These buffers are also filled up with the values of the constants.

7) `_GetInputBuffer`

This function does almost the same as the `_AllocateOutputBufferToShader` but with the input buffers. So we create one buffer per input stream with `clCreateBuffer` for the device side, and one buffer per input stream with dynamic memory for the host side. The host side buffers are needed because Calavera initializes the input buffers at the API functions

level (the top), so if we return directly an OpenCL buffer, this buffer cannot be initialize without using the OpenCL API function `clEnqueueWriteBuffer`, and if we force the user to use this function, the way of programming with Calavera-OpenCL would drastically change compared to the previous versions of Caravela. Therefore, the creation of host side buffers is the best solution to preserve the way of programming with Caravela.

The array of buffers for the host side is returned in order to initialize them in the main program.

8) `_MapInputBufferToShader`

Here we should transfer the input data from the host buffers to the device buffers. Therefore we use the function `clEnqueueWriteBuffer` for each input stream to execute the copy operation.

9) `_FireShader`

In this function we should prepare the kernel arguments for the execution and run this kernel. Thus, we use the `clSetKernelArg` function to assign the input buffers, the output buffers and the constant buffers to the kernel. Then, the function `clEnqueueNDRangeKernel` is used to execute the kernel. Because the parameters of the `clEnqueueNDRangeKernel` function change depending on the NDRange dimension selected by the user (for example, if the dimension is one the function expects just an int in the `GlobalWorkSize` and `LocalWorkSize` parameters but if the dimension is more than one the function expects an array in those parameters), we have implemented in the code three different calls to the function (one for each dimension) controlled by a conditional operator.

10) `_GetOutputBuffer`

In this function we must return the host side output buffer with the output data after the execution. But before we must transfer the data between the buffers (device to host) so we use the function `clEnqueueReadBuffer`.

11) `_FreeOutputBufferFromShader`

This function frees both the device buffers and the host buffers. Therefore for the device buffers we use the function `clReleaseMemObject`. We must free also the arrays that contain the buffers.

12) `_UnmapInputBufferFromShader`

The same as the previous function but for the input buffers (device and host).

13) `_FinalizeLowlevel`

Here we release all the OpenCL resources used in the process like the kernel, program, command queue and the context. So the functions `clReleaseKernel`, `clReleaseProgram`, `clReleaseCommandQueue` and `clReleaseContext` are used.

All these low-level functions will be programmed in a new .DLL project called “CaravelaOCL”, and will be used by the Caravela API functions.

Next we will explain the FlowModel structure used by Caravela and the changes in this structure required for the OpenCL version.

3.3 Flow Model

The flow-model is a structure that packs all the information used by the Caravela in a XML file to execute the program. Using this structure, the programmer only has to worry about writing a flow-model and mapping it to the Caravela environment using the Caravela API functions. Then the Caravela uses the information of this flow-model to perform the execution of the kernel.

The values required by the flow-model are:

- i) **NumData**. The number of data that will be processed. For example, if we want to add 2 vectors of 1024 elements, NumData will be 1024.
- ii) **DataType**. The type of the values of the streams (FLOAT, INT, SHORT).
- iii) **NumInput**. Number of input streams.
- iv) **NumOutput**. Number of output streams.

v) **ShaderProgram**. Here we write the kernel program that will be executed on the GPU.

vi) **FunctionName**. The name of the function of the kernel program.

vii) **LangType**. Language used to write the shader program.

viii) **RuntimeType**. Runtime that we want to use to execute the program.

ix) **ShaderVersion**. Version of the shader language (not used in OpenCL).

x) **ConstValues**. List of the constant values used in the kernel. This list has a special format that will be explained in the next section.

xi) **ConstTypes**. List of types (FLOAT, INT, SHORT) of the constant values.

xii) **ConstNames**. List of names of the constant values.

xiii) **NumConstant**. Number of constant values.

In addition to the previous parameters, OpenCL needs the user to introduce 3 more values to execute the kernel. These are:

xiv) **Dimension**. The dimension (1, 2, 3) of the NDRange.

xv) **Threads**. This list of 3 elements is the GlobalWorkSize in each dimension.

xvi) **Blocks**. This list of 3 elements is the LocalWorkSize in each dimension.

The Figure 15 shows an example of an XML flow-model.

```

<?xml version="1.0"?>
<FlowModelInfo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <NumData>8</NumData>
  <DataType>FLOAT</DataType>
  <NumInput>2</NumInput>
  <NumOutput>1</NumOutput>
  <ShaderProgram>
    __kernel void VectorAdd(__global const float* input1, __global const float*
input2, __global const float* const1, __global float* output){
      int iGID = get_global_id(0);
      if (iGID &gt;= const1[0])
      {
        return ;
      }
      output[iGID] = input1[iGID] + input2[iGID];

    }
  </ShaderProgram>
  <FunctionName>VectorAdd</FunctionName>
  <LangType>SHADERLANG_OPENCL1.0</LangType>
  <RuntimeType>RUNTIME_OPENCL</RuntimeType>
  <ShaderVersion>0</ShaderVersion>
  <ConstValues>1, 256, 0, 0, 0</ConstValues>
  <ConstTypes>FLOAT4</ConstTypes>
  <ConstNames>const_1</ConstNames>
  <NumConstant>1</NumConstant>
  <Dimension>1</Dimension>
  <Threads>256, 1, 1</Threads>
  <Blocks>256, 1, 1</Blocks>
</FlowModelInfo>

```

Figure 15. Flow Model of VectorAdd example

Next we will explain the pattern designed for the constant values list.

3.3.1 Constant values structure

In the previous version of the Caravela, each constant value had the structure of texture data, i.e. 4 data per pixel (Red, Green, Blue, and Alpha) because it used the graphics runtimes OpenGL and DirectX. However OpenCL does not need that structure, so a new one has been designed.

The new structure is described as follows: a list of arrays in which each array is a list of constant values of the same type. Each array is considered one constant value, so if

we have 3 in the flow-model `NumConstant` parameter it means that we have 3 arrays of constant values. Each array can have any number of constant values.

The `ConstValues` list follows this pattern: (X, x1,...x(4*X) , Y, y1,...y(4*Y), Z, z1,...z(4*Z)). The first number of each array (X, Y, Z) sets the number of constant values of the array multiplied per 4. Then, the values of the constant values of each array are written. So, for example in this list (1, 256, 0, 0, 0), we have 1 array of constant values with only one value “256”. The rest three “0” appear in the list because the FlowModelCreator reads a whole row of four values, due to the interface design, so it may be extra non-used values, like in this case. The user can access inside the kernel to every constant value in the arrays.

However, the user has not to worry about these implementation details because with the GUI Flow Model Creator, explained in the next section, these issues are hidden.

3.4 Flow Model Creator

The FlowModelCreator is a graphical user interface (GUI), programmed in C#, that allows the user to easily create a XML flow-model file. A more detailed explanation of this interface is provided in the appendix A, so in this section we will only explain the changes performed to the GUI in order to create a flow-model for OpenCL. The Figure 16 shows the top window with the new features highlighted.

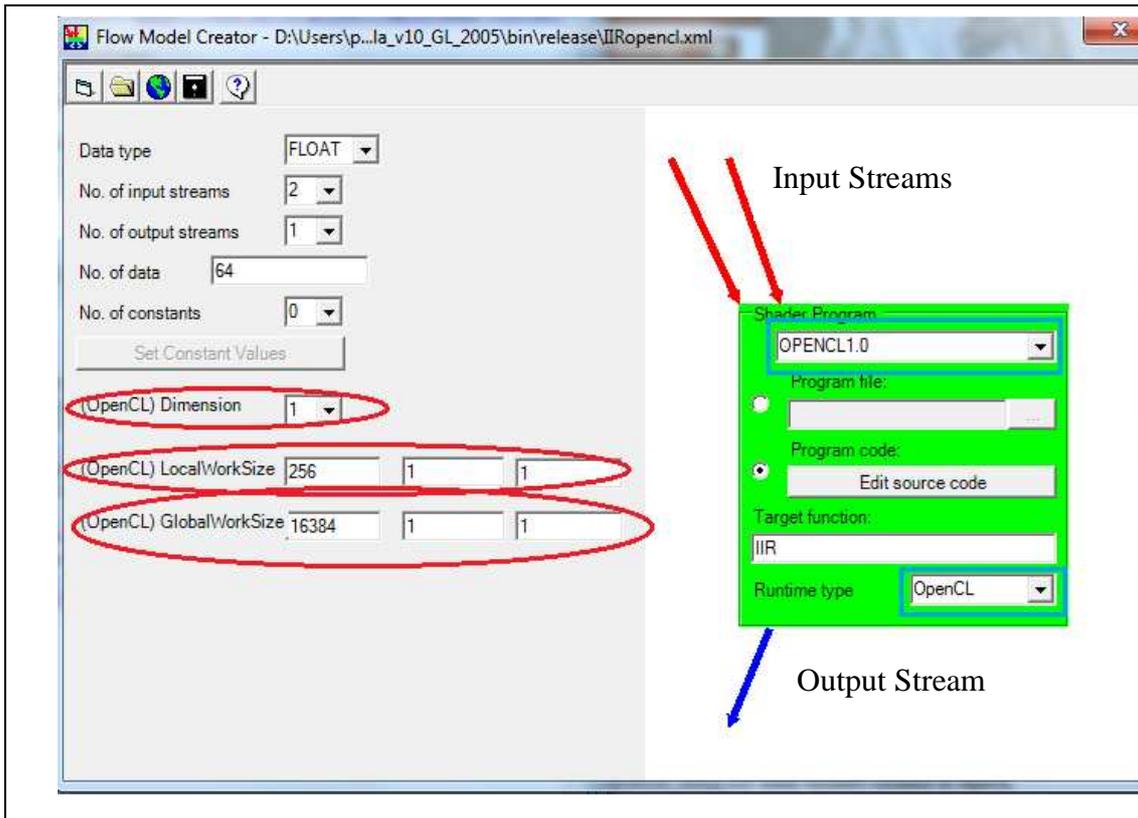


Figure 16. Changes in the top window of FlowModelCreator

First of all we modified the combo boxes “Shader Program” and “Runtime type” in order to add a new item for OpenCL. Then new methods for introducing the new OpenCL parameters of the flow-model (dimension, local work size and global work size) were added in the left side of the screen. Finally, a completely new window for introducing the constant values has been developed, as shown in Figure 17.

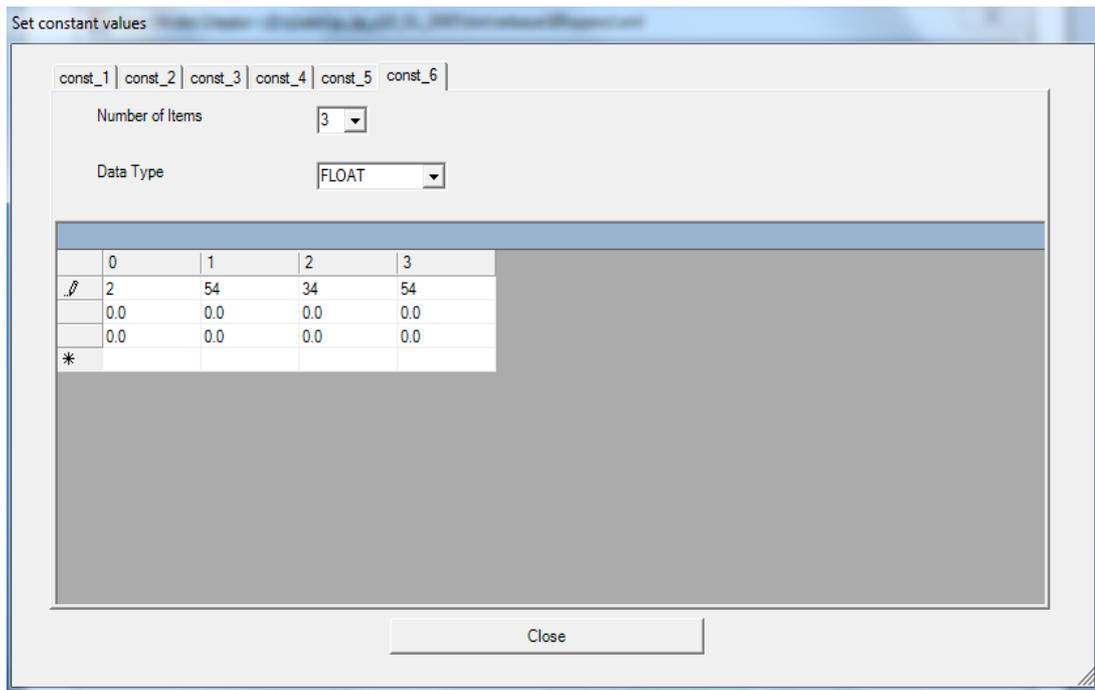


Figure 17. New interface for introducing the constant values

This multi-tab window dynamically generates one tab for each constant array. The name of each array is automatically generated. Then, inside each tab we can introduce the type and values of the constants of the array. After closing the window the changes performed remain until we close the FlowModelCreator.

3.5 Kernel Structure

In this section we will explain the structure of the kernel header that must be followed in order to be accepted by the Caravela.

As stated before, a flow-model has 2 kinds of inputs (input stream and constant values) and 1 kind of output (output stream). There may be more than one of each kind of these. Therefore the kernel parameters must match this structure of the flow-model. In the CUDA version of the Caravela the solution was to make each parameter an array of arrays like this: function (**input, **constant, **output). With this pattern, each array contains all the streams of one kind (input, output and constants). However this is not possible in OpenCL because the buffers are 1-dimensional, so the parameters in an OpenCL kernel cannot be an array of arrays, i.e. 2-dimensional. Therefore the solution developed is to follow this pattern: function (*input1, *input2 ..., *inputN, *const1, *const2 ..., *constN, *output1, *output2 ..., *outputN).

This means that the user has to write the kernel header as follows: first all the input streams, then the constant values arrays and finally the output streams. There must be at least one input stream and one output stream, but the constant value parameter is optional. If this order is not strictly followed the Caravela will not work properly and the results may be unexpected. Also if the number of kernel parameters does not match with the sum of the flow-model parameters “NumInput”, “NumOutput” and “NumConstant”, the Caravela will return an error warning this issue.

Figure 18 shows a correct example of kernel header for the Caravela.

```
__kernel void VectorAdd(__global const float* input1,
__global const float* input2,
__global const float* const1,
__global float* output)
```

Figure 18. Example of kernel header for Caravela

3.6 Swap Mechanism

As mentioned in section 2.3.1, the swap mechanism of the Caravela allows the user to change (swap) the output and the input streams of the flow-model in order to perform repeated executions of a kernel without copying the results from the device to the host side. Therefore the data remains on the device side until it is required by the CPU.

In OpenCL, the implementation of this mechanism has been realized using the `clSetKernelArg` function. First we defined a new low-level function called `_SwapFlowmodelIOOCL` that swaps the pointers of the device-side buffers of a previously defined I/O pair with the Caravela API function `CARAVELA_CreateSwapIoPair`. In order to know which input buffers have been swapped to not make a data transfer on them, we use a global array of bools called “swapped”. So for instance, “swapped [2] == true”, means that the buffer with the index 2 has been swapped. The values of this array are changed in the `_SwapFlowmodelIOOCL`, and are reinitialized after each execution. The `_SwapFlowmodelIOOCL` function also makes “TRUE” a global flag called “swap” in order to warn the Caravela that we are using the swap mechanism so the input buffers do not have to be rewritten during the `_MapInputBufferToShader` function.

Then during the new fire operation, the swapped buffers are linked again to the kernel using the `clSetKernelArg`, so the data is never copied back to the host side.

The implementation of this mechanism allows the user to rewrite with new data the input buffers that are not swapped. As mentioned before, if we use the swap mechanism the `_MapInputBufferToShader` function does not rewrite the input buffers, but the user can change this using the `GetInputData` API function. If we use this function before the execution, the input buffer that we retrieve will be rewritten in the `_MapInputBufferToShader` function even if we are using the swap mechanism. Notice that a swapped buffer will not be rewritten never even if we retrieve it with the `GetInputData` function. We have implemented this mechanism using a global array of bools called “rewrite”. If `rewrite[i] == “true”` the input buffer with the index `i` will be rewritten. The values of the array are changed in the `_GetInputBuffer` function and are reinitialized after each execution.

Because we swap also the host side buffers during the `_SwapFlowmodelIOCL` function, the user does not have to worry about which buffer contains the real output data after the last execution, so he can retrieve the output data normally using the Caravela API function `GetOutputData`.

Therefore with this method we can repeatedly execute a kernel without losing time copying the data to the host side after one execution and transfer it again to the device side before the next execution.

3.7 Implementation conclusions

According to the implementation showed in this chapter, the flow-model for the OpenCL presents the same structure and behavior implemented on the previous versions of the Caravela. Also the GUI FlowModelCreator keeps the compatibility between different versions of the Caravela, and the user just has to select the runtime and the language in which the kernel program is written. Therefore, just creating a flow-model, the execution framework of the Caravela supports the runtimes with the legacy architecture of the GPUs and the runtimes with the recent architecture. Moreover, since OpenCL is a platform-independent API, the Caravela can be executed in any new GPU independently of the manufacturer.

In the next chapter we will discuss the performance aspect on this implementation.

Chapter 4 Performance Evaluation

In this section we will evaluate the performance between the new Caravela version using OpenCL and the legacy implementation of the Caravela using OpenGL.

Two different applications are evaluated: one is a straight forward execution flow-model without iteration, using a matrix-multiply program; another is an infinite impulse response (IIR) filter that iterates recursively the flow-model.

We have measured the time to execute an OpenCL program with and without using the Caravela platform. Also two version of each OpenCL program have been measured: one using only global memory and another using shared memory. Finally we have measured the execution time of the Caravela OpenGL version to compare the performance of recent GPU architecture to the legacy one. The environment for the evaluation is listed in the Table 2.

Table 2. Environment for the evaluation

CPU	Intel® Core™ 2 Duo E7500 @ 2.93GHz
GPU	GeForce Gt 220 (Core: 625GHz / DDR3 VRAM: 1GB)
OS	Windows 7 Professional
OpenCL	Version 1.0

4.1 Straightforward application: Matrix multiply

The matrix multiplication program processes $A * B$ of $N \times N$ matrices straightforward, i.e. does not have any iteration or recursive I/O. Figure 19 shows the execution times of this program varying the matrix size N from 512 to 1024.

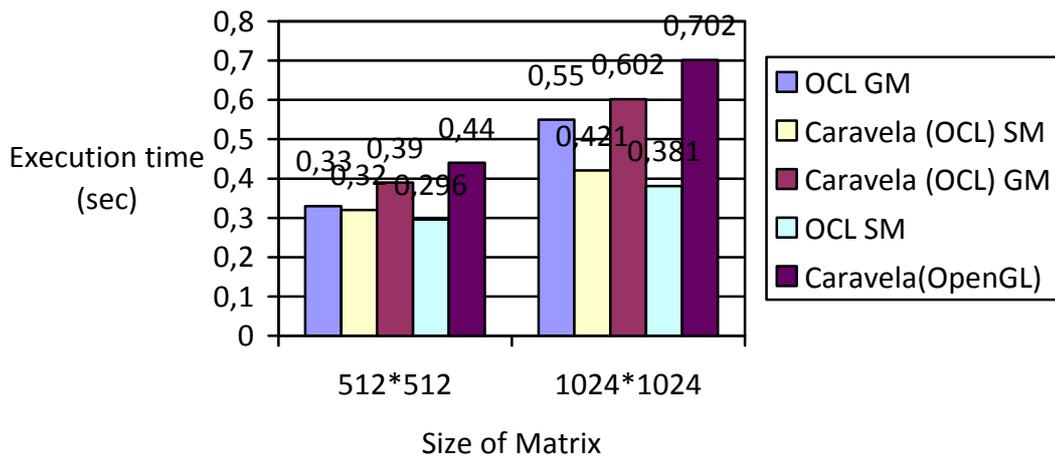


Figure 19. Execution times using matrix multiply

We are measuring five different versions of the matrix multiply program:

- i) OCL GM: OpenCL version using only global memory.
- ii) OCL SM: OpenCL version using shared memory.
- iii) Caravela (OCL) GM: Caravela-OpenCL version using only global memory.
- iv) Caravela (OCL) SM: Caravela-OpenCL version using shared memory.
- v) Caravela (OpenGL): Caravela-OpenGL version.

All OpenCL versions show better performance than the OpenGL version over the Caravela because the flow-model execution mechanism of the OpenGL version potentially includes redundant processes following the graphics processing method in the legacy architecture. When OpenCL uses the global memory (GM) the execution times become 1.12-1.45 times longer than the ones with shared memory (SM). Let's take a look at the kernels to see the differences of using shared memory or not. The left code of Figure 20 is the GM version and the right code is the SM version.

<pre> __kernel void matrixMultOCL(__global float* a, __global float* b, __global float* c) { int TileDIM = 1024; int row = get_global_id(1); int col = get_global_id(0); float sum = 0.0f; for (int i = 0; i < TileDIM; i++){ sum += a[row*TileDIM+i] * b[i*TileDIM+col]; } c[row*TileDIM+col] = sum; } </pre>	<pre> #define AS(i, j) As[j + i * 1024] #define BS(i, j) Bs[j + i * 1024] __kernel void matrixMultOCL(__global float* a, __global float* b, __global float* c) { int TileDIM = 1024; __local float As[TileDIM* TileDIM]; __local float Bs[TileDIM]* TileDIM]; int row = get_global_id(1); int col = get_global_id(0); int x = get_local_id(0); int y = get_local_id(1); float sum = 0.0f; AS(y, x) = a[row* TileDIM +x]; BS(y, x) = b[y* TileDIM +col]; barrier(CLK_LOCAL_MEM_FENCE); for (int i = 0; i < TileDIM; i++) { sum += AS(y, i) * BS(i, x); } c[row* TileDIM +col] = sum; } </pre>
---	--

Figure 20 Matrix Multiply Kernels

In this program ($A*B = C$), each work-item reads one row of the matrix A, one column of the matrix B and calculates one element of the result matrix. If we use only global memory, we read the same data from global memory more than once, decreasing the execution time. For example, the first row of the matrix A is read by all the work items that calculate the first row of the matrix C and the first column of the matrix B is read by all the items that calculate the first column of matrix C. This happens with every row and column.

To avoid this, we use shared memory as shown in the right code of Figure 19. We create two arrays of shared memory As and Bs. In these arrays we store a whole row and a whole column of the matrix A and B respectively, reading only once from the global memory. This shared memory is shared by all the work-items within a work-group so these work-items will read the values from shared memory, which is faster than reading from global memory. To synchronize the work-items we put a barrier

(barrier (CLK_LOCAL_MEM_FENCE)) to stop the execution of all the work-items until the shared memory is completely created.

Thus, to achieve a maximum performance, the OpenCL version needs to use the shared memory, otherwise the performance would be close to the OpenGL version, as shown in the 1024*1024 sample.

Also, we have measured the matrix multiply problem (1024*1024) using the CPU to see the power of a GPU compared to a CPU. The CPU needs 20.765 seconds to calculate the result, very far from the times achieved by using the GPU.

Comparing the versions with/without Caravela runtime, the OpenCL versions with Caravela are slower than the versions without it due to these reasons.

- 1) The Caravela over OpenCL needs to access the driver level or the OpenCL runtime API via several dynamic linked libraries that causes calling overhead to load nested functions.
- 2) Also the Caravela needs to load and analyze the flow-model including in a XML file.

However this overhead caused by the Caravela is fixed, so it is independent of the data size.

Therefore the Caravela over OpenCL keeps better performances than Caravela over the legacy GPU architectures with OpenGL. Moreover, the overhead of the Caravela runtime itself is not significant and is independent of the data size.

4.2 Recursive application: IIR filter

This evaluation analyzes the performance compatibility of the swap mechanism using a recursive program as the Infinite Impulse Response (IIR), in which the output is calculated using its previous output. In Figure 21 the execution times of this program varying the input data size from 16K to 64K are shown.

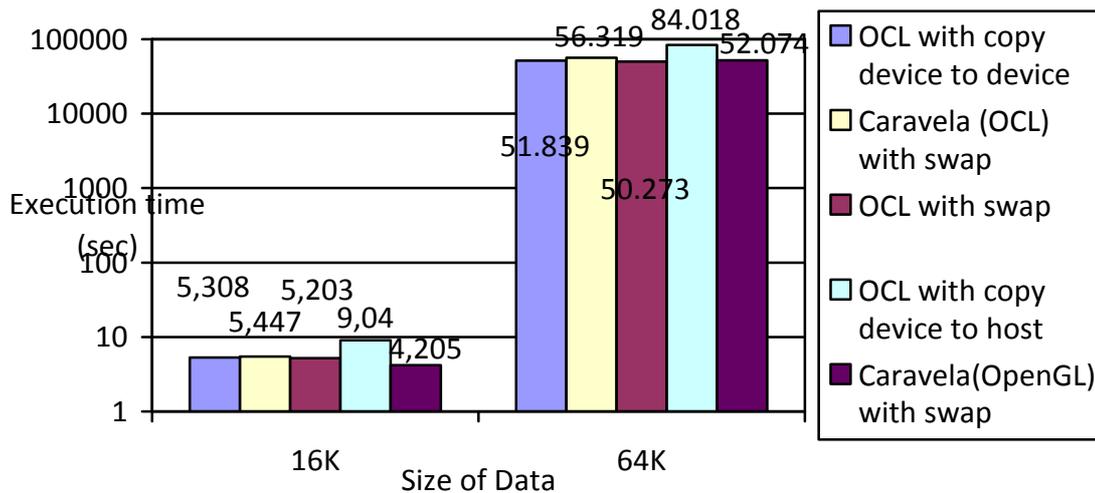


Figure 21 Execution times using IIR filter

We are measuring five different versions of the IIR filter program:

- i) OCL with copy device to device: OpenCL version using an OpenCL API function to transfer the data from the output buffer to the input buffer in the device side.
- ii) OCL with swap: OpenCL version using the swap mechanism used in Calavera i.e. using the SetKernelArg function to set the output buffer as an input parameter.
- iii) Caravela (OCL) with swap: Caravela-OpenCL version using the swap mechanism.
- iv) OCL with copy device to host: OpenCL version using the OpenCL API functions to read and write the buffers to transfer the data from the device to the host and vice versa after each iteration.
- v) Caravela (OpenGL): Caravela-OpenGL version with the swap mechanism.

Figure 22 shows the kernel code of this program.

```
__kernel void IIR(__global float *x1, __global float *x2, __global float *y){
    int bx = get_group_id(0);

    int idx = get_global_id(0);
    __local float a[8 + 1];
    __local float b[8 + 1];
    int i;
    float res;
    if (idx < 8)
    {
        a[idx] = ((float)((idx * 191) % 101) / 100 - 0.5) / 2;
        b[idx] = ((float)((idx * 191) % 101) / 100 - 0.5) / 2;
    }
    barrier(CLK_LOCAL_MEM_FENCE);
    res = 0;
    for(i = 0; i < 8; i++)
    {
        if (idx - i >= 0){
            res += b[i] * x1[idx - i];
        }
        if (idx - (i + 1) >= 0)
        {
            res += a[i] * x2[idx - (i + 1)];
        }
    }
    y[idx] = res;
    return;
}
```

Figure 22 Kernel code of IIR program

As mentioned before, with this program we are measuring the performance of the swap mechanism implemented in the OpenCL version of Calavera. We test the program with two different sizes (16K and 64K) of the input data. The number of iterations (swaps) is the same as the input data (16K times and 64K times).

The data presented in Figure 20 shows, as expected, that the execution times of the versions that use this mechanism are faster than the version that makes copy operations between host side and device side. This improvement is greater as the data size increases, because the bandwidth of the bus that connects the host to the GPU device is lower than the bus inside the GPU device.

However when comparing the two OpenCL versions that do not transfer data back to the host side, we see that the execution time is almost the same in both versions. In the “OCL with copy device to device” version we use the OpenCL API function `clEnqueueCopyBuffer` to copy the data from the output buffer to the input buffer after each execution of the kernel. On the other hand, in the “OCL with swap” version we do not use this function because we just change the kernel arguments before the executions. For example, if we have a kernel with two arguments “kernel void test(float *input, float *output)” and two read/write buffers “src” and “dst”, in the first execution we use the `clSetKernelArg` function to set the buffer “src”, initialized with the input data, to the argument 0 (the input) and the buffer “dst” to the argument 1 (the output). Then before the second execution we change the order and we set the buffer “dst” to the argument 0 and the buffer “src” to the argument 1. This way the data stored in “dst” from the first execution is the input data in the second execution without explicitly perform a copy operation between buffers.

Because the execution time of these versions is very similar, the function `clSetKernelArg` might implicitly perform a copy operation using the device bus when setting the buffers to the kernel. Due to this, the swap mechanism implemented in the Calavera, which also uses the `clSetKernelArg` function to make the swap, may not be a “true” swap between buffer pointers without any kind of copy operation.

To clarify this issue, we have performed an extra evaluation. We have measured the execution times of a simple VectorAdd ($a [] + b [] = c []$) program, swapping the output and the input buffer using the previously mentioned swap methods. The results obtained are showed in figure 23 and 24.

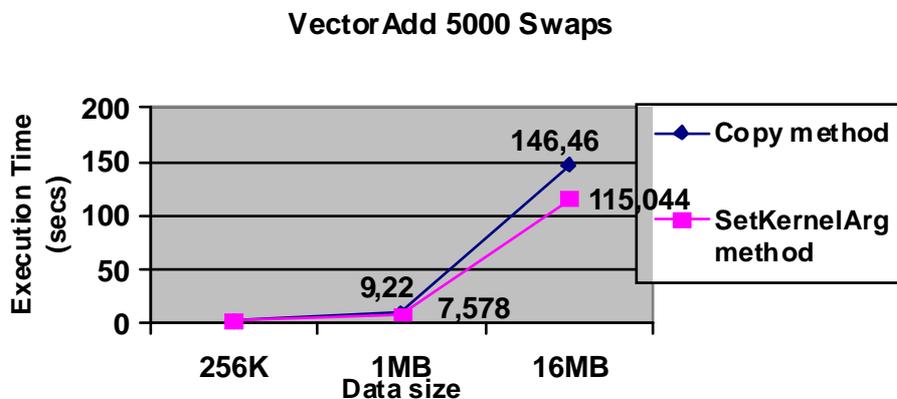


Figure 23. VectorAdd execution times with 5000 swaps

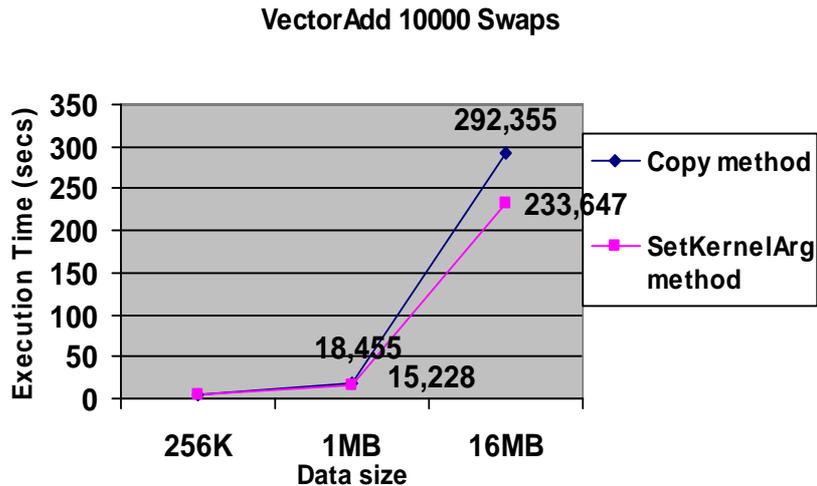


Figure 24. VectorAdd execution times with 10000 swaps

As we see in the above graphs, there is difference in the execution time between these two methods. Using the data of the figure 24, we see that using 16 MB of data size there is a difference of approximately 60 seconds between the methods. Supposing that the cost of SetKernelArg function is 0, a single transference of 16MB of data between buffers on the device side takes approximately $60/10000 = 0,006$ seconds. This time still seems very low, but it can be explained if we see our GPU bandwidth, which is approximately 25 GB/s. As a conclusion, difference between the execution times of these methods is small because the power of the hardware of the GPUs nowadays.

Finally, as happened in the matrix multiplication evaluation too, the Caravela-OpenCL version is slower than the OpenCL without Caravela version. When we increase the number of swaps, due to the fact that the Calavera needs to access the driver level via several dynamic linked libraries so for each execution we have to access the low-level functions to get the input buffers and fire the kernel, the execution time is affected when compared to the OpenCL without Calavera versions.

Therefore, the swap mechanism implemented maintains the performance improvement over methods that make transferences $\text{host} \leftrightarrow \text{device}$ each iteration. Also we have demonstrated that the mechanism implemented does not perform any data transference between buffers, even in the device side.

Chapter 5 Conclusions and future work

In this project we have presented the migration process of a stream-based computing platform, Caravela, to the OpenCL language. As presented in chapter 1, the main objective of this project was to allow the Caravela to take advantage of the new GPU architecture and the new runtimes for GPGPU, without losing performance speed and the compatibility with the legacy runtimes (OpenGL, DirectX). Also another goal of this project was to maintain the structure and the execution style of the flow-model in the new version of the Caravela, as well as all the good features of the Caravela like the swap mechanism.

According to the implementation presented in chapter 3, we have succeeded in preserving the flow-model structure and the features of the old version of the Caravela. In terms of performance evaluation, according to the results presented in chapter 4, the comparison using a straightforward application between the pure OpenCL version and the Caravela over OpenCL shows that the performance is only affected by fix degradation due to the process of reading and loading of the flow-model file. Also the swap mechanism, which prevents transferences between host memory and device memory when we execute iteratively a kernel, has been successfully implemented. Therefore we can conclude that the goals proposed in chapter 1 have been achieved.

The Caravela has been proven to be a true stream-based computing platform for GPGPU that is able to take advantage of the most recent GPGPU APIs. Therefore it is a good idea to continue with the improvement of this software, such as the followings features:

1. Increase the compatibility of the Caravela porting it to the language *DirectCompute*. This new environment is part of the Microsoft DirectX collection of APIs and runs in both DirectX 10 and DirectX 11. With the migration of the Calavera to this language, will increase its compatibility with systems with the Windows OS.
2. Meta-pipeline. This mechanism allows a flow-model to be virtually connected to any other flow-model(s). It implements a virtual structure with multiple flow-models, which can be used, for instance, to solve large problems through a set of flow-models executed in different processing units.

3. Development of a hardware compiler for the flow-model execution method. This project will aim to develop an environment that generates hardware description from stream-based program. Because the hardware description can be implemented by pipelined components, the hardware design productivity can become better than the naturally implemented application by hand.

Bibliography (references)

[1] At least nine contributions have emerged in recent years that explore the implications HPC presents for the future of humanities research. See “Links about Digital Humanities and HPC.” Available on-line at https://www.sharcnet.ca/dh-hpc/index.php/Links_about_Digital_Humanities_and_HPC [June 29, 2008].

[2] Gustafson, J.; "The program of grand challenge problems: expectations and results," *Parallel Algorithms/Architecture Synthesis, 1997. Proceedings. Second Aizu International Symposium*, vol., no., pp.2-7, 17-21 Mar 1997 doi: 10.1109/AISPAS.1997.581619

[3] The Jaguar Supercomputer.

<http://www.nccs.gov/jaguar/>.

[4] J. Gummaraju and M. Rosenblum. Stream programming on general-purpose processors. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 343–354, 2005.

[5] GPGPU research projects of ICHEC

http://www.ichec.ie/research/gpgpu_projects

[6] Kapre, N.; DeHon, A.; "Performance comparison of single-precision SPICE Model-Evaluation on FPGA, GPU, Cell, and multi-core processors," *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, vol., no., pp.65-72, Aug. 31 2009-Sept. 2 2009 doi: 10.1109/FPL.2009.5272548

[7] DirectX homepage. <http://www.microsoft.com/directx>.

[8] OpenGL Architecture Review Board, D. Shreiner, M. Woo, J. Neider, and T. Davis. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2*. Addison Wesley, 2005.

- [9] I. Buck, T. Foley, D. Horn, J. Sugerma, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.
- [10] Sh: A high-level metaprogramming language for modern GPUs. <http://libsh.org/>.
- [11] NVIDIA CUDA Zone. [Http://www.nvidia.com/cuda](http://www.nvidia.com/cuda).
- [12] OpenCL. [Http://www.khronos.org/ocl/](http://www.khronos.org/ocl/).
- [13] S.Yamagiwa, L.Sousa. Design and Implementation of a Stream-Based Distributed Computing Platform using Graphics Processing Units. In *ACM International Conference on Computing Frontiers*, May 2007.
- [14] Yamagiwa, S.; Sousa, L.; "Caravela: A Novel Stream-Based Distributed Computing Environment," *Computer*, vol.40, no.5, pp.70-77, May 2007 doi: 10.1109/MC.2007.161
- [15] Schatz, M.C., Trapnell, C., Delcher, A.L., Varshney, A. (2007) [High-throughput sequence alignment using Graphics Processing Units](#). *BMC Bioinformatics* 8:474.
- [16] S. Yamagiwa, L. Sousa, K. Ferreira, K. Aoki, M. Ono, and K. Wada. Maestro2: Experimental evaluation of communication performance improvement techniques in the link layer. *Journal of Interconnection Networks*, 7(2):295-318, 2006.
- [17] GPGPU Homepage. [Http://www.gpgpu.org/](http://www.gpgpu.org/)
- [18] P. S. McCormick, J. Inman, J. P. Ahrens, C. Hansen, and G. Roth. Scout: A hardware-Accelerated System for Quantitatively Driven Visualization and Analysis. In *VIS '04: Proceedings of the conference on Visualization '04*, pages 171-178, Washington, DC, Usa, 2004. IEEE Computer Society

[19] Shinichi Yamagiwa, Leonel Sousa, Diogo Antao, "Data Buffering optimization methods toward a uniform programming interface for GPU-based applications", *Proceeding of ACM Intl' Conference on Computing Frontiers*, pp. 205 - 212, May 2007.

[20] The Khronos Group. [Http://www.khronos.org/](http://www.khronos.org/).

[21] Farrell, C.A.; Kieronska, D.H.; "Formal specification of SIMD execution," *Algorithms and Architectures for Parallel Processing, 1996. ICAPP '96. 1996 IEEE Second International Conference on*, vol., no., pp.319-325, 11-13 Jun 1996
doi: 10.1109/ICAPP.1996.562891

[22] DeBenedictis, E.P.; "Will Moore's Law Be Sufficient," *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*, vol., no., pp. 45- 45, 06-12 Nov. 2004
doi: 10.1109/SC.2004.68

[23] <http://en.wikipedia.org/wiki/DirectCompute>

[24] Thinking Machines Corporation. C* User's Guide, June 1991.

Appendix I User guide of Flow Model Creator GUI.

1 Introduction

1.1 FlowModelCreator

The FlowModelCreator is an interface designed for the Caravela platform to help the user to design a flow-model structure. These flow-models can be saved and loaded to/from a XML file. In this guide we will explain the functionality of this application

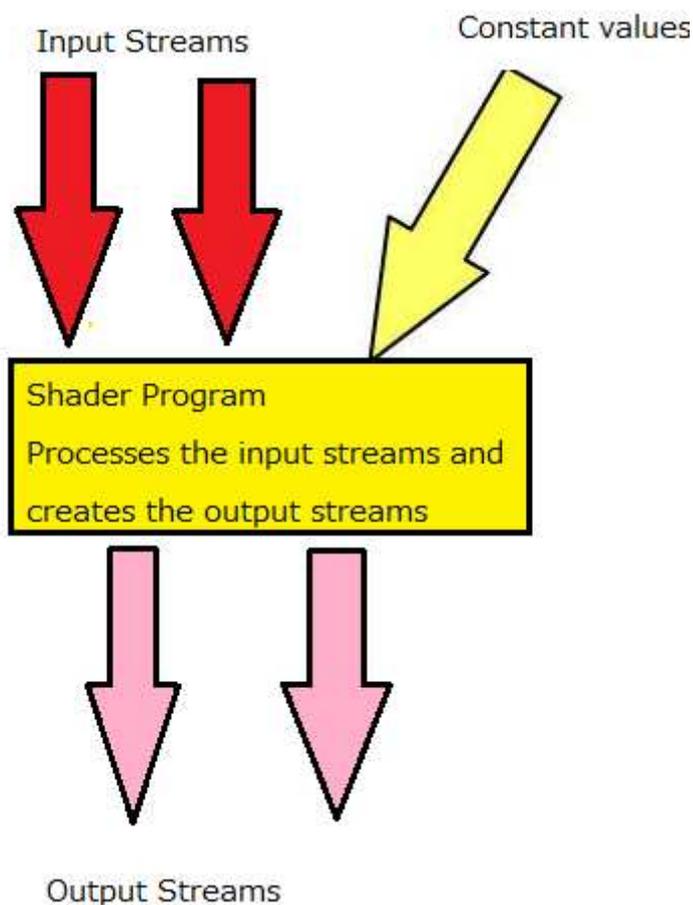
1.2 Authors

The author of the first version of this interface is Professor Shinichi Yamagiwa (SIPS group, INESC-ID Portugal). The changes performed in order to support the OpenCL version of the Caravela, were developed by Pablo Lamilla Álvarez.

2 Functionality

2.1 Basics concepts

As its name says, the FlowModelCreator creates a structure needed by the Caravela platform called flow-model. The flow-model is just a set of data that includes the information for a number of inputs, a number of outputs, a number of constants, constant values and a shader program. The FlowModelCreator will manage the way of packing all this information into a flow-model XML file. An example of a flow-model is shown in the next figure:

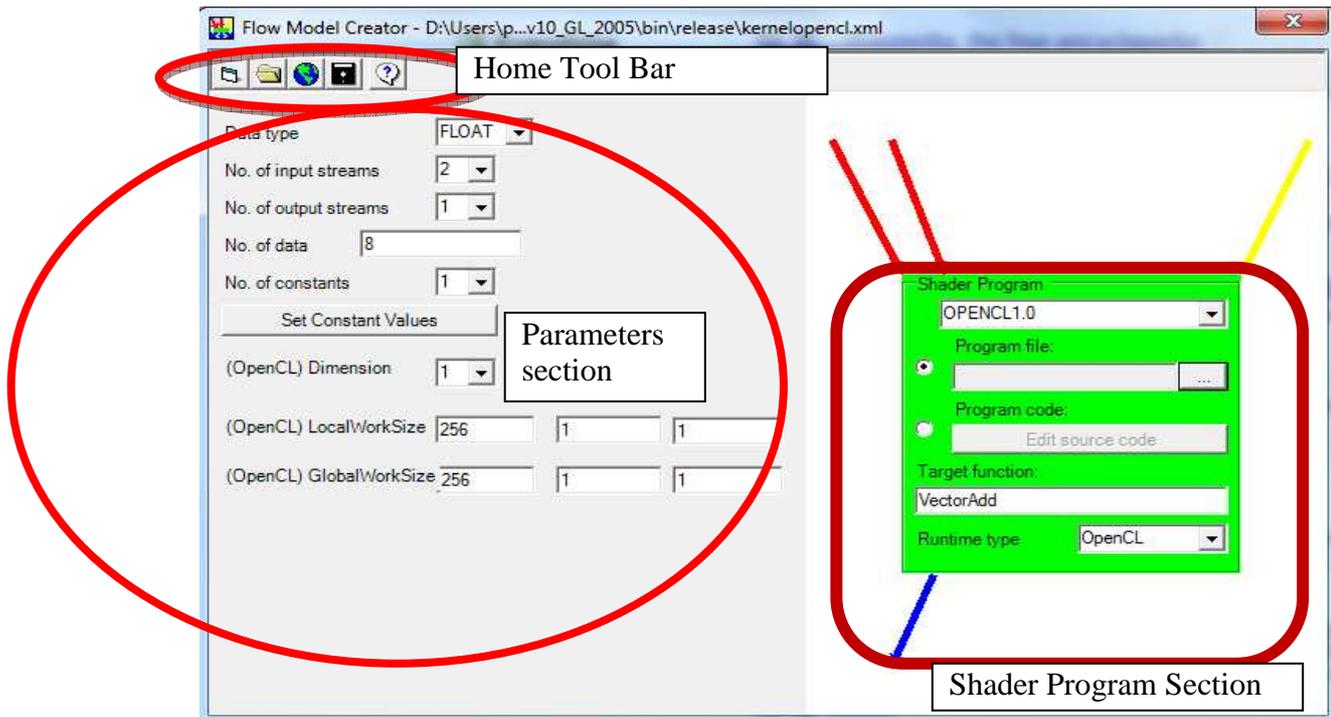


In this example, the flow-model has two input data streams, one constant stream and two output streams, and it processes the input streams by the shader program. This program can be written in HLSL (High Level Shader Language) of DirectX, in GLSL (OpenGL Shading Language), or in OpenCL.

The flow-model is stored in a XML file that follows the Caravela specification. Because it is very hard for the user to manage manually the file contents and format, this FlowModelCreator provides a very easy interface to the user.

2.2 Top Window

The following image illustrates the main window of the FlowModelCreator:



1) Home Tool Bar.

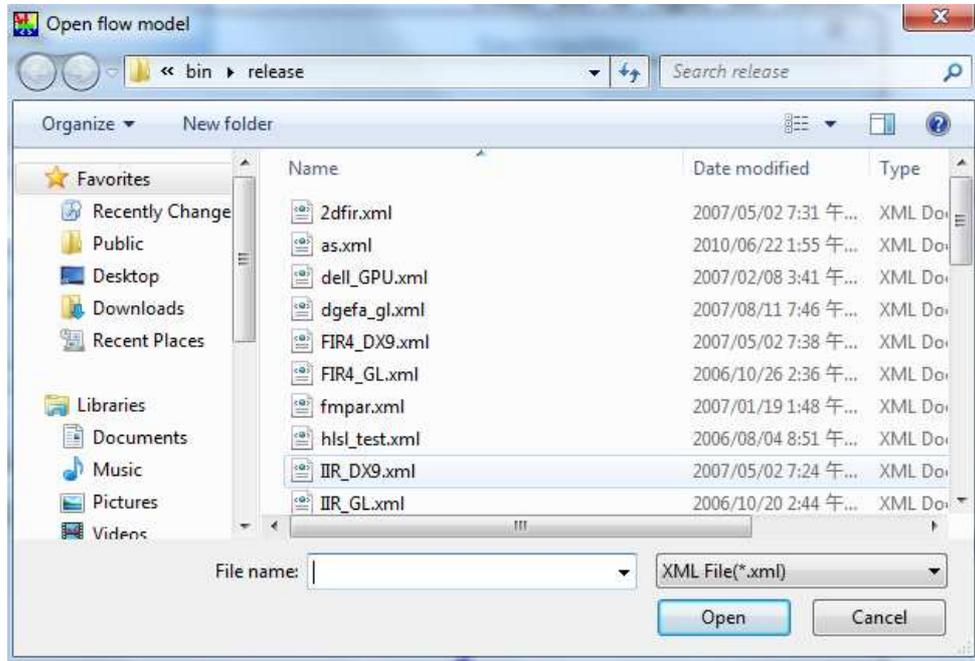
From left to right, the buttons of this tool bar contains the followings functionalities:

- i) Create New Flow Model 

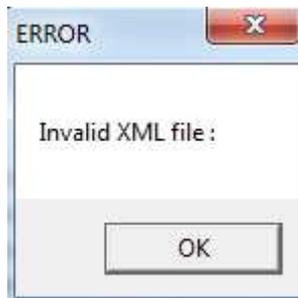
This button allows the creation of a flow-model from scratch. After clicking this button, you must choose the type of the input and output streams selecting one item from the “Data type” combo box in the Parameters section. After this action, the whole interface is unblocked and ready to use.

- ii) Open Local Flow Model File 

This button allows loading a previously saved flow-model from a local XML file. After clicking this button, the following window for the selection of the XML file appears:

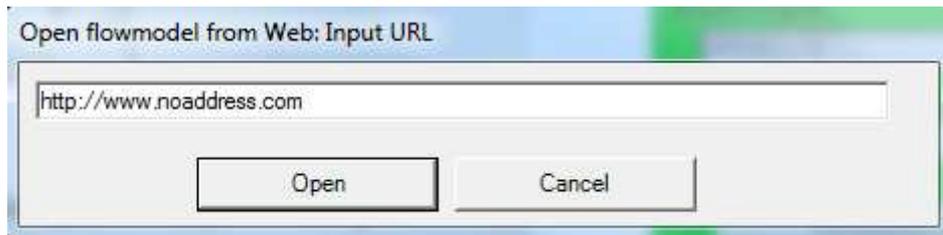


The flow-model will be loaded to the interface after selecting the file and clicking the button "Open". To return to the previous screen without loading anything press "Cancel". After pressing the "Open" button, if the file is not correct the following warning window will appear:

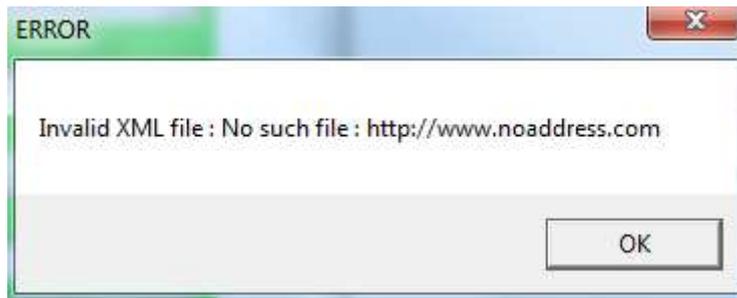


iii) Open Flow-Model From Web 

This button allows loading a flow-model XML file from an URL. After clicking this button the following window for introducing the web address appears:



After pressing the button “Open”, if the address is not correct the following error window appears:



iv) Save flow-model 

With this button, the current flow-model will be saved into a XML file.

The following errors may appear while saving a flow-model, not allowing you to save it:

- 1) **Number of Constant is strange.** This happens when the number of constants in the combo box does not match with the number of constants introduced in the “Set Constant Values” interface.
- 2) **Specify a shader program.** This error happens when a shader program, either via file or via code, is not introduced in the interface.
- 3) **Specify a target function name.** When using OpenGL or OpenCL, a function name must be written inside the textbox “Target function”.
- 4) **Invalid data size.** A value larger than 0 must be set in the “No. of Data” textbox.

v) Help 

This button displays a briefly tutorial for using the FlowModelCreator interface.

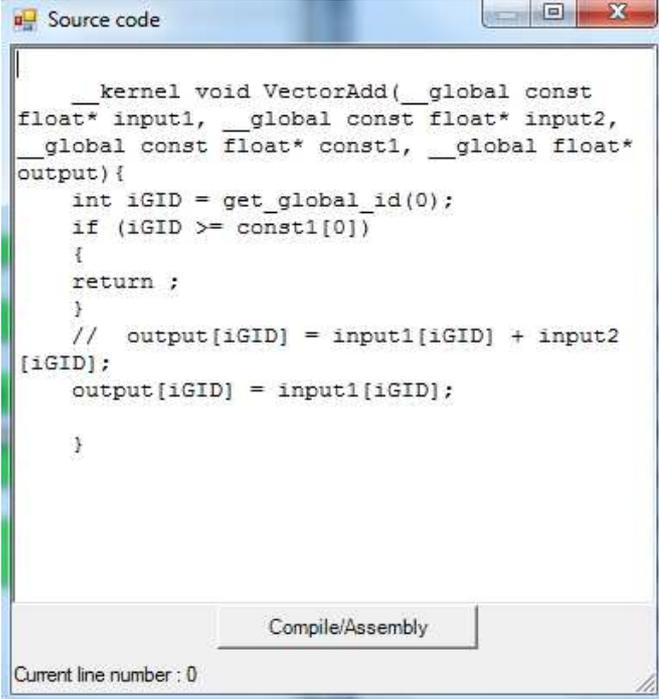
2) Parameters section

Located on the left side of the top window, in the parameters section the user can set the values of various parameters of the flow-model, such as the number of input and output streams, number of total data, number of constants and the type of the data streams. The last 3 parameters (Dimension, LocalWorkSize and GlobalWorkSize) are specific for the OpenCL version

3) Shader program section

On the right side of the top window inside a green square is located the shader program section. Here we can perform the following actions:

- i) Change the runtime (OpenCL, OpenGL, DirectX) we want to use to execute the flow-model.
- ii) Select the language we want to use to write the shader program.
- iii) Write the name of the target function (if we are using OpenGL or OpenCL).
- iv) Introduce a program shader. This can be done through two methods:
 - Load the program from an extern file.
 - Write the code directly using the “Edit source code” button. This action will open the following window where the user can write the code and compile it to check if is correct.



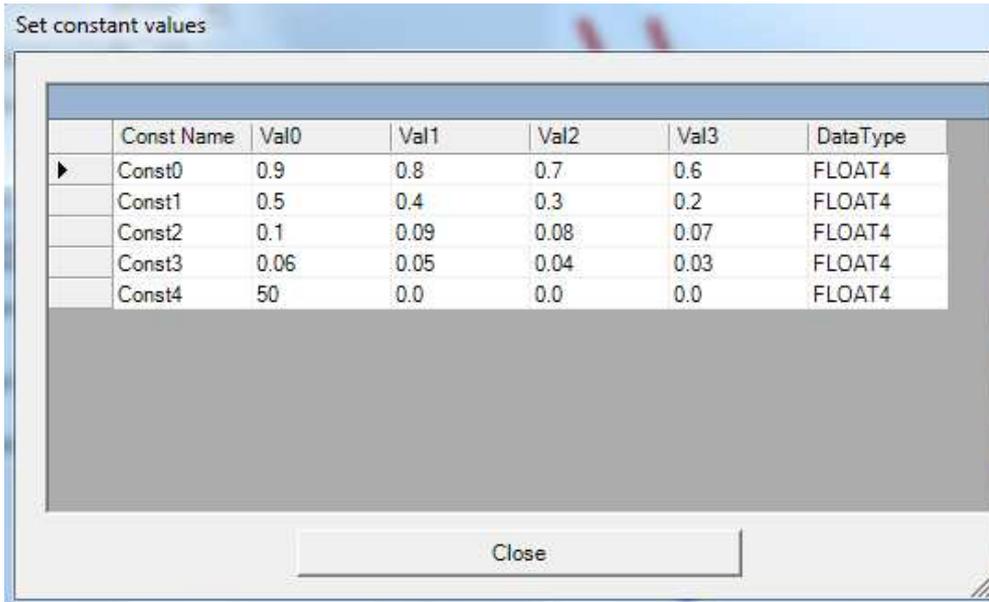
```
__kernel void VectorAdd(__global const
float* input1, __global const float* input2,
__global const float* const1, __global float*
output){
    int iGID = get_global_id(0);
    if (iGID >= const1[0])
    {
        return ;
    }
    // output[iGID] = input1[iGID] + input2
[iGID];
    output[iGID] = input1[iGID];
}

Compile/Assembly

Current line number : 0
```

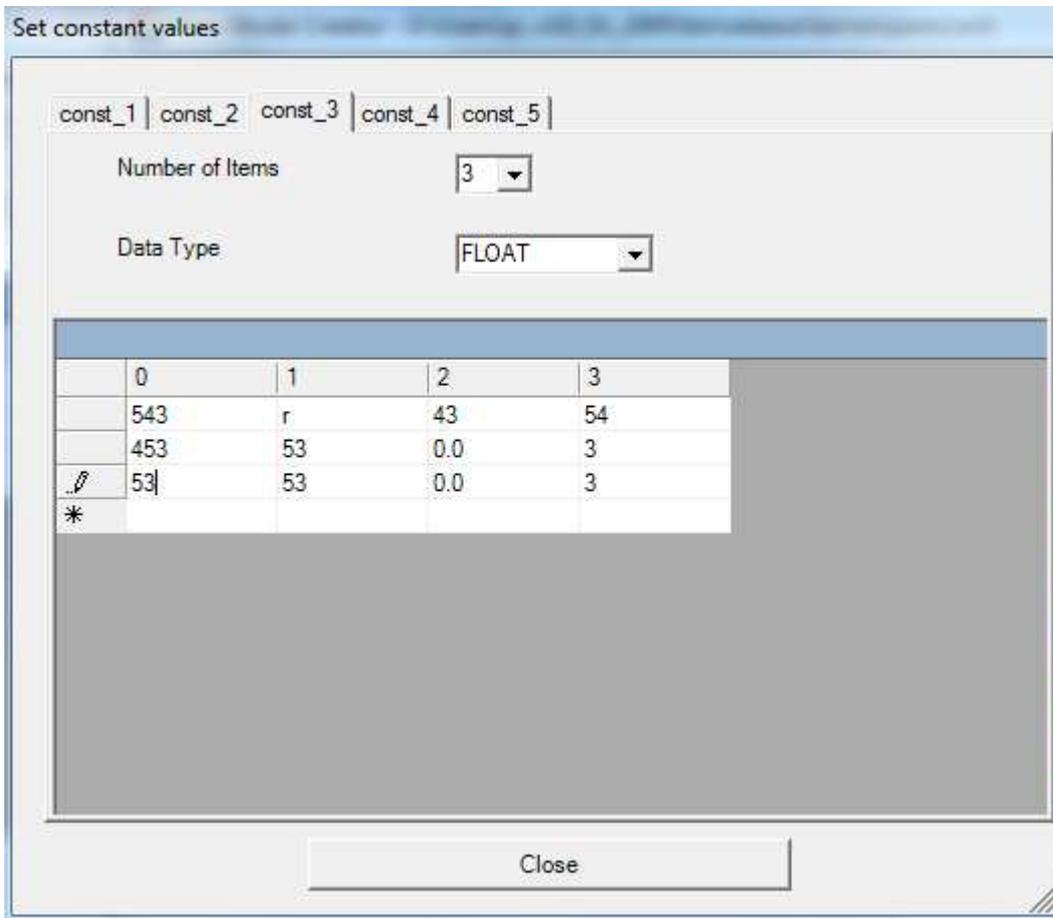
2.3 Set constant values window

The “Set constant values” window is used to introduce the values of the constant values of the flow-model. This interface has two different versions, one for OpenCL and another one for the rest of runtimes. Let’s see first the window for the OpenGL/DirectX runtimes:



In this interface the user can modify the names, the values and the data type of each constant. To change the number of constants (rows) that appear in this window, the user must use the combo box “No. of constants” located on the top window.

Next, the “Set constant values” screen for the OpenCL version is shown here:



For each constant value that the user has introduced using the combo box “No. of constants” located on the top window, this window creates one tab like the one shown before. Here each constant value is an array of constant values of the same data type. The name of each array (constant value) is generated automatically.

Inside each tab, the user can select the length of the array using the combo box “Number of Items” which increases or decreases the number of rows of the table. The array is structured from up to down and from left to right, so for instance the position of the number “453” (second row, first column) is the fifth position in the array. Then the user must select the data type of all the values inside the current array using the combo box “Data Type”. Finally, after introducing the values in the grid, when the user closes the window these values are saved in the interface, so if the user opens again this window the values remain.