Paralell Programing with CUDA

Bruno Cartaxo, Guilherme Cavalcanti

September 7, 2014

Abstract

CUDA is a parallel computing platform and programming model created by NVIDIA. It enables increases in computing performance by harnessing the power of the graphics processing unit (GPU). In this paper, we discuss CUDA's memory and programming models, CUDA programming basics, GPU architecture for computing and Tools. This work aims to present an introduction to CUDA programing, showing not only its programing and memory model, but also its architecture and related tools.

1 Introduction

The development of microprocessors design has been shifting to multi-core and many-core architectures. It is motivated by keeping up with the Moore's law even with the challenges faced by traditional paradigm of continuously increasing CPU frequency: physical limit of transistors size, power consumption, and heat dissipation. Consequently, it is expected that future generations of applications would exploit the parallelism offered by the multi-core architectures. Thus, parallel computing that has been traditionally associated with the high performance computing community, is now becoming a common sense for the mainstream computing due to the recent development of commodity multi-core and many-core architectures [8].

In this context, there are two approaches to deal with parallelism. The first, multi-core approach, integrates a few cores into a single chip, seeking to keep the execution speed of sequential programs. Laptops and desktops use this kind of processor. The second, many-core approach uses a large number of cores and is specially oriented to the execution throughput of parallel

programs. This approach is exemplified by the Graphical Processing Units (GPUs) [3].

The availability of General Purpose computation on GPUs (GPGPUs) aims to use the capabilities of multi-core CPUs and many-core GPUs together. A wise implementation that use those technologies can generate massive speedups [3] The challenge is to develop application software that transparently scales its parallelism to leverage the increasing number of processor cores, much as 3D graphics applications transparently scale their parallelism to manycore GPUs with widely varying numbers of cores [11].

The Compute Unified Device Architecture (CUDA) parallel programming model was designed to overcome this challenge while maintaining a low learning curve for programmers familiar with standard programming languages such as C. At its core are three key abstractions - a hierarchy of thread groups, shared memories, and barrier synchronization - that are simply exposed to the programmer as a minimal set of language extensions [11].

Therefore, this work aims to present an introduction to CUDA programing, showing not only its programing and memory model, but also its architecture and related tools. The section 2 presents applications of CUDA to real world problems, as well as raises some research topics related to energy consumption. Sections 3 and 4 introduce the CUDA programming model and memory hierarchy, respectively. Section 5 shows the CUDA programming basics by examples. Section 6 presents tools that aids CUDA development and section 7 dissects the GPU architecture for computing.

2 Applications of CUDA

According to NVIDIA, responsible for CUDA technology, it is possible to use CUDA for problems related to a variety of knowledge areas, such as: bioinformatics, computational chemistry, computational finance, computational fluid dynamics, computational structural mechanics, data science, defense, electronic design automation, imaging, computer vision, machine learning, medical imaging, numerical analytics and weather and climate [14]. Thus, this section present studies that applied CUDA to specific problems, as well as their main results. Additionally, some studies covering the important topic of energy consumption is showed at the end of this section.

Bakhoda et al. studied the CUDA performance with a variety of non-graphic applications, such as: cryptographic algorithms, graph processing,

molecular dynamics, electromagnetics calculus, monte carlo statistical simulation, pattern recognition with neural networks, and others [1]. Garland et al. also shared successful experiences using CUDA to parallelize applications of several domains [4].

Within algorithms and data structure domain, Kumar et al. modified a well-established algorithm to solve the classical problem of short path in a graph, aiming to make it parallel. For dense graphs, the solution using CUDA gained a speedup of 10 times over the previous proposed parallel solution that uses CPU threads [9]. Still about algorithms, Garland et al. presented experiments that showed enhancements on computing prefix sums of linked lists using CUDA. Their results showed an increase of scalability for lists between 1 and 256 millions of nodes [18]. Harris also executed experiments related to the prefix sums of linked lists and his implementation with CUDA presented a speedup of 6 times with lists with 16 million of elements, compared to the one that used CPU threads instead [5].

In image processing domain, Yang et al. implemented classical algorithms with CUDA, such as: histogram equalization, removing clouds, edge detection, and others. As the image size increased, histogram computation could get more than 40 times speedup, removing clouds could get about 79 times speedup, and edge detection more than 200 times [19]. Also about image processing, Datla e Gidijala detailed the experience of porting the motion JPEG 2000 encoder to the CUDA architecture. Theirs experiments demonstrated that the CUDA based implementation worked 20.7 times faster than the original implementation on the CPU [2].

Beyond the performance improvements for general purpose applications, some studies present the energy consumption profile of CUDA applications, as well as evidences of reduction of energy consumption. According to Sankaranarayanan et al.the first level data cache is critical to the performance of a GPGPU, but they are extremely energy inefficient due to the large number of cores they need to serve. In order to mitigate this problem they propose a solution that reduced 37% of energy consumption [16]. Furthermore, Ma et al. presented a framework that dynamically splits and distributes workloads to GPU and CPU, as well as throttles the frequencies of GPU cores and memory, based on their utilizations, for maximized energy savings with only marginal performance degradation. Experiment results showed that the framework achieved an average of 21.04% of energy savings [10].

3 CUDA Programming Model

The CUDA programming model is an extension to the C programming language, which makes it easy for programmers to offload computationally intensive computations to the GPU to take the advantage of its computational power. The CUDA programming model is based upon the concept of a kernel. [7]

A kernel is a function that is executed multiple times in parallel, each instance running in a separate thread. The threads are organized into one-, two- or three-dimensional blocks, which in turn are organized into one- or two-dimensional grids. The blocks are completely independent of each other and can be executed in any order. Threads within a block however are guaranteed to be run on a single multiprocessor. This makes it possible for them to synchronize and share information efficiently using the on-chip memory of the SM (streaming multiprocessor).

Because a GPU device contains multiple multiprocessors, multiple blocks and threads can be triggered and executed in parallel on the device. The ID of each thread can be calculated uniquely from the indexes of the block and the grid it belongs to. With respect to data sharing, each thread has data for its own usage, and threads within a block can share data amongst each other, thus allowing for data synchronization within a block. [6]

When a kernel is launched, threads in the same block will execute on the same multiprocessor. Thus, if there are too many threads in a block to fit inside the resources of a single multiprocessor, the kernel launch fails. Similarly, multiple blocks are grouped into a grid, but different blocks in a grid may or may not execute in parallel, depending on the number of multiprocessors on the GPU, the amount of shared memory they use, and the number of registers per multiprocessor.

An execution kernel can access memory in a number of ways. One is via the on-chip memory; each multiprocessor contains a set of 32-bit registers along with a shared memory region which is quickly accessible by any core on the multiprocessor, but hidden from other multiprocessors. There is also off-chip memory that contains both local memory and global memory. Compared to on-chip memory, the read latency of the off-chip memory is much higher, but its size is much larger. Besides the above two types, a multiprocessor contains two read-only caches, one for textures and the other for constants, to improve memory access to texture and constant memory, respectively. One simple example how threads can be organized is given on the Figure 1.

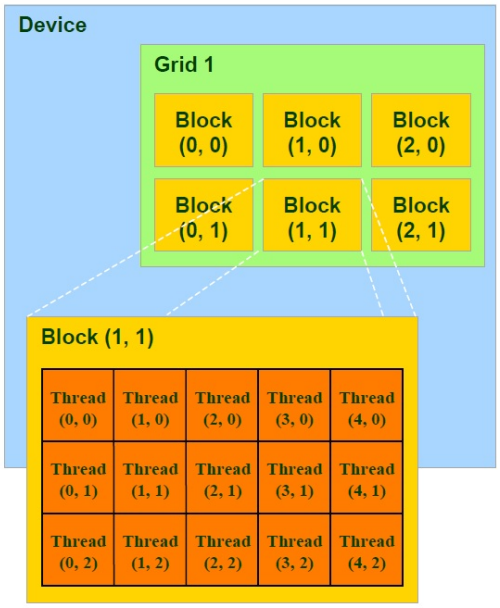

Figure 1: Thread's organization

4 CUDA Memory Hierarchy

This section presents the memory hierarchy of CUDA which is composed by some memory spaces with particular characteristics. Each thread has a local private memory. Each block of threads has a memory space shared by all threads of the block. All threads have access to a single global memory space [11]. In addition, CUDA has two read-only memory spaces accessible by all threads, they are: constant and texture memory spaces. All memory spaces have its trade-offs which should be taken into account when projecting a solution that uses CUDA kernels. For instance, the global memory has large space but high latency. Conversely, the shared memory has low latency but low storage capacity [17]. Figure 2 shows all memory spaces offered by CUDA architecture. Thus, the following items present detailed characteristics of the main memory spaces of CUDA.

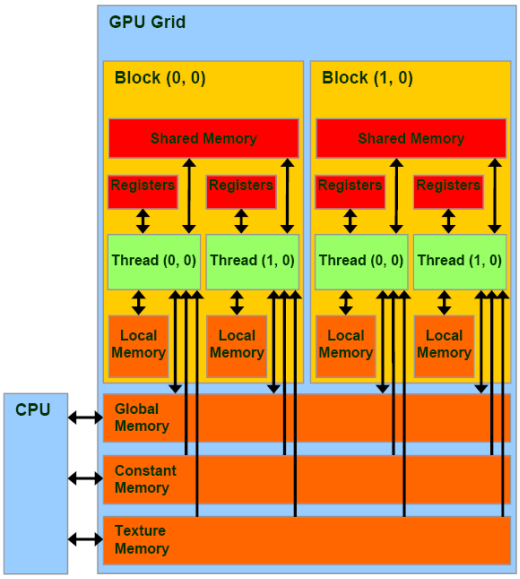

Figure 2: Memory Hierarchy of CUDA.

4.1 Registers

By default, variables declared in the scope of a kernel function and not decorated with any attribute are stored in register memory. This is the fastest type of memory, but its storage capacity is limited. Arrays declared in kernel functions are also stored in the registers, as long as accesses to their elements are made through constant indexes, so their values can be defined in compile-time. Register variables are private to the thread and exist only during the thread existence. This kind of variables can be both read and write and do not need any synchronization mechanism once they are private to each thread [17].

4.2 Local

Any variable that extrapolates the register space allowed for the kernel will be automatically allocated into local memory. This type of memory has high access latency, similar with the global one. Moreover, the local memory is private to each thread and consequently its existence is direct related its thread life-cycle. Like in the registers, variables allocated on local memory do not have any attribute decoration. The compiler will allocate them when

there are arrays accessed with variable indexes and structures exceeding the registers' capacity for the thread. Through manual verification in the PTX code (the one generated with -ptx compile option) is possible to verify if the compiler allocated variables in local memory or in the registers [17].

4.3 Shared

Variables decorated with the __shared__ attribute are stored in shared memory. They have low latency, approximately 100 times faster than the global memory. Shared memory must be declared within the scope of the kernel function but has a lifetime of the block, so its lifetime is extended to the existence of the block. Variables declared in the shared memory are shared by all threads in the block. Thus, access to those variables should use synchronization primitives like the __syncthreads() function [17].

4.4 Global

Variables decorated with the __constant__ attribute are allocated in constant memory. This memory space has high latency, approximately 100 times slower than the local memory, but offers large storage capacity, reaching 12GB depending on the graphic card. Variables allocated into global memory has a lifetime of the application and is accessible to all threads of all kernels. One should take care when reading from and writing to global memory since thread execution cannot be synchronized across different blocks. The only way to ensure access to global memory is synchronized is by invoking separate kernel invocations. Variables in global memory are declared on the host process using cudaMalloc and freed in the host process using cudaFree. Thus, pointers to global memory can be passed to a kernel function as parameters to the kernel. Unlike the registers, the local and the shared memory, the global memory can be read and written using the cudaMemcpy function [17].

4.5 Constant

Variables decorated with the __constant__ attribute are allocated in constant memory. They are declared outside of kernel functions and share the same memory banks as the global memory. However, the amount of memory reserved for constant variables is restricted with an approximately value of 64

KB. Variables at constant memory exist through the entire application duration. The access of constant memory is faster than global one, once there is a cache to that kind of variables. The cudaMemcpyToSymbol function is used to write in the constant memory and cudaMemcpyFromSymbol for write on it [17].

5 CUDA Programming Basics

This session introduces basic GPU programming using CUDA. Through CUDA, a developer can program GPUs using C, C++, Fortran, Java, Python, and more. We will talk about CUDA C/C++ based on the work by Sanders at al [15].

CUDA C/C++ is based on industry standard C/C++, a handful of language extensions to allow heterogeneous programs, and includes straightforward APIs to manage devices, memory, etc.

We will show how to start from a "Hello, World!", write and launch CUDA C kernels, manage GPU memory, run parallel kernels in CUDA C parallel communication and synchronization, race conditions and atomic operations.

The developer (probably) need experience with C or C++, but do not need any GPU experience, graphics experience or parallel programming experience.

5.1 Basic CUDA C Program

We would like to start with the following terminology: *host* - the CPU and its memory (host memory); *device* - the GPU and its memory (device memory).

Let's see a simple Hello World:

```
int main(void) {
        printf("Hello, World!\n");
        return 0;
}
```

This basic program is just standard C that runs on the host. The NVIDIA's compiler (nvcc) will not complain about CUDA programs with no device code. At its simplest, CUDA C is just C.

Now, look at the Hello, World! with device code:

```
__global__ void kernel(void) {
}

int main(void) {
        kernel<<<1,1>>>();
        printf ( "Hello, World!\n" );
        return 0;
}
```

Here, we have two notable additions to the original "Hello, World!": (1) CUDA C keyword `__global__`, which indicates that a function runs on the device and is called from host code. At this point `nvcc` splits source file into host and device components. The NVIDIA's compiler handles device functions like kernel() and the standard host compiler handles host functions like main() (e.g gcc, Microsoft Visual C). Finally, (2) the triple angle brackets mark a call from host code to device code, it is sometimes called a "kernel launch" (we'll discuss the parameters inside the angle brackets later). This is all that's required to execute a function on the GPU. For now, our function kernel() does nothing.

Let's look a more complex example, a simple kernel to add two integers:

```
__global__ void add(int *a, int *b, int *c ) {
        *c = *a + *b;
}
```

As before, `__global__` is a CUDA C keyword meaning that add() will execute on the device and `add()` will be called from the host. Notice that we use pointers for our variables,`add()` runs on the device, so a,b, and c must point to device memory. This leads to the following question: how do we allocate memory on the GPU?

In CUDA Memory Management, host and device memory are distinct entities, device pointers point to GPU memory and may be passed to and from host code, and may not be dereferenced from host code. Host pointers point to CPU memory and may be passed to and from device code and may not be dereferenced from device code also.

The basic CUDA API for dealing with device memory is `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`, all similar to their C equivalents, `malloc()`, `free()` and `memcpy()`.

Now, the following `main()` for the previous example:

```
int main( void ) {
       int a, b, c;                          // host copies of a, b,
           c
       int *dev_a, *dev_b, *dev_c; // device copies of a, b, c
       int size = sizeof(int);      //we need space for an integer

       // allocate device copies of a, b, c
       cudaMalloc( (void**)&dev_a, size );
       cudaMalloc( (void**)&dev_b, size );
       cudaMalloc( (void**)&dev_c, size );

       a = 2;
       b = 7;

       // copy inputs to device
       cudaMemcpy(dev_a, &a, size,cudaMemcpyHostToDevice);
       cudaMemcpy(dev_b, &b, size,cudaMemcpyHostToDevice);

       // launch add() kernel on GPU, passing parameters
       add<<< 1, 1 >>>(dev_a,dev_b,dev_c);

       // copy device result back to host copy of c
       cudaMemcpy(&c,dev_c, size,cudaMemcpyDeviceToHost);
       cudaFree(dev_a);
       cudaFree(dev_b);
       cudaFree(dev_c);
       return 0;
}
```

5.2 Parallel Programming in CUDA C

GPU computing is about massive parallelism, so the solution to run code in parallel on the device lies in the parameters between the triple angle brackets:

```
from add<<<1, 1>>>(dev_a,dev_b,dev_c); to
add<<<N, 1>>>(dev_a,dev_b,dev_c);
```

It means that, instead of executing add() once, add() is executed N times in parallel.

With this in mind, we will do a vector addition. Each parallel invocation of `add()`referred to as a block, Kernel can refer to its block's index with the variable `blockIdx.x` and each block adds a value from `a[]` and `b[]`, storing the result in `c[]`:

```
__global__ void add(int*a,int*b,int*c ) {
       c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

By using `blockIdx.x` to index arrays, each block handles different indices. This is what runs in parallel on the device:

```
Block 0
c[0] = a[0] + b[0];
Block 1
c[1] = a[1] + b[1];
Block 2
c[2] = a[2] + b[2];
Block 3
c[3] = a[3] + b[3];
```

Now, look the following `main()` for the previous example:

```
#define N 512
int main( void ) {
       int *a,*b,*c;                          // host copies of a, b,
           c
       int*dev_a, *dev_b, *dev_c;   // device copies of a, b, c
       int size =N *sizeof(int);    // we need space for 512
           integers

       // allocate device copies of a, b, c
       cudaMalloc( (void**)&dev_a, size );
       cudaMalloc( (void**)&dev_b, size );
       cudaMalloc( (void**)&dev_c, size );

       a = (int*)malloc( size );
       b = (int*)malloc( size );
       c = (int*)malloc( size );
```

```
        random_ints( a, N );
        random_ints( b, N );

        // copy inputs to device
        cudaMemcpy(dev_a, a, size,cudaMemcpyHostToDevice);
        cudaMemcpy(dev_b, b, size,cudaMemcpyHostToDevice);

        // launch add() kernel with N parallel blocks
        add<<<N, 1 >>>(dev_a,dev_b,dev_c);

        // copy device result back to host copy of c
        cudaMemcpy( c,dev_c, size,cudaMemcpyDeviceToHost);

        free( a ); free( b ); free( c );
        cudaFree(dev_a);
        cudaFree(dev_b);
        cudaFree(dev_c);
        return 0;
}
```

A block can even be split into parallel threads. Let's change vector addition to use parallel threads instead of parallel blocks:

```
__global__ void add(int*a,int*b,int*c ) {
        c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

We use `threadIdx.x` instead of `blockIdx.x` in `add()`, `main()` will require one change as well:

```
int main( void ) {
...
        // launch add() kernel with N parallel blocks
        add<<<1, N>>>(dev_a,dev_b,dev_c);
...
}
```

We've seen parallel vector addition using many blocks with 1 thread apiece and 1 block with many threads. Let's adapt vector addition to use lots

of both blocks and threads. But, before we want to discuss data indexing.

To index array with 1 thread per entry (using 8 threads/block) :

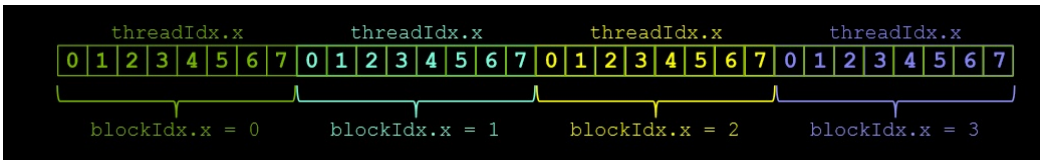

Figure 3: Indexing array

If we have M threads/block, a unique array index for each entry is given by:

```
from int index = threadIdx.x + blockIdx.x * M;
to int index = x + y * width;
```

Now, in the following example, the red entry (Figure 4) would have an index of 21:

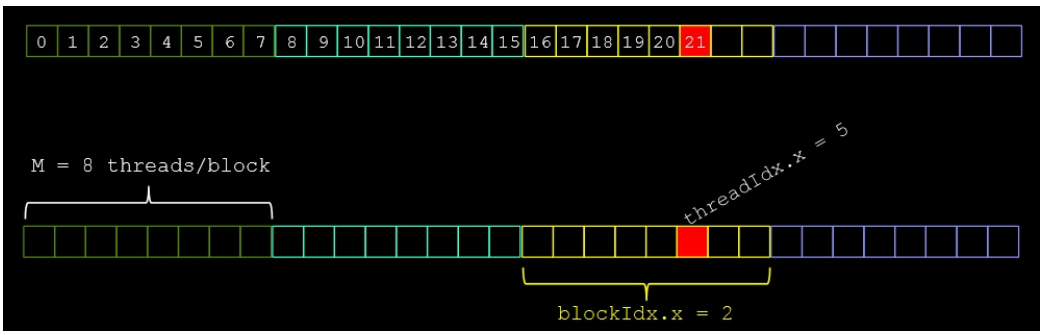

Figure 4: Calculating entry

```
int index     = threadIdx.x + blockIdx.x * M;
              = 5 + 2 * 8;
              = 21;
```

There is also a a built-in variable for threads per block, the `blockDim.x` variable:

```
int index= threadIdx.x + blockIdx.x * blockDim.x ;
```

Finally, a combined version of our vector addition kernel to use blocks and threads is:

```
__global__ void add(int*a,int*b,int*c ) {
        int index = threadIdx.x + blockIdx.x * blockDim.x;
        c[index] = a[index] + b[index];
}
```

And the main():

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main( void ) {
        int*a, *b, *c;                          // host copies of a, b,
            c
        int*dev_a, *dev_b, *dev_c;   // device copies of a, b, c
        int size = N *sizeof(int);   // we need space for N integers

        // allocate device copies of a, b, c
        cudaMalloc( (void**)&dev_a, size );
        cudaMalloc( (void**)&dev_b, size );
        cudaMalloc( (void**)&dev_c, size );

        a = (int*)malloc( size );
        b = (int*)malloc( size );
        c = (int*)malloc( size );

        random_ints( a, N );
        random_ints( b, N );

        // copy inputs to device
        cudaMemcpy(dev_a, a, size,cudaMemcpyHostToDevice);
        cudaMemcpy(dev_b, b, size,cudaMemcpyHostToDevice);

        // launch add() kernel with blocks and threads
        add<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(dev_a,dev_b,dev_c);

        // copy device result back to host copy of c
        cudaMemcpy( c,dev_c, size,cudaMemcpyDeviceToHost);

        free( a ); free( b ); free( c );
```

```
        cudaFree(dev_a);
        cudaFree(dev_b);
        cudaFree(dev_c);
        return 0;
}
```

Threads seem unnecessary, they added a level of abstraction and complexity, but unlike parallel blocks, parallel threads have mechanisms to communicate and and synchronize. But how?

For this purpose, consider a Dot Product example, unlike vector addition, dot product is a reduction from vectors to a scalar:

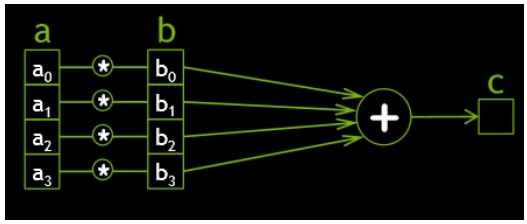

Figure 5: Dot Product

Parallel threads have no problem computing the pairwise products, so we can start a dot product CUDA kernel by doing just that:

```
__global__ void dot( int*a,int*b,int*c ) {
        // Each thread computes a pairwise product
        int temp = a[threadIdx.x] * b[threadIdx.x];

        // Can't compute the final sum
        // Each thread's copy of 'temp' is private
}
```

But, as seen in the example, we need to share data between threads to compute the final sum.

A block of threads shares memory, the so-called shared memory, they are extremely fast, on chip memory (user managed cache), they are declared with the `__shared__` CUDA keyword, and are not visible to threads in other blocks running in parallel.

Finally, we perform parallel multiplication, serial addition:

```
#define N 512
__global__ void dot(int*a,int*b,int*c ) {
        // Shared memory for results of multiplication
        __shared__ int temp[N];
        temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

        // Thread 0 sums the pairwise products
        if( 0 == threadIdx.x ) {
                int sum = 0;
                for(int i= 0;i< N;i++ )
                        sum += temp[i];
                *c = sum;
        }
}
```

Recapitulating, in the Parallel Dot Product example, we perform parallel, pairwise multiplications using shared memory to store each thread's result and we sum these pairwise products from a single thread. It sounds good, but we've made a mistake.

There are two steps, (1) in parallel, each thread writes a pairwise product (2) Thread 0 reads and sums the products. Now, suppose thread 0 finishes its write in (1), then thread 0 reads index 12 in (2) before thread 12 writes to index 12 in step 1, the thread 0's read returns garbage.

Suppose thread 0 finishes its write in step 1, then thread 0 reads index 12 in step 2 before thread 12 writes to index 12 in step 1, the thread 0's read returns garbage!

We need threads to wait between the sections of dot():

```
__global__ void dot(int*a,int*b,int*c ) {
        // Shared memory for results of multiplication
        __shared__ int temp[N];
        temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

        // * NEED THREADS TO SYNCHRONIZE HERE *
        // No thread can advance until all threads
        // have reached this point in the code

        // Thread 0 sums the pairwise products
        if( 0 == threadIdx.x ) {
                int sum = 0;
```

```
                for(int i= 0;i< N;i++ )
                        sum += temp[i];
                *c = sum;
        }
}
```

We can synchronize threads with the function __syncthreads(), where threads in the block wait until all threads have hit the __syncthreads(). Threads are only synchronized within a block. Finally:

```
#define N 512
__global__ void dot(int*a,int*b,int*c ) {
        // Shared memory for results of multiplication
        __shared__ int temp[N];
        temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

        __syncthreads();

        // Thread 0 sums the pairwise products
        if( 0 == threadIdx.x ) {
                int sum = 0;
                for(int i= 0;i< N;i++ )
                        sum += temp[i];
                *c = sum;
        }
}
```

With this properly synchronized `dot()` routine, we have the following `main()`:

```
#define N 512
int main( void ) {
        int *a,*b,*c;                       // host copies of a, b,
           c
        int*dev_a, *dev_b, *dev_c;   // device copies of a, b, c
        int size =N *sizeof(int);    // we need space for 512
           integers

        // allocate device copies of a, b, c
        cudaMalloc( (void**)&dev_a, size );
```

```
        cudaMalloc( (void**)&dev_b, size );
        cudaMalloc( (void**)&dev_c, sizeof(int) );

        a = (int*)malloc( size );
        b = (int*)malloc( size );
        c = (int*)malloc( sizeof(int));

        random_ints( a, N );
        random_ints( b, N );

        // copy inputs to device
        cudaMemcpy(dev_a, a, size,cudaMemcpyHostToDevice);
        cudaMemcpy(dev_b, b, size,cudaMemcpyHostToDevice);

        // launch dot() kernel with 1 block and N threads
        dot<<<1, N>>>(dev_a,dev_b,dev_c);

        // copy device result back to host copy of c
        cudaMemcpy( c,dev_c, sizeof(int),cudaMemcpyDeviceToHost);

        free( a ); free( b ); free( c );
        cudaFree(dev_a);
        cudaFree(dev_b);
        cudaFree(dev_c);
        return 0;
}
```

Summarizing, we launched `add()` with N threads: `add<<< 1, N >>>()`, used `threadIdx.x` to access thread's index, `(threadIdx.x + blockIdx.x * blockDim.x)` to index input/output; besides `N/THREADS_PER_BLOCK` blocks and `THREADS_PER_BLOCK` threads gave us N threads total. We used `__shared__` to declare memory as shared memory which shared data among threads in a block and that is not visible to threads in other parallel blocks. Finally, we used `__syncthreads()` as a barrier, telling that no thread executes instructions after `__syncthreads()` until all threads have reached the `__syncthreads()`, and which is necessary to prevent data hazards.

Our dot product uses one block, launching with one block will not utilize much of the GPU, it would be great a multiblock version of dot product, where each block computes a sum of its pairwise products like before and

then contributes its sum to the final result:

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
__global__ void dot(int*a,int*b,int*c ) {
        __shared__ int temp[THREADS_PER_BLOCK];
        int index = threadIdx.x + blockIdx.x * blockDim.x;
        temp[threadIdx.x] = a[index] * b[index];

        __syncthreads();

        if( 0 == threadIdx.x ) {
                int sum = 0;
                for(int i= 0;i<THREADS_PER_BLOCK;i++ )
                        sum += temp[i];
                *c += sum;
        }
}
```

But we have a race condition. A *race condition* occurs when program behavior depends upon relative timing of two (or more) event sequences. What actually takes place to execute the line in question, `*c += sum`, is (1) read value at address c, (2) add sum to value and (3) write result to address c (a *Read-Modify-Write* operation), which can rise unexpected results if two threads are trying to do this at the same time.

Read-modify-write is uninterruptible when *atomic*. Many atomic operations on memory are available with CUDA C: `atomicAdd(),atomicSub(),atomicMin(),atomicMax(),atomicInc(), atomicDec(),atomicExch() and atomicCAS()`, which gave us predictable result when simultaneous access to memory is required. So, we need to atomically add `sum` to `c` in our multiblock dot product:

```
__global__ void dot(int*a,int*b,int*c ) {
...
                for(int i= 0;i<THREADS_PER_BLOCK;i++ )
                        sum += temp[i];
                atomicAdd(c , sum );
        }
}
```

Finally, it leads to the following modification in the `main()`:

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main( void ) {
...
        // launch dot() kernel
        dot<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(dev_a,dev_b,dev_c);
...

}
```

The compilation process can be summarized by the following Figure 6:

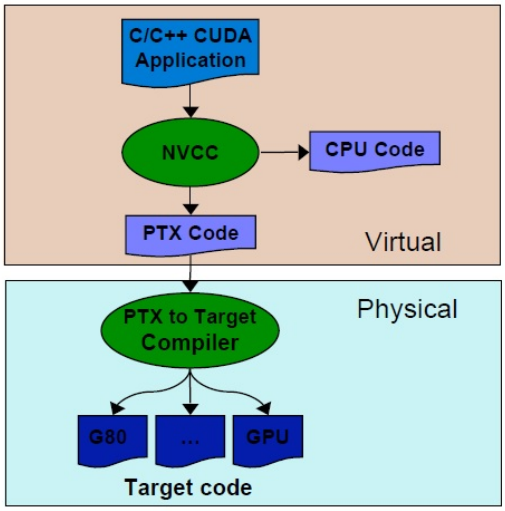

Figure 6: Compilation Process

6 CUDA Development Tools

This section describes the NVIDIA profiling tools and the CUDA debugging tool, some of the tools offered to the developer to better enjoy the CUDA environment.

6.1 Profiling Tools

The NVIDIA profiling tools and APIs enable the developers to understand and optimize the performance of their CUDA application. [13]

The *Visual Profiler* is a graphical profiling tool that displays a timeline of the application's CPU and GPU activity, and that includes an automated analysis engine to identify optimization opportunities. The Visual Profiler displays a timeline of the application's activity on both the CPU and GPU so that the developer can identify opportunities for performance improvement. In addition, the Visual Profiler will analyze the application to detect potential performance bottlenecks and direct you on how to take action to eliminate or reduce those bottlenecks.

There is also a *Command Line Profiler*, which that can be used to measure performance and find potential opportunities for optimization for CUDA applications executing on NVIDIA GPUs. The command line profiler allows users to gather timing information about kernel execution and memory transfer operations. Profiling options are controlled through environment variables and a profiler configuration file. Profiler output is generated in text files either in Key-Value-Pair (KVP) or Comma Separated (CSV) format.

The CUDA profiling tools do not require any application changes to enable profiling; however, by making some simple modifications and additions, the user can greatly increase the usability and effectiveness of the profilers.

By default, the profiling tools collect profile data over the entire run of the application. But, as explained below, the user typically only want to profile the region(s) of the application containing some or all of the performance-critical code. Limiting profiling to performance-critical regions reduces the amount of profile data that both the user and the tools must process, and focuses attention on the code where optimization will result in the greatest performance gains.

There are several common situations where profiling a region of the application is helpful: (1) the application is a test harness that contains a CUDA implementation of all or part of the algorithm. The test harness initializes the data, invokes the CUDA functions to perform the algorithm, and then checks the results for correctness. Using a test harness is a common and productive way to quickly iterate and test algorithm changes. When profiling, the user wants to collect profile data for the CUDA functions implementing the algorithm, but not for the test harness code that initializes the data or checks the results. (2) The application operates in phases, where a different

set of algorithms is active in each phase. When the performance of each phase of the application can be optimized independently of the others, the developer want to profile each phase separately to focus his optimization efforts. (3) The application contains algorithms that operate over a large number of iterations, but the performance of the algorithm does not vary significantly across those iterations. In this case the user can collect profile data from a subset of the iterations.

To limit profiling to a region of an application, CUDA provides functions to start and stop profile data collection, `cudaProfilerStart()` is used to start profiling and `cudaProfilerStop()` is used to stop profiling. To use these functions the developer must include `cuda_profiler_api.h`

6.2 The NVIDIA CUDA Debugger

CUDA-GDB: The NVIDIA CUDA Debugger is a ported version of GDB: The GNU Debugg er, version 6.6. The goal of its design is to present the user with an all-in-one debugging environment that is capable of debugging native host code as well as CUDA code. Therefore, it is an extension to the standard i386 port that is provided in the GDB release. As a result, standard debugging features are inherently supported for host code, and additional features have been provided to support debugging CUDA code.[12]

The goal of CUDA-GDB is to provide developers a mechanism of debugging a CUDA application on actual hardware in real time. Therefore, a user will be able to verify program correctness without variations often introduced by simulation and/or emulation environments.

Just as the CUDA programming model provides a seamless mechanism for programming host and GPU code, CUDA-GDB provides a model to seamlessly debug both host and GPU code. Therefore, GPU memory is treated as an extension to host memory, and GPU threads/blocks are treated as extensions to host th reads. Furthermore, there is no difference between CUDA-GDB and GDB when debugging host code.

CUDA-GDB supports setting breakpoints at any host and/or device function residing in the CUDA application by using the function symbol name and/or the source file line number. This can be accomplished in the same way for either host or device code

CUDA-GDB supports stepping GPU code at the finest granularity of a warp. This means that typing next or step from the CUDA-GDB command line (when in the focus of device code) will advance all threads in the same

warp as the current thread of focus. In order to advance the execution of more than one warp, the developer must set a breakpoint at the desired location. A special case is stepping the thread barrier call, `__syncthreads()`. In this case, an implicit breakpoint is set immediately after the barrier and all threads are continued to this point. An important note is that it is not currently possible to step over a subroutine. Since all subroutines are implicitly inlined, CUDA-GDB will always step into a subroutine

The GDB `print` command has been extended to decipher the location of any program variable, and can be used to display the contents of any CUDA program variable

CUDA-GDB provides an additional command (`info cuda threads`) which displays a summary of all CUDA threads that are currently resident on the GPU. Since this summary only shows thread coordinates for the start and end range, it may be unclear how many threads or blocks are actually within the displayed range. This can be checked by printing the values of gridDim and/or blockDim. CUDA-GDB also has the ability to display a full list of each individual thread that is currently resident on the GPU by using the info cuda threads all command.

CUDA-GDB also provides an additional command (`info cuda state`) which displays information such as the current hardware being used and memory that has been allocated via `cudaMalloc()`.

CUDA-GDB provides support for debugging kernels that appear to be hanging or looping indefinitely. The Ctrl-C signal will freeze the GPU and report back the source code location. At this point, the program can be modified and then resumed or terminated at the user's discretion.

Finally, CUDA-GDB supports an initialization file, which must reside in the home directory (~/.cuda-gdbinit). This file can take any CUDA-GDB command/extension as input to be processed upon executing the cuda-gdb command. This is just like the .gdbinit file used by standard versions of GDB, only renamed.

CUDA also provides an emulation mode, that allows to compile and execute in emulation on CPU and allows CPU-style debugging in GPU source.

7 GPU Architecture for Computing

NVIDIA GPU Computing Architecture is a scalable parallel computing platform present in laptops, desktops, workstations, servers.

Using as example the NVIDIA 8-Series family (Figure 7), 8-series GPUs deliver 50 to 200 GFLOPS on compiled parallel C applications. The GPU parallel performance is pulled by the insatiable demands of PC game market and GPU parallelism is doubling every year with a programming model scaling transparently.

The NVIDIA 8-Series GPU Computing is a massively multithreaded parallel computing platform with 12,288 concurrent threads, hardware managed and 128 Thread Processor cores at 1.35 GHz, wich gives 518 GFLOPS peak.

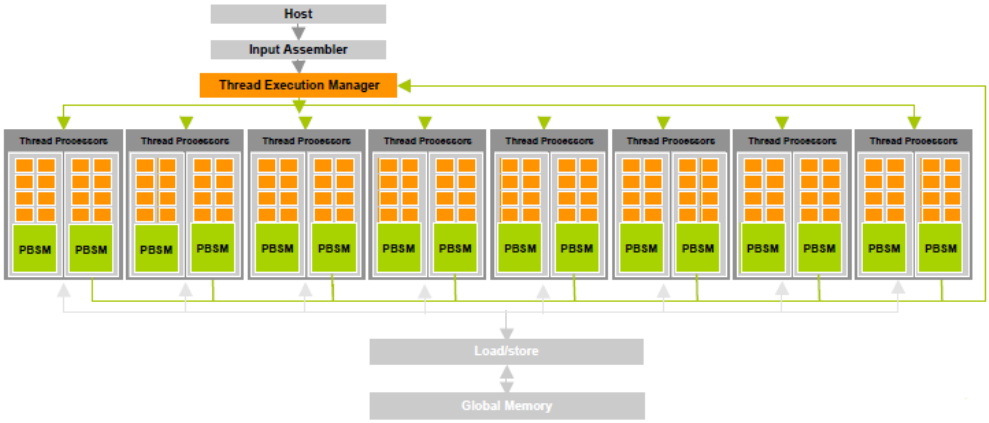

Figure 7: NVIDIA 8-Series Architecture

There is also the so-called *Streaming Multiprocessor* (SM) (Figure 8), whose processing elements are 8 scalar thread processors(SP), with 32 GFLOPS peak at 1.35 GHz, 8192 32-bit registers(32KB), usually operating float, int and branch. Its hardware multithreading is up to 8 blocks resident at once and up to 768 active threads in total. Finally, there is a 16KB on-chip memory with low latency storage, shared among threads of a block, and supporting thread communication.

Concerning the Hardware Multithreading, (1) hardware allocates resources to blocks, which need thread slots, registers, shared memory and don't run until resources are available; (2) Hardware schedules threads, which have their own registers, any thread not waiting for something can run and the context switching is free - every cycle; (3) hardware relies on threads to hide latency - *i.e.*, parallelism is necessary for performance.

NVIDIA call their parallel programming model as *SIMT* - "Single In-

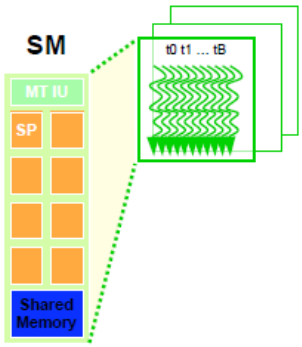

Figure 8: Streaming Multiprocessor

struction, Multiple Threads". It has groups of 32 threads formed into *warps*, which is always executing the same instruction. The warps share instruction fetch/dispatch, and some become inactive when code path diverges, with the hardware automatically handling the divergence. Warps are the primitive unit of scheduling.

SIMT execution is an implementation choice: sharing control logic leaves more space for Arithmetic Logic Units(ALUs), it is largely invisible to programmer and must be understood for performance, not correctness.

Finally, blocks run on multiprocessors (see Figure 9).

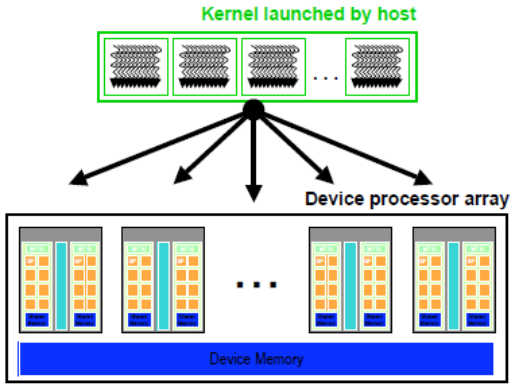

Figure 9: Blocks Distribution

References

[1] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174, April 2009.

[2] S. Datla and N. Gidijala. Parallelizing motion jpeg 2000 with cuda. In *Computer and Electrical Engineering, 2009. ICCEE '09. Second International Conference on*, volume 1, pages 630–634, Dec 2009.

[3] J. Diaz, C. Munoz-Caro, and A. Nino. A survey of parallel programming models and tools in the multi and many-core era. *Parallel and Distributed Systems, IEEE Transactions on*, 23(8):1369–1386, Aug 2012.

[4] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel computing experiences with cuda. *Micro, IEEE*, 28(4):13–27, July 2008.

[5] M. Harris. Parallel prefix sum (scan) with cuda, Apr 2007.

[6] S. Huang, S. Xiao, and W.-c. Feng. On the energy efficiency of graphics processing units for scientific computing. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[7] M. Januszewski and M. Kostur. Accelerating numerical solution of stochastic differential equations with cuda. *Computer Physics Communications*, 181(1):183–188, 2010.

[8] H. Kasim, V. March, R. Zhang, and S. See. Survey on parallel programming model. In J. Cao, M. Li, M.-Y. Wu, and J. Chen, editors, *Network and Parallel Computing*, volume 5245 of *Lecture Notes in Computer Science*, pages 266–275. Springer Berlin Heidelberg, 2008.

[9] S. Kumar, A. Misra, and R. Tomar. A modified parallel approach to single source shortest path problem for massively dense graphs using cuda. In *Computer and Communication Technology (ICCCT), 2011 2nd International Conference on*, pages 635–639, Sept 2011.

[10] K. Ma, X. Li, W. Chen, C. Zhang, and X. Wang. Greengpu: A holistic approach to energy efficiency in gpu-cpu heterogeneous architectures. In *Parallel Processing (ICPP), 2012 41st International Conference on*, pages 48–57, Sept 2012.

[11] C. NVIDIA. Cuda c programming guide.

[12] C. NVIDIA. Cuda-gdb: The nvidia cuda debugger.

[13] C. NVIDIA. Cuda profile users guide.

[14] C. NVIDIA. Gpu applications.

[15] J. Sanders and E. Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[16] A. Sankaranarayanan, E. Ardestani, J. Briz, and J. Renau. An energy efficient gpgpu memory hierarchy with tiny incoherent caches. In *Low Power Electronics and Design (ISLPED), 2013 IEEE International Symposium on*, pages 9–14, Sept 2013.

[17] J. van Oosten. Cuda memory model.

[18] Z. Wei and J. Jaja. Optimization of linked list prefix computations on multithreaded gpus using cuda. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–8, April 2010.

[19] Z. Yang, Y. Zhu, and Y. Pu. Parallel image processing based on cuda. In *Computer Science and Software Engineering, 2008 International Conference on*, volume 3, pages 198–201, Dec 2008.