Stevina Dias, Sherrin Benjamin, Mitchell D'silva, Lynette Lopes / International Journal of Engineering Research and Applications (IJERA) ISSN: 2248-9622 www.ijera.com
Vol. 3, Issue 1, January -February 2013, pp.225-230

GPU Programming Models

Stevina Dias* Sherrin Benjamin* Mitchell D'silva* Lynette Lopes*
*Assistant Professor
Dwarkadas J Sanghavi College of Engineering, Vile Parle

Abstract

The CPU, the brains of the computer is being enhanced by another part of the computer – the Graphics Processing Unit (GPU), which is its soul. GPUs are optimized for taking huge batches of data and performing the same operation repeatedly and quickly, which is not the case with PC microprocessors. A CPU combined with GPU, can deliver better system performance, price, and power. The version 2 of Microsoft's Accelerator offers an efficient way for applications to implement array-processing operations. These operations use the parallel processing capabilities of multi-processor computers. CUDA is a parallel computing platform and programming model created by NVIDIA. CUDA provides both lower and a higher level APIs.

Keywords– GPU, GPU Programming CUDA, CUDA API, CGMA ratio, CUDA Memory Model, Accelerator, Data Parallel Arrays

I. INTRODUCTION

Computers have chips that facilitate the display of images to monitors. Each chip has different functionality. Intel's integrated graphics controller provides basic graphics that can display applications like Microsoft PowerPoint, low-resolution video and basic games. The GPU is a programmable and powerful computational device that goes far beyond basic graphics controller functions [1]. GPUs are special-purpose processors used for the display of three dimensional scenes. The capabilities of GPU are being harnessed to accelerate computational workloads in areas such as financial modelling, cutting-edge scientific research and oil and gas exploration. The GPU can now take on many multimedia tasks, such as speed up Adobe Flash video, translating video to different formats, image recognition, virus pattern matching and many more. But the challenge is to solve those problems that have an inherent parallel nature – video processing, image analysis, signal processing. The CPU is composed of few cores and a huge cache memory that can handle a few software threads at a time. Conversely, a GPU is composed of hundreds of cores that can handle thousands of threads simultaneously [2]. This ability of a GPU can accelerate some software by 100x over a CPU alone. The GPU achieves this acceleration while being more powerful and cost-efficient as compared to a CPU. This paper describes 2 different ways of programming a GPU namely Accelerator and CUDA.

II. ACCELERATOR

Accelerator is a high-level data parallel library which uses parallel processors such as the GPU or multi core CPU to speed up execution. The Accelerator application programming interface (API) helps in implementing a wide variety of array-processing operations by supporting a functional programming model. All the details of parallelizing and running the computation on the selected target processor, including GPUs and multi core CPUs are handled by the accelerator. The Accelerator API is almost processor independent, so the same array-processing code runs on any supported processor with only minor changes [3].

The basic steps in Accelerator programming are:

i. Create data arrays.
ii. Load the data arrays into Accelerator data-parallel array objects.
iii. Process the data-parallel array objects using a series of Accelerator operations.
iv. Create a result object.
v. Evaluate the result object on a target.

A. Data Parallel Array Objects

Applications do not have direct access to the array data and cannot manipulate the array by index [3]. Applications implement array processing schemes by applying Accelerator operations to data-parallel array objects—which represent an entire array as a unit—rather than working with individual array elements.

5 data-parallel array objects supported by Accelerator v2 are:

i. BoolParallelArray
ii. DoubleParallelArray
iii. Float4ParallelArray
iv. FloatParallelArray
v. IntParallelArray

An example for Float4ParallelArray is as follows:

```
typedef struct
 { float a;
        float b;
        float c;
        float d;  } float4;
```

Float4 is an Accelerator structure that contains a quadruplet of float values. It is used primarily in graphics programming. float4 xyz (0.7f, 5.0f, 4.2f, 9.0f). The API includes a large collection of operations that applications can use to manipulate the contents of arrays and to combine the arrays in various ways. Most operations take one or more data-parallel array objects as input, and return the processed data as a new data-parallel object. Accelerator operations work with copies of the input objects; they do not modify the original objects. Operation inputs are typically data-parallel objects, but some operations can also take constant values.

Table 1 Type of Accelerator Operations

Operation	Explanation
Creation and conversion	Create new data-parallel array objects and convert them from one type to another.
Element-wise	Operate on each element of one or more data-parallel array objects and return a new object with the same dimensions as the originals. ▪ **Add**. ▪ **Abs** ▪ **Subtract** ▪ **Divide , etc**
Reduction	Reduce the rank of a data-parallel array object by applying a function across 1D or more dimensions.
Transform	Transform the organization of the elements in a data-parallel array object. These operations reorganize the data in the object, but do not require computation. For example: ▪ **Transpose** ▪ **Pad**
Linear algebra	Perform standard matrix operations on data-parallel array objects, including matrix multiplication, scalar product, and outer product.

B. Sample Code

```
using System;
using Microsoft.ParallelArrays;
using FloatArray =
Microsoft.ParallelArrays.FloatParallelArray;
using PArray =
Microsoft.ParallelArrays.ParallelArrays;
namespace SubArrays
{
class Calculation
  {
staticvoid Main(string[] args)
    {
int arraySize = 100;
Random ranf = newRandom();
float[] Array1 = newfloat[arraySize];
float[] Array2 = newfloat[arraySize];
float[] Array3 = newfloat[arraySize];
DX9Target result = newDX9Target();          // (1)
for (int i = 0; i < arraySize; i++)                // (2)
    {
      Array1[i] = (float)(Math.Sin((double)i / 10.0) +
ranf.NextDouble() / 5.0);
      Array2[i] = (float)(Math.Sin((double)i / 10.0) +
ranf.NextDouble() / 5.0);
    }
FloatArray fpInput1 = new FloatArray (Array1);
                                                   // (3)
FloatArray fpInput2 = new FloatArray (Array2);
FloatArray fpStacked = PArray.Subtract(fpInput1,
fpInput2);                                         // (4)
FloatArray fpOutput = PArray.Divide(fpStacked, 2);
 Array3 = result.ToArray1D(fpOutput);        // (5)
for (int i = 0; i < arrayLength; i++)           // (6)
{
Console.WriteLine(Array3[i].ToString());    //(7)
} } } }
```

The above code uses two commonly used types: ParallelArrays and FloatParallelArray. ParallelArrays represented by PArray, contains the Accelerator operation methods [3]. FloatParallelArray represented by FloatArray, contains floating point arrays in the Accelerator environment.

1. Create a target object: Each processor that supports Accelerator has one or more target objects. Target Objects convert Accelerator operations to processor-specific code and run them on the processor. StackArrays uses DX9Target, which runs Accelerator operations on any DirectX 9-compatible GPU, using the DirectX 9 API.

2. Create input data arrays: the input arrays for StackArrays are generated by the application, but they can also be created from stored data.

3. Load each input array: Each input array is loaded into an Accelerator data parallel array object.

4. Array Processing: Use one or more Accelerator operations to process the arrays.

5. Evaluation: evaluate the results of the series of operations by passing the result object to the target's ToArray1D method. ToArray1D evaluates the 1D arrays and ToArray2D, evaluates 2-D arrays.

6. Process the result: Stack Arrays displays the elements of the processed array in the console window.

C. Accelerator Architecture

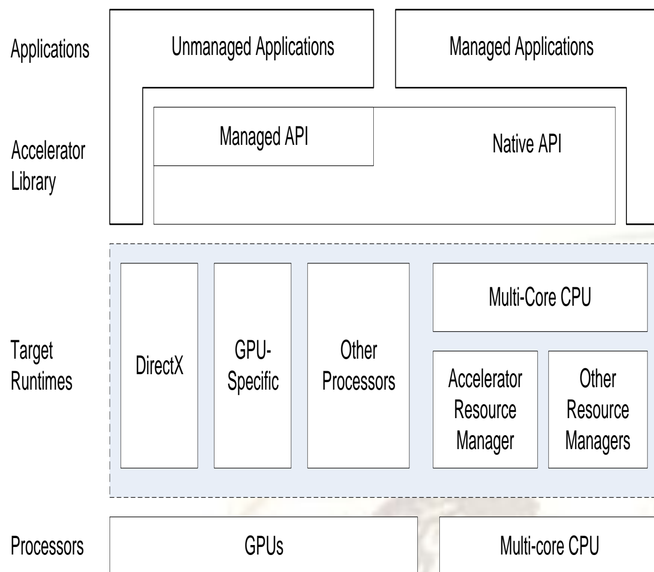


Figure 1: Accelerator Architecture

1) The Accelerator Library and APIs

The Accelerator library is used by the applications to implement the processor-independent aspects of Accelerator programming. The library exposes two APIs namely Managed and Native. C++ applications use the native API, which is implemented as unmanaged C++ classes. Managed applications use the managed API.

2) Targets

Accelerator needs a set of target objects. Each target translates Accelerator operations and data objects into a suitable form for its associated processor and runs the computation on the processor. A processor can have multiple targets, each of which accesses the processor in a different way. For example, most GPUs will probably have DirectX 9 and DirectX 11 targets, and possibly a vendor-implemented target to support processor-specific technologies. Each target consists of a small API, which are called by applications to interact with the target. The most commonly used methods are**ToArray1D** and **ToArray2D**.

3) Processor Hardware

The version 2 of Accelerator can run operations on different targets. Currently it includes targets for: Multicore x64 CPUs, GPUs, by using DirectX 9

III. CUDA

CUDA is a high level language which stands for Compute Unified Device Architecture. It is a parallel computing platform and programming model created by NVIDIA. The CUDA platform is accessible to software developers through CUDA-accelerated libraries, compiler directives and extensions to industry-standard programming languages, including C, C++ and FORTRAN. Programmers use 'C/C++' with CUDA extensions to express parallelism, data locality, and thread cooperation, which is compiled with "nvcc", NVIDIA's LLVM-based C/C++ compiler, to code algorithms for execution on the GPU.CUDA enables heterogeneous systems(i.e., CPU+GPU). CPU & GPU are separate devices with separate DRAMs Scale to 100's of cores, 1000's of parallel threads. CUDA is an Extension to the C Programming Language [4].

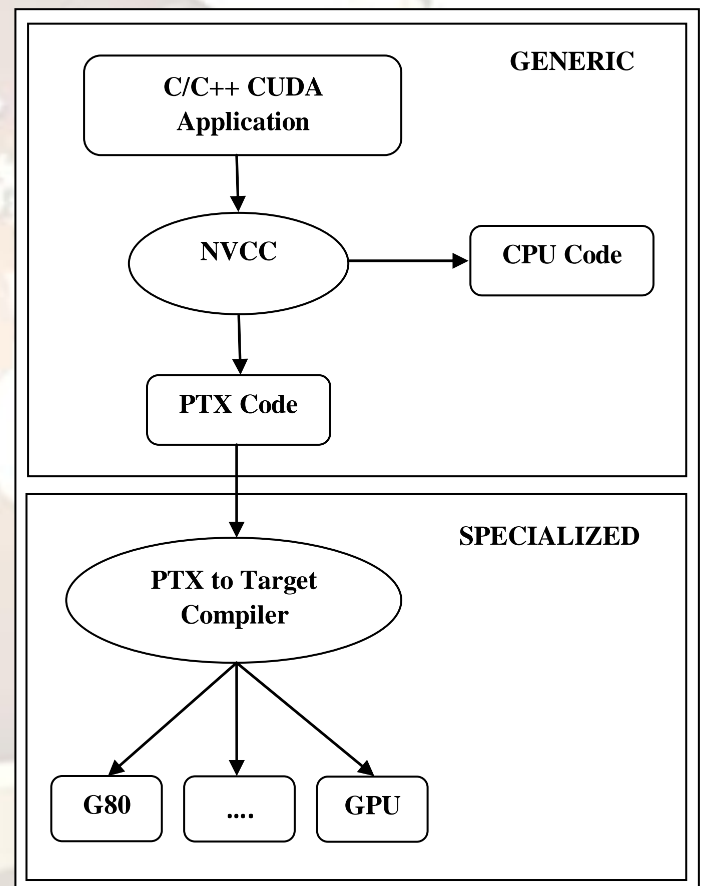


Figure 2: Compilation Procedure of CUDA

CUDA Programming Basics consists of CUDA API basics, Programming model and its Memory model

A. CUDA API Basics:

There are three basic APIs in CUDA namely, Function type qualifiers, Variable type qualifiers and Built-in variables. Function type qualifiers are used to specify execution on host or device. Variable type qualifiers are used to specify

the memory location on the device. Four built-in variables are used to specify the grid and block dimensions and the block and thread indices.

1) Function Type Qualifiers

Table 2 Function type Qualifier

Function Type Qualifier	Executed on	Callable from
__device__	Device	Device
__global__	Device	Host
__host__	Host	Host

2) Variable Type Qualifier

Table 3 Variable Type Qualifier

Variable Type Qualifier	Location	Lifetime	Accessible from
__device__	Global memory space	lifetime of an application	all the threads within the grid and from the host through the runtime library
__constant__	Constant memory space	lifetime of an application	all the threads within the grid and from the host through the runtime library (optionally used together with __device__)
__shared__	Shared memory space	lifetime of an block	all the threads within the block (optionally used together with __device__)

3) Built in Variables

Table 4 Built in Variables

Built in variables	Type	Explanation
gridDim	dim3	dimensions of the grid
blockIdx	uint3	block index within the grid
blockDim	dim3	dimensions of the block
threadIdx	uint3	thread index within the block

4) Execution Configuration (EC):

EC must be specified for any call to a __global__ function. Defines the dimension of the grid and blocks specified by inserting an expression between function name and argument list:

Function Declaration:
__global__ void function_name(float* parameter)

Function Call:
function_name <<< Dg, Db, Ns >>> (parameter)
Where Dg, Db, Ns are: Dg is of type dim3, dimension and size of the grid Dg.x * Dg.y = number of blocks being launched; Db is of type dim3, dimension and size of each block Db.x * Db.y * Db.z = number of threads per block.

B. Programming Model

The GPU is seen as a computing device to execute a portion of an application that has to be executed several times, can be isolated as a function and works independently on different data. Such a function can be compiled to run on the device. The resulting program is called a Kernel. The batch of threads that executes a kernel is organized as a grid of thread blocks.

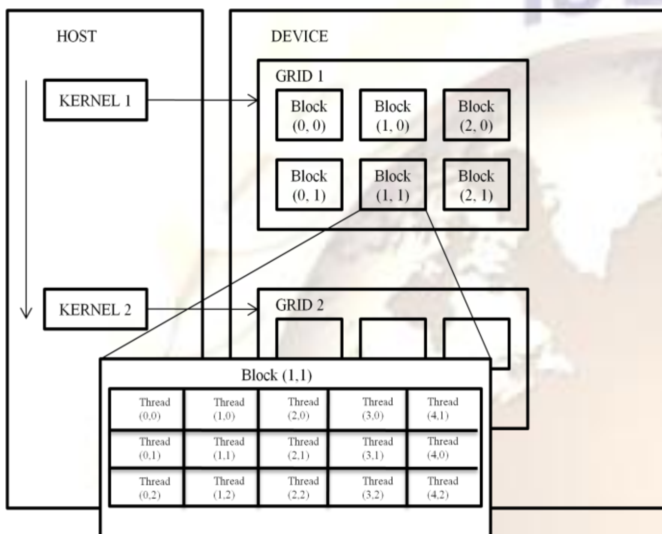
Following are steps in CUDA code
Step 1: Initialize the device (GPU)
Step 2: Allocate memory on the device
Step 3: Copy the data from host array to device array
Step 4: Execute **kernel** on device
Step 5: Copy data from device (GPU) to host
Step 6: Free allocated memories on the device and host

Kernel Code (xx_kernel.cu): A kernel is a function callable from the host and executed on the CUDA device -- simultaneously by many threads in parallel. Calling a kernel involves specifying the name of the kernel plus an execution configuration.

C. The Memory Model

Each CUDA device has several memories as shown in Figure 3. These memories can be used by programmers to achieve high CGMA (Compute to Global Memory Access) ratio and thus high execution speed in their kernels. CGMA ratio is the number of floating-point calculations performed for each access to the global memory within a region of a CUDA program. Variables that reside in shared memories and registers can be accessed at very high speed in a parallel manner. Each thread can only access registers that are allocated to them. A kernel function uses registers to store frequently accessed variables. These variables are private to each thread.

Shared memories are allocated to thread blocks. All threads in a block can access variables in the shared memory locations of the block. Threads can share their results via Shared memories. The global memory can be accessed by all the threads at anytime of program execution. The constant memory allows faster and more parallel data access paths for CUDA kernel execution than the global memory. Texture memory is read from kernels using device functions called texture fetches. The first parameter of a texture fetch specifies an object called a texture reference. A texture reference defines which part of texture memory is fetched.



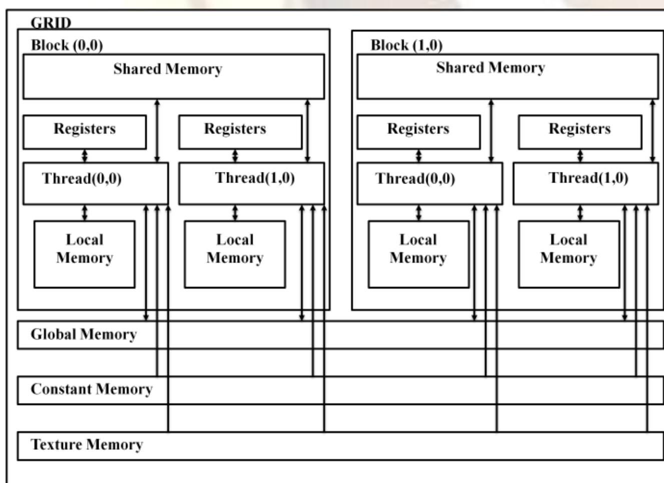


Figure 3: CUDA Memory Model

CUDA defines registers, shared memory, and constant memory that can be accessed at higher speed and in a more parallel manner than the global memory. Using these memories effectively will likely require re-design of the algorithm. It is important for CUDA programmers to be aware of the limited sizes of these special memories. Their capacities are implementation dependent. Once their capacities are exceeded, they become limiting factors for the number of threads that can be assigned to each SM [5].

Table 2 Different types of CUDA Memory

Memory	Location	Cached	Access	Who
Local	Off-chip	No	Read/Write	One Thread
Shared	On-chip	N/A	Read/write	All threads in a block
Global	Off-chip	No	Read/write	All threads + CPU
Constant	Off-chip	Yes	Read	All threads + CPU
Texture	Off-chip	Yes	Read	All threads + CPU

D. An example of a CUDA program

The following program calculates and prints the square root of first 1000 integers.

```
#include <stdio.h>                              // (1)
#include <cuda.h>
#include <conio.h>

__global__ void sq_rt(float*a,int N)            // (2)
{intidx=blockIdx.x*blockDim.x+threadIdx.x;
        if(idx<N)
                a[idx]=sqrt(a[idx]);}

 int main(void)                                 // (3)

{float*arr_host,*arr_dev;                       // (4)

        const int C=100;                        // (5)
        size_t length=C*sizeof(float);
        arr_host= (float*)malloc(length);       // (6)

        cudaMalloc((void**)&arr_dev,length);// (7)

        for(int i=0;i<C;i++)
        arr_host[i]=(float)i;

        cudaMemcpy(arr_dev,arr_host,
        length,cudaMemcpyHostToDevice);// (8)
```

```
int blk_size=4;                                    //  (9)
int n_blk=C/blk_size+(C%blk_size==0);
sq_rt<<<n_blk,blk_size>>>(arr_dev,C);

cudaMemcpy(arr_host,arr_dev,length,cudaMemcpyDeviceToHost);   // (10)

        for (int i=0;i<C;i++)                      //(11)
        printf("%d\t%f\n",i,arr_host[i]);

        free (arr_host);                           //  (12)
        cudaFree(arr_dev);
        getch();}
```

1. Include header files
2. Kernel function that executes on the CUDA device
3. main ( ) routine, that the CPU must find
4. Defines pointers to host and device arrays
5. Defines other variables used in the program, size_t is an unsigned integer type of at least 16 bit
6. Allocate array on the host
7. Allocate array on device (DRAM of the GPU)
8. Copy the data from host array to device array.
9. Kernel Call, Execution Configuration
10. Retrieve result from device to host in the host memory
11. Print result
12. Free allocated memories on the device and host

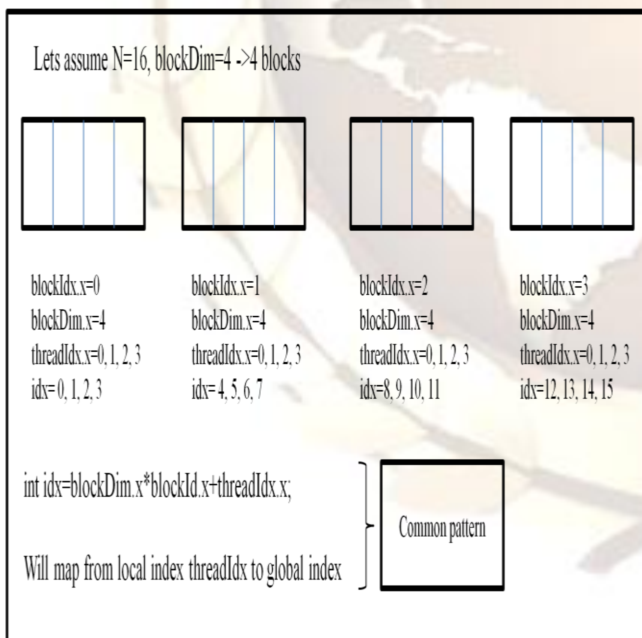


Figure 4: Illustration of code

CONCLUSION

The stagnation of CPU clock speeds has brought parallel computing, multi-core architectures, and hardware-acceleration techniques back to the forefront of computational science and engineering.A GPU allows you to run the single processor applications in parallel processing environment. In this paper, we introduced GPU computing, CUDA and Accelerator programming environment by explaining, step-by-step, how to implement an efficient GPU-enabled application. The above mentioned programming models help to achieve the goal of Parallel Processing. Due to the presence of these programming models and many more models, the gaming applications have become more lively and creative.

REFERENCES

[1] http://blogs.nvidia.com/2009/12/whats-the-difference-between-a-cpu-and-a-gpu/

[2] http://barbagroup.bu.edu/gpuatbu/Program_files/Cruz _gpuComputing09.pdf

[3] research.microsoft.com/en.../accelerator/accelerator_i ntro.docx 11/12/2012 4:06 PM

[4] http://en.wikipedia.org/wiki/CUDA 11/12/2012 3:52 PM

[5] www.cs.duke.edu/courses/fall09/cps296.3/cuda_docs/NVIDIA_CUDA_ProgrammingGuide_2.3.pdf

[6] De Donno, D.; Esposito, A.; Tarricone, L.; Catarinucci, L. Antennas and Propagation Magazine, IEEE Volume: 52, Issue: 3, "Introduction to GPU Computing and CUDA Programming: A Case Study on FDTD [EM Programmer's Notebook]"