International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 1, Number 2, pages 230–243, July 2011

A Uniform Platform to Support Multigenerational GPUs for High Performance Stream-based Computing

Pablo Lamilla Álvarez and Shinichi Yamagiwa
School of Information, Kochi University of Technology / JST PRESTO
Kami, Kochi, 782-8502 Japan

Masahiro Arai and Koichi Wada
Department of Computer Science, University of Tsukuba, 305-8573
Ibaraki, Tsukuba, Japan

Received: January 31, 2011
Revised: May 19, 2011
Accepted: June 20, 2011
Communicated by Yasuaki Ito

Abstract

GPU-based computing has become one of the popular high performance computing fields. The field is called GPGPU. This paper is focused on design and implementation of a uniform GPGPU application that is optimized for both the legacy and the recent GPU architectures. As a typical example of such the GPGPU application, this paper will discuss the uniform implementation of the Caravela platform. Especially the flow-model execution mechanism will be considered referring the recent GPU architectures. To verify the design and the implementation on CUDA and OpenCL platform, this paper will evaluate the compatibility among the architectures, and also test measurements of performance.

Keywords: GPGPU, Stream computing, OpenGL, CUDA, OpenCL

1 Introduction

Intensive computing has demanded the high performance stream-based computing on graphics processing units (GPUs) to be processed using the strong horse power [11]. The fastest supercomputer has reached to Peta FLOPS performance, which hires more than 35,000 CPU IC chips [14]. Each CPU IC chip has several CPU cores. In the meanwhile, a GPU IC chip is going to achieve TFLOPS-based performance, which embeds hundreds of stream-based processors. If not considering the programming aspect of the stream-based computing on GPUs, the performance attracts researchers working on the high performance computing field to the potential computing power of GPUs.

Architecture of GPUs has been altered in this decade. Mainly, two architectures are categorized to the well-known GPU architectures. One has three kinds of processors called a *vertex processor*, a *rasterizer* and a *fragment processor* that are dedicated respectively to perform vertex processing to operate perspectives of graphics objects, rasterizing to generate flat planes from the discrete vertices and finally coloring to the planes from the texture inputs. The vertex and fragment processors have become programmable. Dedicated to graphics processing, the heterogeneous organization has been controlled by graphics runtimes such as DirectX [2] and OpenGL [10].

Another integrates tens or hundreds of a standardized general purpose stream processors. Each processor invokes a program to generate each element in output data stream(s) from the 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 to the graphics processing, it releases the stream processor resources to general purpose computing via special runtimes such as CUDA [8] and OpenCL [9]. To distinguish the difference among the architectures above, the former architecture is generally called a *legacy* GPU.

During the era of the legacy GPU architecture, the GPGPU applications were developed as graphics processing application that generates results to pixels on a screen such as the computing mechanism reported in [6] although the interface for the graphics runtimes were covered by transparent graphics APIs such as [5] and [12]. Providing a compiler-oriented programming interface based on *kernel* function framework, the Brook [1] was developed to absorb the difficulty to plan the calculation displaying pixels as the result. Caravela platform [3, 13, 15, 16] was developed to form the true stream-based computing based on the *flow-model* that maintains the number of I/O streams, constant values and a program invoked on a targeted GPU. The recent GPU architecture makes available to provide the Brook style programming interface for GPGPU applications using CUDA or OpenCL.

As mentioned above, the programming styles between the legacy and the recent GPU architectures are not compatible. Therefore, not only any algorithm but also any optimization technique can not be shared between the architectures. However, just defining the number of I/O streams and a program, the Caravela Platform has a unified framework to implement stream-based computing using flow-model. Thus the Caravela platform can provide a unified programming style between the different GPU generations because the stream-based computing concept is standardized.

Caravela Platform is now implemented on the legacy GPU architecture. This paper is focused on a case study of migrating Caravela Platform to the recent GPU architecture on CUDA and OpenCL runtimes. Moreover, this paper will show performance evaluation with/without Caravela's framework and will evaluate efficiency of the mechanism. According to the discussion, we will try to find the same migration problem occurred in the GPGPU application.

The next section shows backgrounds of this research regarding GPGPU and explains advantages of Caravela platform in the stream-based computing using GPUs. In Section 3, this paper will show design and implementation of a new Caravela platform that supports CUDA and OpenCL. Section 4 illustrates the performance aspect of the Caravela platform, and finally section 5 concludes the paper.

2 Background and definitions

2.1 GPU and the architectures

A video adaptor that includes a GPU and a Video RAM (VRAM) is connected to a peripheral bus of a CPU as depicted in Figure 1. The video adaptor works as a peripheral device of the CPU, and its GPU is controlled by the CPU to help a part of visualization tasks in the system. To utilize the GPU as a computing resource for GPGPU applications, the CPU downloads the application program to the GPU's instruction memory and also prepares input data for the program. The program fetches the data and generates the result to the memory areas as categorized in Figure 1. The GPU reads/writes the VRAM directly to execute the calculation for the program. In this case, the original data is prepared in the main memory. The CPU copies the data to the VRAM. During the execution of the program, the GPU generates the results to the VRAM. The CPU copies the results from the VRAM to the main memory. On the other hand, it is available for the recent GPU to access the main memory directly. The area is *pinned*[18] not to be moved by a paging function of operating system.

Figure 2 shows two GPU architectures for a) the legacy and b) the recent architectures. The legacy GPU architecture includes one or more vertex processors, a rasterizer and multiple pixel

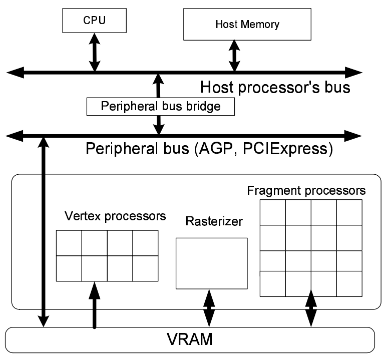


Figure 1: A typical system organization with a legacy GPU.

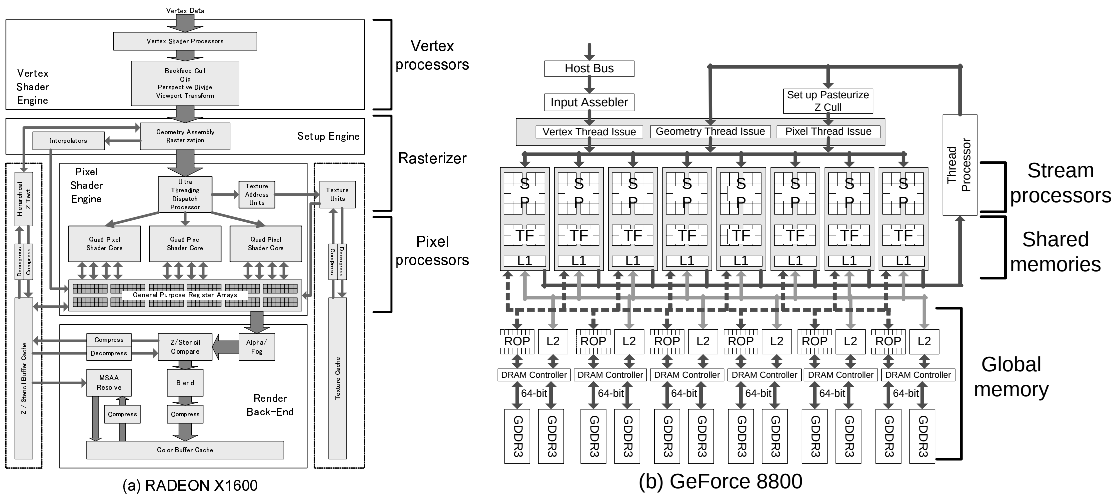


Figure 2: (a) the legacy and (b) the recent GPU architectures.

processors. The vertex processor receives coordinates of objects and maps it to a standardized axis with applying perspective operations to the object such as a rotation, a resize etc. The processor treats the coordinate with four dimensions (i.e. x, y, z and w) stored in a register. The standardized coordinates are passed to the rasterizer and it makes planes from the discrete vertices of objects. The planes are inputted to the pixel processor that colors the planes with reading texture data. The coloring operation is parallelized with multiple pixel processors, and also is operated with four color values (i.e. R, G, B and A). The final screen image is written to the video buffer. These processes to generate a graphics image are operated by the graphics runtimes such as OpenGL and DirectX. On the other hand, new architectures of GPUs have only a kind of processor called the *stream processor*. The processor works for general purpose processes in any kind of calculation. However, the computing style must be followed in stream-based one distributing elements included in streams into multiple stream processors. Employing a dedicated stream-based programming interface called CUDA, 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.

Moreover, the recent architecture has two types of memory called *global* and *shared* memories. The global memory is provided by the memory placed outside of GPU such as DDR3 VRAM. The shared memory is placed beside of the stream processor that works as if a cache.

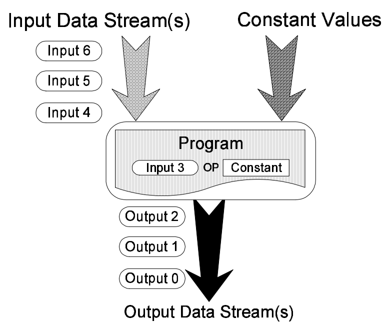


Figure 3: The structure of flow-model.

2.2 Stream-based computing platforms

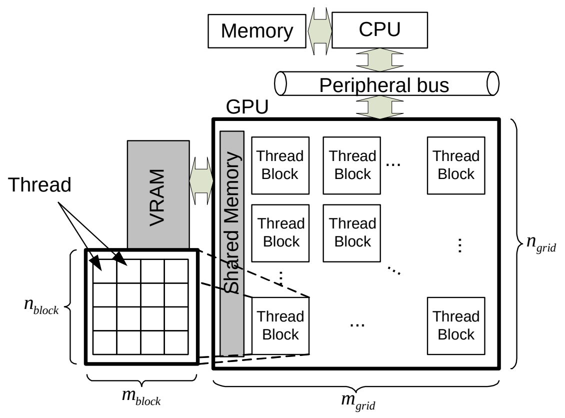
The researchers in the high performance computing field have ever encouraged to use the GPUs for a supercomputer platform as tried in the web site [4]. During the era of the legacy architecture, to solve the problem of disparities between graphics runtime environments and general purpose processing on GPUs, some solutions have been proposed, such as Sh, Scout, Brook and Caravela. Sh [12] is a graphics processing interface with an object oriented interface for C++. Scout [5] is another wrapper for graphics which uses a language based on C*. Although details of graphics runtime are hidden by these two systems, they are still targeted for visual applications. Therefore, the programmer could not completely eliminate graphical dependent environments or issues. Brook [1] was a compiler-oriented interface for GPU-based applications for which programmer just needs to identify functions to be transposed to programs on the fragment processor specified with a special keyword *kernel*. This computing style is inherited by the CUDA. Finally, the Caravela provides a stream-based computing platform that hides implementation details such as the calculation on the GPUs. The programmer can just concentrate to design a *flow-model* data structure that follows a stream computing manner. This paper is focused on the Caravela because it currently supports only the legacy GPUs and also has a potential possibility to port it on the recent GPU architecture.

2.2.1 Caravela platform

Caravela platform [3, 13, 15, 16] is an interface for stream-based computing implemented by the authors of this paper. The Caravela platform uses the concept of flow-model as shown in Figure 3 for programming a given stream-based computing task. The flow-model packs the stream-based computing concept itself, which 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 fit to any kind of stream processor such as the ones of the new 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. Therefore, the program can be written in GLSL and/or HLSL.

Applications on the platform use the *Caravela library* for mapping the flow-model into a GPU. The library has functions not only for execution of the flow-model but also optimization functions of the flow-model execution itself, called *swap mechanism*[17], which exchanges the input and the output buffers of flow-model in the VRAM side without copy operation between the host memory and the VRAM. This mechanism is very efficient especially under the GPU resources.

Currently the Caravela platform supports the legacy architecture of GPUs. Therefore, it must use the graphics runtime functions to perform the stream-based computation. The library functions from OpenGL or DirectX emulate to prepare an NxN pixel screen where the output data streams are outputted based on the unit of *pixel* (R, G, B and A). As long as the OpenGL and DirectX are used for the supported runtimes, the program in the flow-model is compiled at the execution time using the embedded interpreter in those runtimes. This means the program source is able to be



(a) Execution model of kernel threads in CUDA

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

Figure 4: Programming concepts in CUDA.

embedded to the flow-model.

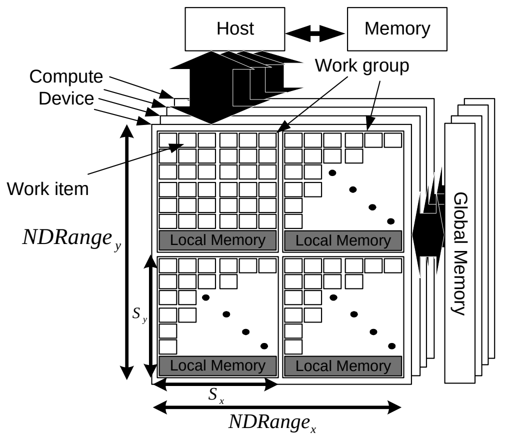
Contrarily, the runtime that supports the recent architecture of GPU such as the CUDA provided from NVIDIA treats massive data based on scholars without relationship to the graphics issue. Let us take a look at the overview the runtime in the next section.

2.2.2 CUDA

The Compute Unified Device Architecture (CUDA) has been proposed by NVIDIA Corporation [7]. The tools and APIs for programming on CUDA environment is now provided by the company's website. Although the programming environment is provided freely for use, the environment is validated only on the most recent of their graphics hardware available in the market.

The CUDA assumes an architecture model as illustrated in Figure 4 (a). The model defines a GPU which is connected to a CPU's peripheral bus. A VRAM (the global memory) that maintains data used for calculation on the GPU is connected to the GPU. The data is copied from the host memory before the CPU commands to execute a program on the GPU. The program is executed as a thread in a thread block. The thread may touch the host memory (called *pinned memory*). The thread blocks are tiled in a matrix of from one to three dimensions. In the figure, thread blocks are tiled in two dimension whose size is $n_{grid} \times m_{grid}$. Each thread block has multiple threads in a matrix whose size is varied from one to three dimensions. The figure also shows a thread block that includes $n_{block} \times m_{block}$ threads. Each thread block has individual shared memory space where shared valuables accessed among threads in the block are stored temporally. Thus, the program targeted to GPU in the CUDA environment is invoked as threads. The threads are grouped by the unit of the thread block. Therefore, to obtain a large parallelism, a large number of threads are invoked concurrently.

In the program on the CUDA environment, the threads are described as a stream-based function written in C called a *kernel function* as shown in Figure 4 (b). The program has two parts of the codes targeted to CPU and GPU, which is initially invoked by the CPU; a main program for CPU and a kernel function called as the thread on GPU. The kernel function 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. For example, using these global variables, Figure 4 (b) performs a summation of arrays A and B assigning each summation of the elements in those arrays to a thread and returns the result to the array C. The function is called by the main program specifying the sizes of the grid and the thread block with `<<< >>>`. Finally, reading data from the VRAM transferred by the main program, the kernel function is assigned to GPU, and runs as multiple threads. Thus, because programmer can just simply consider the stream-based kernel function and



(a) Platform model to execute a kernel function

```
...
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];
}”;
```

(b) Example array summation in OpenCL

Figure 5: The execution model and a kernel function example on OpenCL.

the calling code for the function in the main program, using the conventional C language manner, the CUDA provides an easy and transparent interface for GPGPU.

2.2.3 OpenCL

To standardize interface for massively parallel processors for stream-based computing, the OpenCL [9] is proposed by Khronos Group and now has become available to be applied to GPUs by the NVIDIA's and ATI's OpenCL APIs. The OpenCL defines a common platform models that include the processing element and the programming style. Figure 5(a) illustrates the platform model for the processing element. The host processor 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 includes the *work items*. The work item is a processor that processes a data unit in input data streams regarding the assigned ID. The ID is decided by a combination of the work group ID and the local work item ID, or by a global work item ID. The total number of the 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. Another parameter, called the *work group size* ,decides the number of work items in a work item group as the figure presents it with S_x and S_y. Thus, the compute unit manages the work groups to compute a kernel function that processes the corresponding data units specified by the ID.

The memory in the OpenCL is also categorized to the host, the global and the local memories. The host memory is managed by the host processor and is not controlled by the OpenCL side. The *global memory* is managed by the OpenCL function from the host side. The memory is accessed from all the work items. On the other hand, the *local memory* is accessed via the work items that work in the same work group. Therefore, it is used as a cache memory managed manually.

The OpenCL program is written in C as shown in Figure 5(b). It includes both the control code executed by the host and the kernel function distributed on the work items. The resources in the OpenCL are obtained by the *context* created by the `clCreateContextFromType` function. In the figure, the context for a GPU is defined by specifying `CL_DEVICE_TYPE_GPU` as its argument. The kernel function is provided by a source string defined as an array of char. The string is passed to the `clCreateProgramWithSource` function and build by the `clBuildProgram` function. Finally, the `clCreateKernel` function defines the kernel object specifying the top function name such as the "VectorAdd". The buffers for I/O data streams are allocated using a host function such as "new" or "malloc". The programmer can select if the buffers are accessed directly from the compute unit or the buffer mirrors are allocated in the global memory by passing a selective argument to the `clCreatebuffer` function. And then, the `clSetKernelArg` function connects the argument pointers of the kernel function to the actual buffers in the host and/or in the compute device. Finally, the kernel is executed by the `clEnqueueNDRangeKernel` function and the output data streams in the global memory are copied by the `clEnqueueReadBuffer` function.

According to the background mentioned above, let us summarize the objective of this paper and the main direction of our research theme. The recent GPU architecture has advantages in both aspects of performance and functionality because the architecture is dedicated to the stream-based computing concept applying the stream processor. On the other hand, the Caravela platform currently only supports the legacy GPU architectures. Therefore, it is not able to receive any merits from the recent architecture. The Caravela has potentially a stream-based computing style. When it supports the recent architecture, it will implement a heterogeneous environment that accepts any kind of programs for both legacy and recent GPUs. Thus, this paper is focused on how the GPGPU programs that only supports the legacy architecture can be migrated to the recent one. As a significant example, we pick up the Caravela platform and report the case study of the design and implementation issues for supporting the recent architecture in the Caravela platform.

3 A Uniform platform for multigenerational GPU Architectures

3.1 Design

We pickup CUDA and OpenCL to support the recent architecture in the Caravela platform. By supporting these environments, the Caravela platform will be able to take advantage of the new GPU architecture. First of all, let us focus on execution mechanism in the conventional Caravela to support the legacy architecture by the OpenGL and the DirectX. The main points to be considered to support the flow-model execution in the CUDA are listed below:

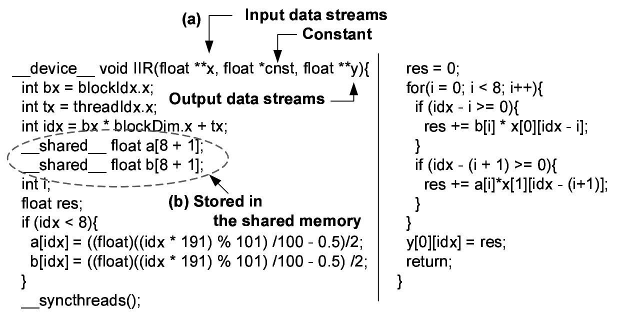
1. The flow-model includes a program that is written in the GLSL/DirectX assembly/HLSL. Its syntax is able to be checked during the programming phase by compiling it. However, the code is saved in the flow-model in text because it will be compiled at the use of the code. This mechanism is used in the OpenCL also because the runtime is extended from the OpenGL and is very familiar to the OpenGL graphics applications. On the other hand, the kernel program of the CUDA must be compiled by *nvcc* and saved in binary format. It will be linked to the executable of the host program. A new implementation mechanism must be considered to support the legacy mechanism for the CUDA case.

2. The format of the program embedded in a flow-model is specified by the kind of the lowest graphics runtime. For example, the program in a flow-model that is written in GLSL or HLSL demands the programmer to specify the textures for I/O streams as arguments of the main function and the constant values as the global ones that are initialized from the host side. Among the programs using the legacy runtimes, the rule for those arguments and the constants is unified. Therefore, it must be unified in the versions of the OpenCL and the CUDA as well.

3. The legacy architecture allows the I/O interfaces to accept textures for the input streams and video buffers for the output ones. However, as explained in the previous section, the recent GPU architecture includes three types of memory regions; *global*, *shared* and *pinned*. These memory types must be separately used for optimizing performance.

4. The swap mechanism is an excellent feature invented in the Caravela platform [17]. It must be implemented in the new architecture. The important point is to find if the three types of memory regions include suitable one that can swap the region directly by just exchanging the pointer or not.

Let us consider how to solve the focused points above that are needed to implement a unified interface when the Caravela is ported to the recent GPU architecture.



```
__device__ void IIR(float **x, float *cnst, float **y){
  int bx = blockIdx.x;
  int tx = threadIdx.x;
  int idx = bx * blockDim.x + tx;
  __shared__ float a[8 + 1];
  __shared__ 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;
  }
  __syncthreads();

  res = 0;
  for(i = 0; i < 8; i++){
    if (idx - i >= 0){
      res += b[i] * x[0][idx - i];
    }
    if (idx - (i + 1) >= 0){
      res += a[i]*x[1][idx - (i+1)];
    }
  }
  y[0][idx] = res;
  return;
}
```

Figure 6: An IIR program written in CUDA embedded in a flow-model.

3.2 Implementation

3.2.1 CUDA case

The Caravela over CUDA is implemented as maintaining the points below. The points solve the design issues discussed in the previous section.

The flow-model for the Caravela over CUDA includes a program written in the CUDA C. An example of an IIR filter program is shown in Figure 6. It is stored in an XML file with the parameters for I/O streams and the constants such as the numbers of I/Os and constants. In the case of the legacy runtime support, the program is saved in the flow-model as a text format because it is compiled at the execution timing. However, on CUDA, it must be compiled before the execution of the host program. Therefore, we have packed both a text and binary versions of the CUDA kernel in a flow-model. The text version is the code such as the one shown in Figure 6. The binary version is a PTX code generated by the nvcc compiler. The PTX is an intermediate language code of the CUDA kernel to be read dynamically and then invoked directly by the host program. Although another way stores a dynamic linked library made by the nvcc in the flow-model to be executed by the host program directly, it is not able to keep the compatibility for different kinds of OS. For example, DLL, used in Windows, does not have the compatibility to be invoked on Linux due to the binary format differences.

As coded in Figure 6(a), the I/O streams and the constant values are passed to the kernel function in the order specified by the rule. In the kernel function of CUDA, it is able to accept pointers of pointer (i.e. **x or **y). Therefore, it makes ease to define a unified program prototype. We have defined the program needs to include the arguments of input streams(**x), constant values (*cnst) and output streams(**y) respectively.

The most sensitive issue against performance is the treatment of memory regions among the ones of global, shared and pinned. In any case, the buffers for I/O streams specified in the flow-model are mapped to the global memory region. The *shared* memory region is used by the kernel program because the region is able to be managed directly from the kernel program without any setup from the host side. For example, the program shown in Figure 6 fetches data once from the shared memory (Figure 6(b)), and then uses the data in the kernel. This optimization eliminates accesses to the outside of GPU.

The *pinned* memory region is used for the swap mechanism. The region is allocated in the host memory and eliminated from the paging operation of the OS. Therefore, GPU can touch it directly. To implement the swap mechanism, the host exchanges the pointers to the I/O stream buffers when the kernel execution has been finished.

To support the swap mechanism, the buffers and the pointers of data streams are allocated as illustrated in Figure 7. The pointers to the buffers can be easily exchanged because those are allocated in the host memory. That means the copy operations at any recursive application will not be occurred in this implementation.

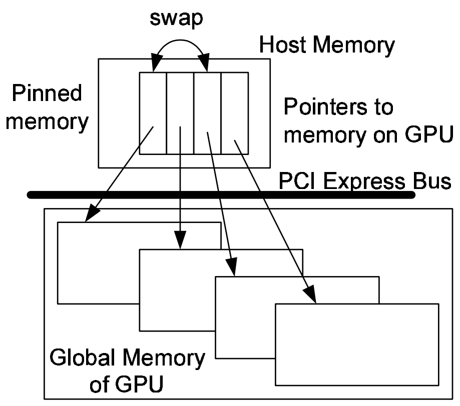
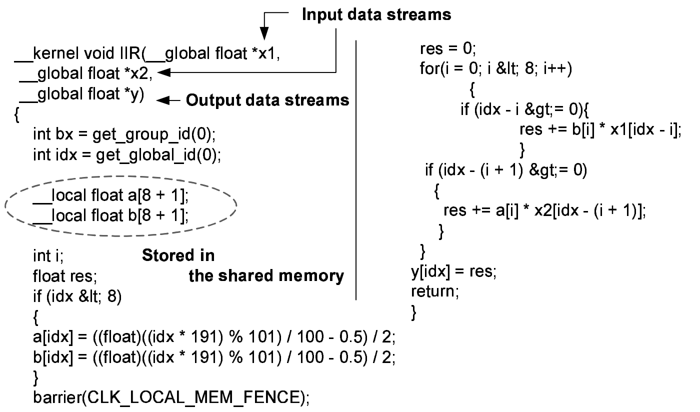


Figure 7: Memory allocation to support swap mechanism.


Input data streams
Output data streams
Stored in the shared memory

```
__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 &lt; 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 &lt; 8; i++)
        {
       if (idx - i &gt;= 0){
                res += b[i] * x1[idx - i];
                }
   if (idx - (i + 1) &gt;= 0)
     {
      res += a[i] * x2[idx - (i + 1)];
     }
  }
 y[idx] = res;
 return;
}
```

Figure 8: An IIR program written in OpenCL embedded in a flow-model.

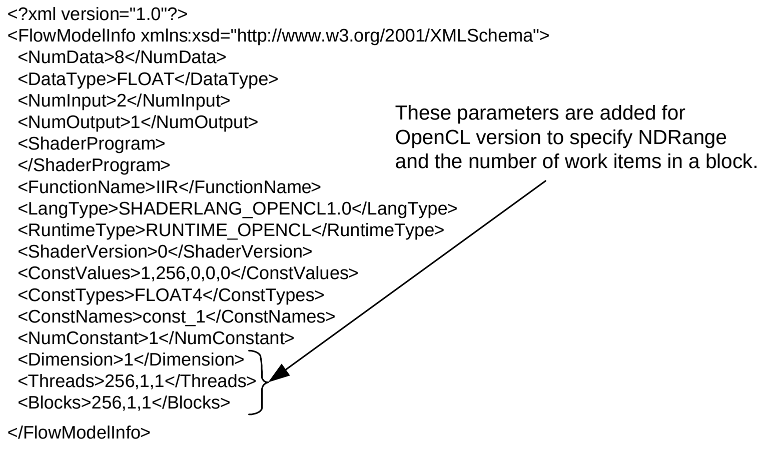
3.2.2 OpenCL case

As well as the CUDA case, we implemented the Caravela over OpenCL considering the points explained below;

The flow-model includes the kernel program written in OpenCL and contains additional parameter dedicated for specifying behavior in the OpenCL runtime. An example of a flow-model is shown in Figure 8. As we can see in the figure, the new parameters required by the OpenCL runtime have been added to the flow-model: the dimension of the NDRange structure (*Dimension*), the total number of work-items (*Threads*) and the number of work-items in a work-group (*Blocks*).

Figure 8 shows an IIR filter program written in OpenCL. The OpenCL runtime builds and compiles the program from a text format at the execution timing as in the legacy runtimes. Therefore, the program is stored in text format in the flow-model. The I/O streams and the constant values are passed as the arguments to the kernel function in the order specified by a rule we defined; the arguments are passed in the order of the input streams, the constants if needed and the output streams. Due to a fatal limitation in the OpenCL specification, the kernel function in OpenCL accepts the data streams passed in one dimensional arrays. A kernel function never accepts the pointers of pointer such as `**x`. Therefore, the arguments consist of all pointers of input and output streams that organize multiple dimensions. The constant values are also passed as one dimensional arrays.

The program shown in Figure 8 also illustrates the treatment of memory regions. Although the recent GPU architecture offers to the programmer three different memory regions; global, shared



```
<?xml version="1.0"?>
<FlowModelInfo xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <NumData>8</NumData>
  <DataType>FLOAT</DataType>
  <NumInput>2</NumInput>
  <NumOutput>1</NumOutput>
  <ShaderProgram>
  </ShaderProgram>
  <FunctionName>IIR</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>
```

These parameters are added for OpenCL version to specify NDRange and the number of work items in a block.

Figure 9: New parameters for OpenCL runtime in flow-model.

and pinned, the arguments in the kernels specified for I/O streams are mapped to the global memory region. The shared memory is used inside as the kernel caches data in the global memory as shown in Figure 8.

The swap mechanism is implemented using the OpenCL function called `clSetKernelArg` that sets a buffer for stream to an argument of the kernel function. Let us explain how the swap mechanism is implemented using the function to exchange the I/O buffers after each execution of the kernel. For example, the code of Figure 8 aims to swap the input stream `x1` with the output stream `y`. The Caravela runtime defines previously allocated buffers in the global memory, for instance *dst* and *src*. At the first execution, the `clSetKernelArg` function associates the *src* and the *dst* buffers to the `x1` and the `y`. After the execution of the kernel, the function exchanges the links between the `x1` and the `y` pointers using the same function. Finally, the *dst* buffer is accessed via the `x1` without explicit copy operation between buffers. Thus, the swap mechanism is implemented in the Caravela.

According to the implementation mentioned above, the flow-model for the recent architecture of GPUs fully presents the behavior implemented on the legacy architecture. Therefore, it is available for the Caravela to keep compatibility in the interface aspect and also to execute it in the same manner used in the conventional way. The special execution of the swap mechanism is also fully compatible with the conventional one. Thus, any GPGPU applications interfacing between the legacy and the recent GPU architectures will be implemented by using the equivalent techniques used in this section. We will discuss the performance aspect on the evaluation in the following section.

Thus, created a flow-model, the execution framework of Caravela supports multi-generational GPU architectures. Therefore, the user can just add another implementation for an additional architecture or adjust the program part of the flow-model to the architecture. The former case (adding a new code to the flow-model) will implement an execution mechanism that can invoke suitable program as the Caravela runtime will choose one of suitable implementations in the flow-model.

4 Experimental performance evaluation

The implementation availability to make compatibility between the legacy and the recent architectures has been shown in the previous sections. However, no performance discussion has been performed. This section evaluates performance differences between Caravela implementation on the legacy and the recent GPU architectures.

We used two applications to the evaluations; one is a matrix multiply that executes a straight

Table 1: Experimental environment for CUDA

CPU	Core i7 920 (2.66GHz) with 3GB DDR3
GPU	GeForce GTX280 (core:602MHz / DDR3 VRAM:1GB)
OS	Windows XP Professional SP3
CUDA	version 3.0

Table 2: Experimental environment for OpenCL

CPU	Core2Duo E7500 (2.93GHz) with 2GB DDR3
GPU	GeForce GT220 (core:625MHz / DDR3 VRAM:1GB)
OS	Windows 7 Professional SP3
OpenCL	Version 1.0

forward flow-model without iteration; another is an infinite impulse response (IIR) filter that iterates recursively the flow-model. We have measured the time to execute a CUDA program, an OpenCL one and a Caravela one that performs each flow-model of the two applications above. For CUDA, we measured two versions of kernels; one uses the global memory, another uses the shared memory. We also measured the execution times of OpenGL versions to compare the versions of the recent GPU architecture with the versions of the legacy one. The environments for the evaluations is listed in the Table 1 and the Table 2.

4.1 Evaluation using a straightforward application: Matrix multiply

The matrix multiply, which processes $A*B$ of NxN matrices, does not have any iteration or recursive I/O. Therefore, it will show the overhead when a large program is straightly invoked on GPU.

Figure 10(a) shows the execution times of the matrix multiply varying the input matrix size N from 2048 to 4096 comparing the versions on CUDA and the legacy platform. All CUDA versions with shared memory (SM) show better performances than the Caravela version over OpenGL because the flow-model execution mechanism of the OpenGL version potentially includes redundant processes following the graphics processing manner in the legacy architecture as depicted in Figure 2(a), and there is not any chance to optimize the GLSL code with techniques such as memory placement. When the CUDA uses the global memory (GM), the execution times have become 3.5-4.5 times longer than the ones of SM. Therefore, it means that any CUDA version needs to brush up the performance using the SM because the OpenGL version would achieve better performance than the one with GM.

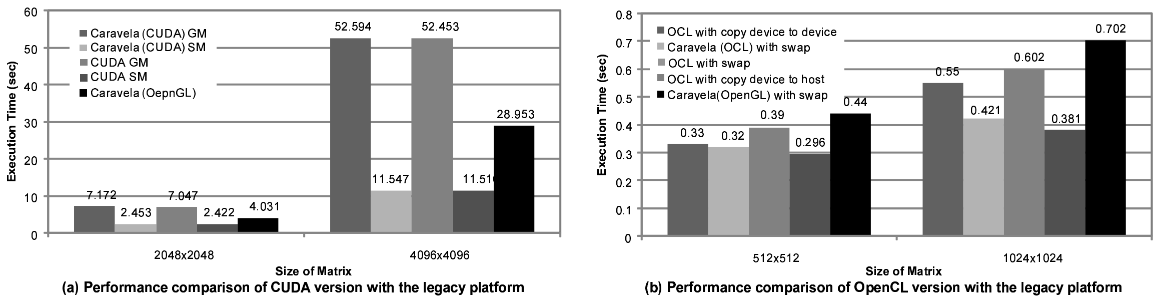
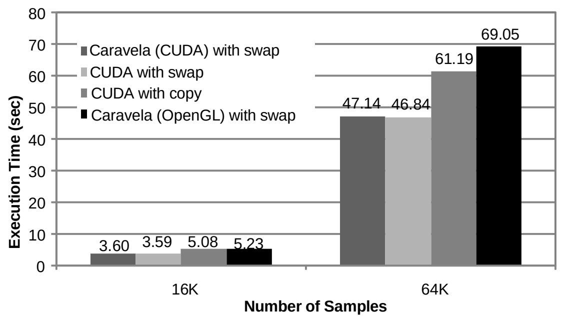


Figure 10: Execution times using matrix multiply.



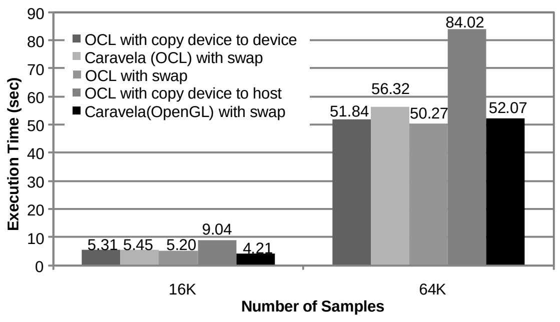


Figure 11: Execution times using IIR filter.

The performance difference with/without Caravela runtime shows the following considerable overheads. (1) The Caravela over CUDA or OpenCL needs to access the driver level or the runtime API via several dynamic linked libraries that causes calling overhead to load nested functions. (2) The Caravela over CUDA needs to load the flow-model and analyzes the XML structure. However, the CUDA and OpenCL versions include the kernel function in the programs compiled by nvcc or the text code embedded in the host program, and that does not need to be loaded.

Thus, the Caravela over CUDA or OpenCL keeps better performances than the ones in the Caravela over the legacy GPU architecture with OpenGL. In addition, the overhead of the Caravela runtime itself is not significant and is independent of the data size in the cases when the input data sizes vary. Therefore, we conclude that the implementation discussed in Section 3.2 is efficient.

4.2 Evaluation using a recursive application: IIR filter kernel

This evaluation analyzes the performance compatibility of the swap mechanism. The kernel code of the CUDA version and the OpenCL version are shown in Figure 6. In the flow-model, the code is packed in the program part. According to the performance results shown in Figure 11, the Caravela over OpenGL version shows the worst performance even if it uses the swap mechanism. This means that the swap mechanism that exchanges I/O streams using the OpenGL runtime function potentially contains overhead to maintain the graphics functions.

Comparing the OpenCL versions as depicted in Figure 11(b) that uses the swap mechanism (in the graph it is shown as "with swap") and the copy operation inside the GPU (in the graph it is shown as "copy device to device") using the method proposed in the previous section, we have confirmed that the Caravela over OpenCL achieves the performance that keeps the potential performance ("copy device to device") with only 10% performance degradation. The Caravela over OpenCL version keeps slightly lower performance than the pure OpenCL version due to the overheads caused by the just in time compilation for the kernel code with analysis of the flow-model's XML, and the setup times for the arguments in the host side program using the Caravela functions. In addition, every execution of the swap operation, the Caravela version needs to setup the buffer links among the arguments in the kernel program and the buffers allocated in the global memory. Moreover, the Caravela in each execution needs to access the driver level via several dynamic linked libraries. Thus, although the number of swaps affects to the execution time of the program this overhead is not affected by the data size.

When the swap mechanism was emulated by copy operations that migrates data copies among the memories of CPU and the GPU, the execution time increases. This means that the mechanism to exchange device memories that we have chosen in section 3.2 is effective. Thus, it is available to port the swap mechanism to the recent GPU architecture effectively.

All in all, in the both application programs, the comparison among the pure CUDA or OpenCL program and the Caravela over CUDA or OpenCL shows only slight performance degradation due to the Caravela's execution mechanism of flow-model. Thus, we have confirmed that the migration techniques from the legacy GPU architecture to the recent one of the execution mechanisms of flow-model discussed in this paper are validated.

5 Conclusions

This paper introduces methodologies and implementations of the stream-based platform that support both the legacy and the recent GPU architecture. We have supported the recent GPU architecture on the Caravela platform and the flow-model execution mechanism has been applied in a uniform way. According to the performance evaluation, we have confirmed that the methods to support multi-generational GPUs that we have shown in the paper validate the common interface and keep performance advantages.

Acknowledgment

This work is partially supported by the Japan Science Technology Agency (JST) PRESTO program.

References

[1] 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.

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

[3] Alexander S. Becker (Editor). *Concurrent and Parallel Computing: Theory, Implementation and Applications, Chapter1: Caravela: A High Performance Stream-based Concurrent Computing Platform.* NOVA Publishers, 2008.

[4] GPGPU homepage. http://www.gpgpu.org/.

[5] Patrick S. McCormick, Jeff Inman, James P. Ahrens, Charles Hansen, and Greg 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.

[6] 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.

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

[8] NVIDIA CUDA Zone. http://www.nvidia.com/cuda.

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

[10] OpenGL Architecture Review Board, Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 3 and 3.1.* Addison Wesley, 2009.

[11] 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.

[12] Sh: A high-level metaprogramming language for modern GPUs. http://libsh.org/.

[13] Leonel Sousa and Shinichi Yamagiwa. Caravela: A distributed stream-based computing platform. In *3rd HiPEAC Industrial Workshop*, May 2007.

[14] The Jaguar Supercomputer. http://www.nccs.gov/jaguar/.

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

[16] 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.

[17] 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.

[18] Shinichi Yamagiwa, Leonel Sousa, Kevin Ferreira, Keiichi Aoki, Masaaki Ono, and Koichi Wada. Maestro2: Experimental evaluation of communication performance improvement techniques in the link layer. *Journal of Interconnection Networks*, 7(2):295–318, 2006.