An INTRODUCTION TO CUDA Programming

Irina Mocanu
University POLITEHNICA Bucharest
irina.mocanu@cs.pub.ro

Abstract. The graphics boards have become so powerful that they are usded for mathematical computations, such as matrix multiplication and transposition, which are required for complex visual and physics simulations in computer games. NVIDIA has supported this trend by releasing the **CUDA** (Compute Unified Device Architecture) interface library to allow applications developers to write code that can be uploaded into an NVIDIA-based card for execution by NVIDIA's massively parallel GPUs. This paper is an introduction to the CUDA programming based on the documentation from [2] and [4].

Introduction

The programmable graphics processor unit (GPU) has evolved into an absolute computing workhorse. Today's GPUs offer a lot of resources for both graphics and non-graphics processing. Data-parallel processing maps data elements to parallel processing threads. Many applications that process large data sets such as arrays can use a data-parallel programming model to speed up the computations. In 3D rendering large sets of pixels and vertices are mapped to parallel threads. Similarly, image and media processing applications can map image blocks and pixels to parallel processing threads. A lot of any other algorithms except the image rendering and processing algorithms are accelerated by data-parallel processing.

In this scope, Nvidia developed CUDA (**Compute Unified Device Architecture**) [1], a new hardware and software architecture for issuing and managing computations on the GPU as a data-parallel computing device without the need of mapping them to a graphics API. It is available for the GeForce 8 Series, Quadro FX 5600/4600, and Tesla solutions.

The paper presents the principal features of CUDA programming. It is presented the CUDA architecture and the application programming interface in CUDA, based on the documentation from [2] and [4]. Also there is an simple CUDA program for adding two matrix which uses the parallel capabilities of CUDA, which was presented in [3].

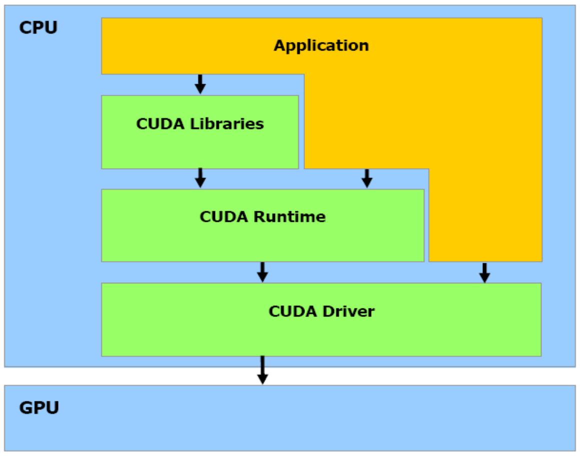

Figure 1. The CUDA software stack

The CUDA Architecture

CUDA is a framework which works in a modern massively-parallel environment. CUDA-enabled graphics processors operate as co-processors within the host computer. This means that each GPU is considered to have its own memory and processing elements that are separate from the host computer. To perform useful work, data must be transferred between the memory space of the host computer and CUDA device(s).

As in [2], the CUDA software stack is composed of several layers: (i) a hardware driver, (ii) an application programming interface (API) and its runtime, (iii) two higher-level mathematical libraries of common usage, as in Figure 1.

CUDA provides general DRAM memory addressing [2]: the ability to read and write data at any location in DRAM, just like on a CPU. CUDA has a parallel data cache with very fast general read and write access, that threads use to share data with each other.

When programmed with CUDA, the GPU is viewed as a **compute device** capable of executing a very high number of threads in parallel. It operates as a coprocessor to the main CPU (**host**). The host and the device maintain their own DRAM, **the host memory** and **device memory**, respectively, as in [2].

The batch of threads that executes a kernel is organized as a grid of thread blocks like in Figure 2, as in [2].

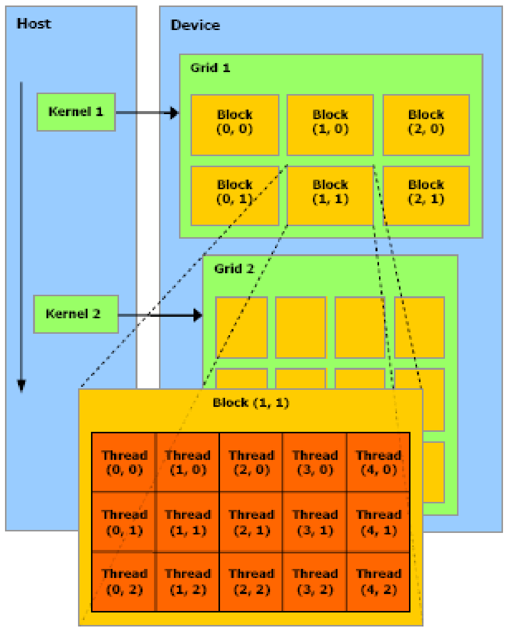

Figure 2. Thread batching

A thread block is a batch of threads that can cooperate together by efficiently sharing data through some fast shared memory and synchronizing their execution to coordinate memory accesses. Each thread is identified by its **thread ID**, which is the thread number within the block. An application can also specify a block as a two- or three-dimensional array of arbitrary size and identify each thread using a 2- or 3-component index instead. There is a limited maximum number of threads that a block can contain. The blocks of same dimensionality and size that execute the same kernel can be batched together into **a grid of blocks**. Threads in different thread blocks from the same grid cannot communicate and synchronize with each other. A device may run all the blocks of a grid sequentially if it has very few parallel capabilities, or in parallel if it has a lot of parallel capabilities, or usually a combination of both. Each block is identified by its **block ID**, which is the block number within the grid. An application can also specify a grid as a two-dimensional array of arbitrary size and identify each block using a 2-component index instead.A thread that executes on the device has only access to the device's DRAM and on-chip memory through the following memory spaces, as illustrated in Figure 3 (as described in [2]): (i) read-write per-thread registers, (ii) read-write per-thread local memory, (iii) read-write per-block shared memory, (iv) read-write per-grid global memory, (v) read-only per-grid constant memory, (vi) read-only per-grid texture memory.

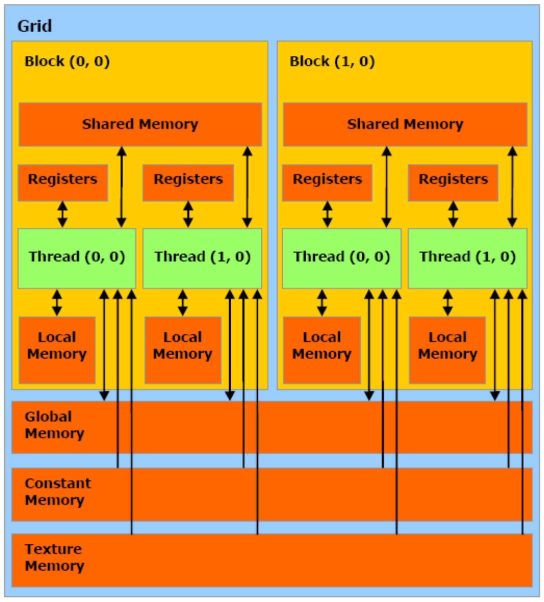

Figure 3. The Memory model

The device is implemented as a set of multiprocessors as illustrated in Figure 4, as described in [2]. Each multiprocessor has a Single Instruction, Multiple Data architecture (SIMD). At any given clock cycle, each processor of the multiprocessor executes the same instruction, but operates on different data. Each multiprocessor has on-chip memory of the four following types: (i) one set of local 32-bit registers per processor, (ii) a parallel data cache (shared memory) that is shared by all the processors and implements the shared memory space, (iii) a read-only constant cache that is shared by all the processors and speeds up reads from the constant memory space, which is implemented as a read-only region of device memory, (iv) a read-only texture cache that is shared by all the processors and speeds up reads from the texture memory space, which is implemented as a read-only region of device memory.

The local and global memory spaces are implemented as read-write regions of device memory and are not cached. Each multiprocessor accesses the texture cache via a texture unit that implements the various addressing modes and data filtering.

A grid of thread blocks is executed on the device by executing one or more blocks on each multiprocessor using time slicing: Each block is split into SIMD groups of threads called **warps**; each of these warps contains the same number of threads, called the **warp size**, and is executed by the multiprocessor in a SIMD fashion; **a thread scheduler** periodically switches from one warp to another to maximize the use of the multiprocessor’s computational resources. A block is processed by only one multiprocessor, so that the shared memory space resides in the on-chip shared memory leading to very fast memory accesses. The multiprocessor’s registers are allocated among the threads of the block. If the number of registers used per thread multiplied by the number of threads in the block is greater than the total number of registers per multiprocessor, the block cannot be executed and the corresponding kernel will fail to launch. Several blocks can be processed by the same multiprocessor concurrently by allocating the multiprocessor’s registers and shared memory among the blocks. The issue order of the warps within a block is undefined, but their execution can be synchronized.

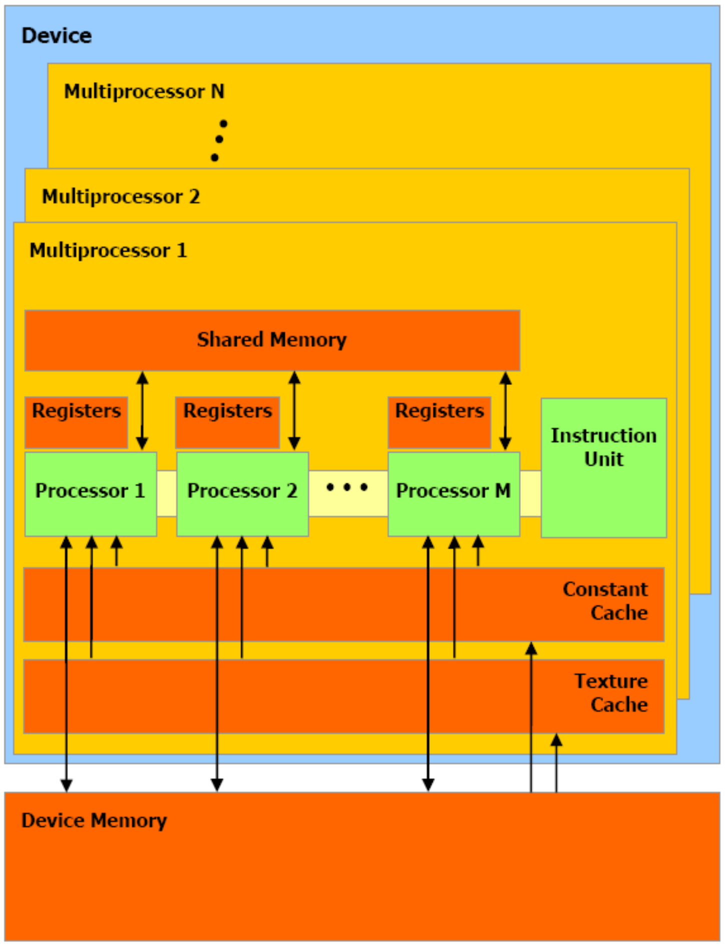

Figure 4. The hardware model

The CUDA application programming interface

The goal of the CUDA programming is to provide a relatively simple path for users familiar with the C. Based on [2], it consists of:

- A runtime library (presented in Table 1) split into:
    - A host component, that runs on the host and provides functions to control and access one or more compute devices from the host;
    - A device component, that runs on the device and provides device-specific functions;
    - A common component, that provides built-in vector types and a subset of the C standard library that are supported in both host and device code.
- A minimal set of extensions to the C language, that allow the programmer to target portions of the source code for execution on the device (composed of four parts presented in Table 2).

The host component	Multiple devices supported

	Memory: linear or CUDA arrays
	OpenGL and DirectX interoperability
	Asynchronicity: __global__ functions and most runtime functions return to the application before the device has completed the requested task
The device component	Synchronization function
	Type conversion
	Type casting
	Atomic functions (performs a read-modify-write operation on one 32 bit word residing in global memory)
A common component	Built-in Vector types (**float1, float2, int3, ushort4** etc)
	Constructor type creation: **int2 i = make int2(i, j)**
	Mathematical functions (standard **math.h** on CPU)
	Time function for benchmarking
	Texture references

Table 1 The runtime library

Function type qualifiers to specify whether a function executes on the host or on the device and whether it is callable from the host or from the device	**__device__** declares a function that is: (i) executed on the device, (ii) callable from the device only.
	__global__ declares a function as being a kernel: (i) executed on the device, (ii) callable from the host only.
	__host__ declares a function that is: (i) executed on the host, (ii) callable from the host only.
Variable type qualifiers to specify memory location on the device of a variable	**__device__** declares a variable that resides on the device: (i) resides in global memory space, (ii) has the lifetime of an application, (iii) it is accessible from all the threads within the the grid and from the host through the runtime library.
	__constant__ (optionally used with __device__), declares a variable that: (i) Resides in constant memory space, (ii) has the lifetime of an application, (iii) it is accessible from all the threads within the grid and from the host through the runtime library.
	__shared__ , (optionally used with __device__), declares a variable that: (i) resides in the shared memory space of a thread block, (ii) has the lifetime of the block, (iii) is only accessible from all the threads within the block.
A new directive to specify how a kernel is executed on the device from the host	● Any call to a __global__ function must specify the execution configuration for that call. ● The execution configuration defines the dimension of the grid and blocks that will be used to execute the function on the

	device. ● It is specified by the expression <<<Dg,Db,Ns>>> inserted between the function name and the parenthesized argument list, where: (i) **Dg** is of type **dim3** and specifies the dimension and size of the grid (Dg.x*Dg.y*Dg.y is the number of blocks being launched), (ii) **Db** is of type **dim3** and specifies the dimension and size of each block, (Db.x*Db.y*Db.z is the number of threads per block), (iii) **Ns** is of type **size_t** and specifies the number of bytes in shared memory that is dynamically allocated per block for this call in addition to the statically allocated memory.
Four built-in variables that specify the grid and block dimensions and the block and thread indices	**gridDim** is of type **dim3** and contains the dimensions of the grid.
	blockIdx is of type **uint3** and contains the block index within the grid.
	blockDim is of type **dim3** and contains the dimensions of the block.
	threadIdx is of type **uint3** and contains the thread index within the block.

Table 2. The set of extensions to the C language

A CUDA example

In the following it is presented a CUDA program for adding two matrix in parallel comparing with the same program write in C, which is described in [3]:

CPU Program	**Observations**	**CUDA Program**

void add_matrix
(float* a, float* b,
float* c, int N) {
int index;
 	
for (int i = 0; i <
N; i++)
for (int j = 0; j <
N; j++)
{
index = i + j*N;
c[index] =
a[index] +
b[index];
}
}
 	
int main() {
add_matrix(a, b,
c, N);
	The nested for-loops are replaced with an implicit grid	__global__ add_matrix
		(float* a, float* b, float* c, int N)
		{
		int i = blockIdx.x * blockDim.x +
		        threadIdx.x;
		int j = blockIdx.y * blockDim.y +
		        threadIdx.y;
		int index = i + j*N;
		
		if (i < N && j < N)
		c[index] = a[index] + b[index];
		}
		
		int main() {
		dim3 dimBlock(blocksize, blocksize);
		dim3 dimGrid(N/dimBlock.x,N/dimBlock.y);
		add_matrix<<<dimGrid,dimBlock>>>
		(a,b,c,N);
		}

}		

Bellow it is presented the whole source version of the above program, as pesented in [3]:

Program	Observations

const int N = 1024;
const int blocksize = 16;
	Set grid size

__global__
void add_matrix(float* a, float *b, float *c, int N)
{
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
int index = i + j*N;
if (i < N && j < N)
c[index] = a[index] + b[index];
}
	Compute kernel

int main() {	

float *a = new float[N*N];
float *b = new float[N*N];
float *c = new float[N*N];
for (int i = 0; i < N*N; ++i) a[i] = 1.0f; b[i] = 3.5f;
	CPU memory allocation

float *ad, *bd, *cd;
const int size = N*N*sizeof(float);
cudaMalloc((void**)&ad, size);
cudaMalloc((void**)&bd, size);
cudaMalloc((void**)&cd, size);
	GPU memory allocation

cudaMemcpy(ad, a, size, cudaMemcpyHostToDevice); cudaMemcpy(bd, b, size, cudaMemcpyHostToDevice);	Copy data to GPU

dim3 dimBlock(blocksize, blocksize);
dim3 dimGrid(N/dimBlock.x, N/dimBlock.y);
add_matrix<<<dimGrid, dimBlock>>>(d, bd, cd, N);
	Execute kernel

cudaMemcpy(c, cd, size, cudaMemcpyDeviceToHost);	Copy result back to CPU

cudaFree(ad); cudaFree(bd); cudaFree(cd);
delete[] a; delete[] b; delete[] c;
return EXIT_SUCCESS;
	Clean up and return

}	

Conclusions

CUDA has several advantages over traditional general purpose computation on GPUs (GPGPU) using graphics APIs:

- It uses the standard C language, with some simple extensions; it is no need to learn graphics API;

- Scattered writes – code can write to arbitrary addresses in memory;
- Shared memory – CUDA exposes a fast shared memory region that can be shared amongst threads;

But CUDA has some limitations:

- Recursive functions are not supported and must be converted to loops.
- Threads should be run in groups of at least 32 for best performance.
- CUDA-enabled GPUs are only available from Nvidia (GeForce 8 series and above, Quadro and Tesla)

Bibliography

1. http://www.nvidia.com/object/cuda_home.html
2. http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf
3. Johan Seland - Cuda Programming, Winter School in Parallel Computing,Geilo, January 20-25, 2008, (http://heim.ifi.uio.no/~knutm/geilo2008/seland.pdf)
4. http://www.gpgpu.org/sc2007/