Chapter 1. The Benefits of Using GPUs

The Graphics Processing Unit (GPU)[1] provides much higher instruction throughput and memory bandwidth than the CPU within a similar price and power envelope. Many applications leverage these higher capabilities to run faster on the GPU than on the CPU (see GPU Applications). Other computing devices, like FPGAs, are also very energy efficient, but offer much less programming flexibility than GPUs.

This difference in capabilities between the GPU and the CPU exists because they are designed with different goals in mind. While the CPU is designed to excel at executing a sequence of operations, called a *thread*, as fast as possible and can execute a few tens of these threads in parallel, the GPU is designed to excel at executing thousands of them in parallel (amortizing the slower single-thread performance to achieve greater throughput).

The GPU is specialized for highly parallel computations and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control. The schematic Figure 1 shows an example distribution of chip resources for a CPU versus a GPU.

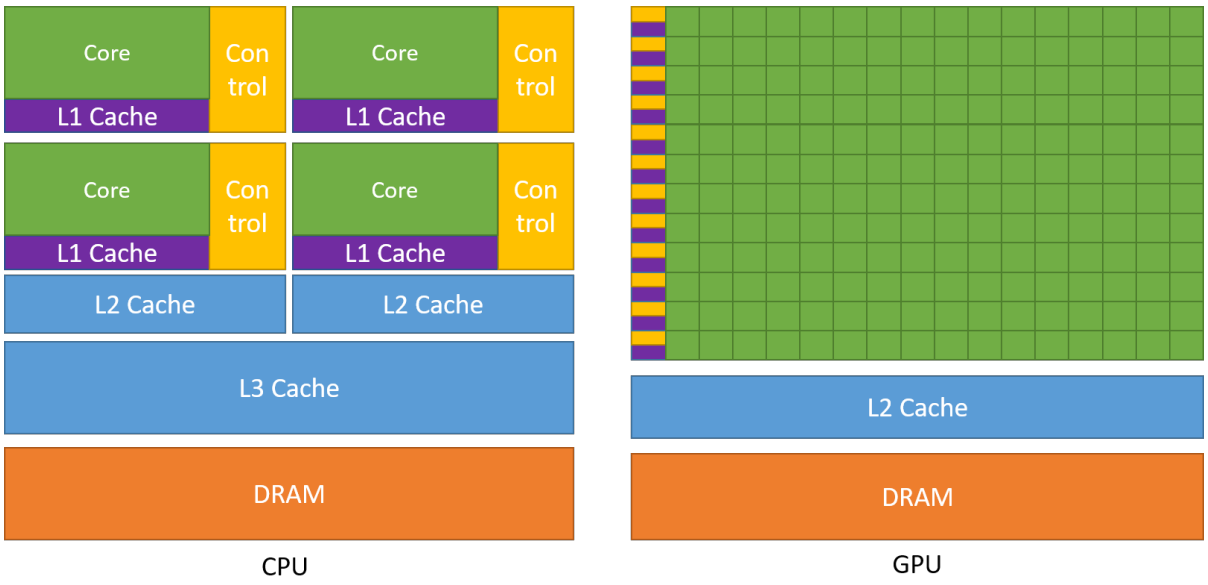

Figure 1: The GPU Devotes More Transistors to Data Processing

Devoting more transistors to data processing, for example, floating-point computations, is beneficial for highly parallel computations; the GPU can hide memory access latencies with computation, instead

[1] The *graphics* qualifier comes from the fact that when the GPU was originally created, two decades ago, it was designed as a specialized processor to accelerate graphics rendering. Driven by the insatiable market demand for real-time, high-definition, 3D graphics, it has evolved into a general processor used for many more workloads than just graphics rendering.

of relying on large data caches and complex flow control to avoid long memory access latencies, both of which are expensive in terms of transistors.

In general, an application has a mix of parallel parts and sequential parts, so systems are designed with a mix of GPUs and CPUs in order to maximize overall performance. Applications with a high degree of parallelism can exploit this massively parallel nature of the GPU to achieve higher performance than on the CPU.

Chapter 2. CUDA®: A General-Purpose Parallel Computing Platform and Programming Model

In November 2006, NVIDIA® introduced CUDA®, a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU.

CUDA comes with a software environment that allows developers to use C++ as a high-level programming language. As illustrated by Figure 2, other languages, application programming interfaces, or directives-based approaches are supported, such as FORTRAN, DirectCompute, OpenACC.

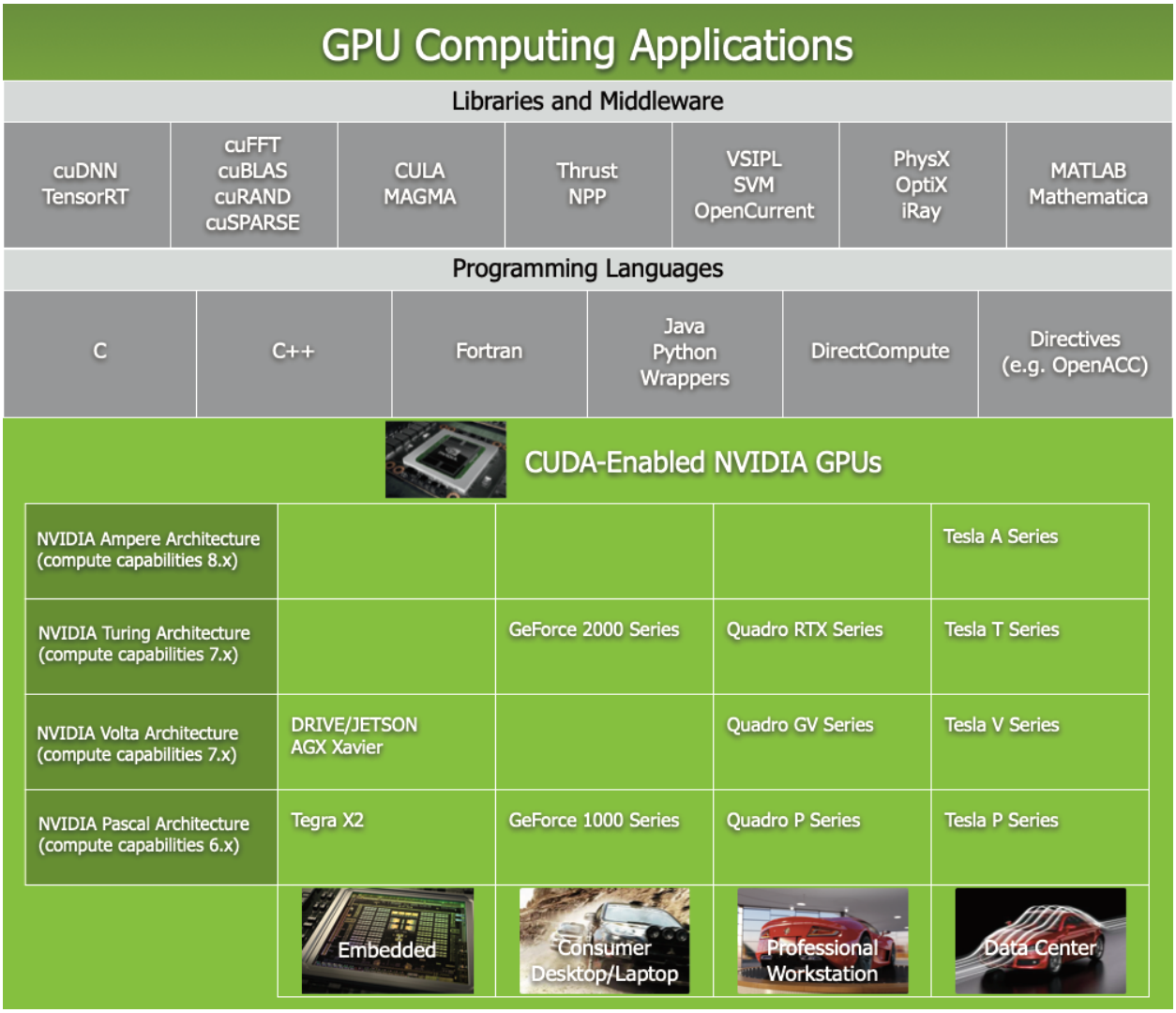

Figure 2: GPU Computing Applications. CUDA is designed to support various languages and application programming interfaces.

Chapter 3. A Scalable Programming Model

The advent of multicore CPUs and manycore GPUs means that mainstream processor chips are now parallel systems. 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.

The CUDA parallel programming model is 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.

These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. They guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block.

This decomposition preserves language expressivity by allowing threads to cooperate when solving each sub-problem, and at the same time enables automatic scalability. Indeed, each block of threads can be scheduled on any of the available multiprocessors within a GPU, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on any number of multiprocessors as illustrated by Figure 3, and only the runtime system needs to know the physical multiprocessor count.

This scalable programming model allows the GPU architecture to span a wide market range by simply scaling the number of multiprocessors and memory partitions: from the high-performance enthusiast GeForce GPUs and professional Quadro and Tesla computing products to a variety of inexpensive, mainstream GeForce GPUs (see CUDA-Enabled GPUs for a list of all CUDA-enabled GPUs).

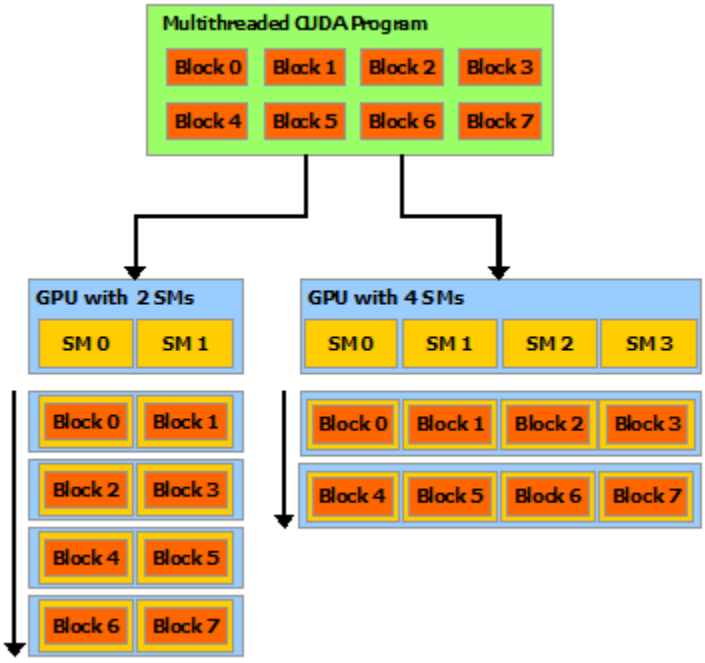

Figure 3: Automatic Scalability

Note: A GPU is built around an array of Streaming Multiprocessors (SMs) (see Hardware Implementation for more details). A multithreaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors.

Chapter 5. Programming Model

This chapter introduces the main concepts behind the CUDA programming model by outlining how they are exposed in C++.

An extensive description of CUDA C++ is given in Programming Interface.

Full code for the vector addition example used in this chapter and the next can be found in the vectorAdd CUDA sample.

5.1. Kernels

CUDA C++ extends C++ by allowing the programmer to define C++ functions, called *kernels*, that, when called, are executed N times in parallel by N different *CUDA threads*, as opposed to only once like regular C++ functions.

A kernel is defined using the `__global__` declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using a new `<<<...>>>` *execution configuration* syntax (see C++ Language Extensions). Each thread that executes the kernel is given a unique *thread ID* that is accessible within the kernel through built-in variables.

As an illustration, the following sample code, using the built-in variable `threadIdx`, adds two vectors *A* and *B* of size *N* and stores the result into vector *C*:

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Here, each of the *N* threads that execute `VecAdd()` performs one pair-wise addition.

5.2. Thread Hierarchy

For convenience, `threadIdx` is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional *thread index*, forming a one-dimensional, two-dimensional, or three-dimensional block of threads, called a *thread block*. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume.

The index of a thread and its thread ID relate to each other in a straightforward way: For a one-dimensional block, they are the same; for a two-dimensional block of size *(Dx, Dy)*, the thread ID of a thread of index *(x, y)* is *(x + y Dx)*; for a three-dimensional block of size *(Dx, Dy, Dz)*, the thread ID of a thread of index *(x, y, z)* is *(x + y Dx + z Dx Dy)*.

As an example, the following code adds two matrices *A* and *B* of size *NxN* and stores the result into matrix *C*:

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same streaming multiprocessor core and must share the limited memory resources of that core. On current GPUs, a thread block may contain up to 1024 threads.

However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks.

Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional *grid* of thread blocks as illustrated by Figure 4. The number of thread blocks in a grid is usually dictated by the size of the data being processed, which typically exceeds the number of processors in the system.

The number of threads per block and the number of blocks per grid specified in the `<<<...>>>` syntax can be of type `int` or `dim3`. Two-dimensional blocks or grids can be specified as in the example above.

Each block within the grid can be identified by a one-dimensional, two-dimensional, or three-dimensional unique index accessible within the kernel through the built-in `blockIdx` variable. The dimension of the thread block is accessible within the kernel through the built-in `blockDim` variable.

Extending the previous `MatAdd()` example to handle multiple blocks, the code becomes as follows.

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
```

(continues on next page)

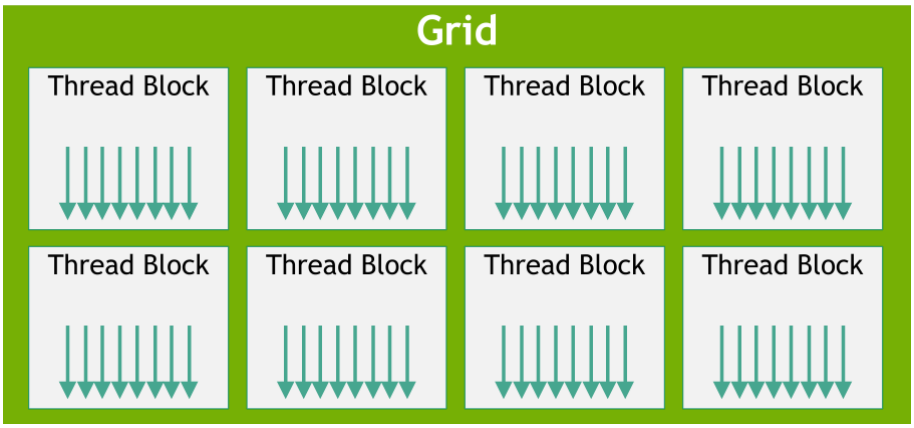

Figure 4: Grid of Thread Blocks

(continued from previous page)

```
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

A thread block size of 16x16 (256 threads), although arbitrary in this case, is a common choice. The grid is created with enough blocks to have one thread per matrix element as before. For simplicity, this example assumes that the number of threads per grid in each dimension is evenly divisible by the number of threads per block in that dimension, although that need not be the case.

Thread blocks are required to execute independently: It must be possible to execute them in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order across any number of cores as illustrated by Figure 3, enabling programmers to write code that scales with the number of cores.

Threads within a block can cooperate by sharing data through some *shared memory* and by synchronizing their execution to coordinate memory accesses. More precisely, one can specify synchronization points in the kernel by calling the `__syncthreads()` intrinsic function; `__syncthreads()` acts as a barrier at which all threads in the block must wait before any is allowed to proceed. Shared Memory gives an example of using shared memory. In addition to `__syncthreads()`, the Cooperative Groups API provides a rich set of thread-synchronization primitives.

For efficient cooperation, the shared memory is expected to be a low-latency memory near each processor core (much like an L1 cache) and `__syncthreads()` is expected to be lightweight.