Chapter 1

The Basics of GPU Programming

Welcome to GPU computing. In the era of sequential codes we could only move walking. Multicore CPUs invented the car. Today you will discover the plane, and that you are surrounded by them. CUDA (Compute Unified Device Architecture) is the computation paradigm and hardware model for you to become a good pilot. Our world is heterogeneous computing where cars and planes coexists, and so you need to know when it is worth to get on the plane. Daily short distances are for the CPU. The GPU brings you more exciting destinations, those of HPC and large-scale computation: Big data, artificial intelligence, ... Let's go for it!

SECTION 1.1

Parallel Programming and Architecture

Many-core GPUs have transformed modern computing into an acceleration race where parallel programming is a must. Multi-core CPUs started the race, with programmers using *task parallelism* to create a bunch of threads executing independent tasks. The GPU follows a model based on *data parallelism*: SIMT (Single Instruction Multiple Thread). With SIMT we create a single program, the GPU kernel, where billions of threads may coexist to work on partitioned data using fine-grain parallelism, and cores execute threads simultaneously to promote scalability.

SIMT

kernels

For example, say that our algorithm is housekeeping. Task parallelism defines (1) sweeping the floor, (2) scrubbing the floor, (3) washing clothes, (4) ironing clothes, ... and you will not find more than a dozen tasks, which means idle cores beyond that number. SIMT starts with the sweeping kernel, creating as many parallel threads as dust particles, and so you write the code to remove one of them. Once the GPU cores execute all these threads, we launch the scrubbing kernel, and so on. Kernels are scalable as long as there are many more dust

scalability

particles (data elements) than GPU cores: In other words, when the GPU doubles the number of cores, execution time accelerates 2x, even without recompiling the code. Meanwhile, the scrubbing thread in the CPU is waiting for the sweeping thread to finish, and so ironing will wait for washing to be completed, all affected by data dependencies. Counting on more cores does not help the CPU either, because you cannot extract massive parallelism.

If your GPU has 100 cores and your house has 200 floor tiles, you may think it is more efficient for the kernel to specify how to sweep a floor tile. That way, each core would execute just two threads instead of billions of them. This idea does not work for a GPU in the future having more than 200 cores, but to preserve scalability you can create a tile of cores for the execution model to operate at a higher level. The tile of data is called a CUDA block in the software paradigm, and the tile of cores is called a multiprocessor (or SM) in the hardware model. Overall, the GPU executes CUDA kernels composed of blocks assigned to multiprocessors and threads per block assigned to those cores within the multiprocessor. Figure 1.1 summarizes all these basic relations.

blocks

SMs

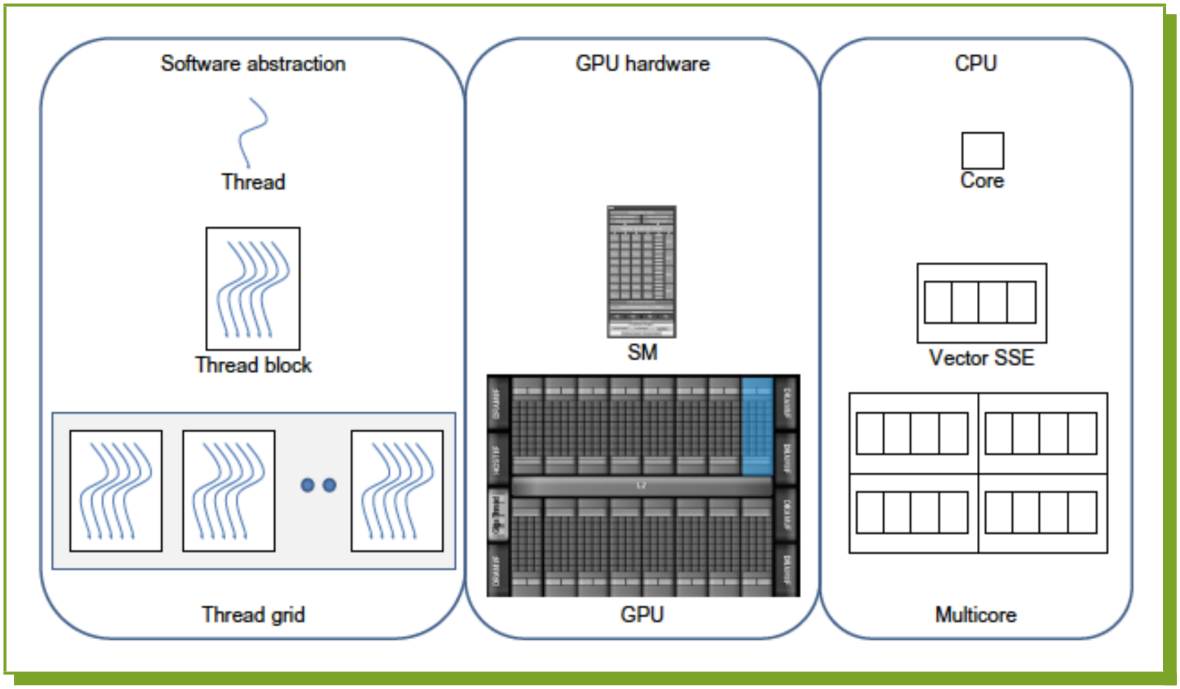

FIGURE 1.1: CUDA software paradigm and hardware model for the many-core GPU architecture, and comparison with the multi-core CPU.

For example, if we launch a kernel composed of 100 blocks and 1024 threads per block on a GPU endowed with 50 multiprocessors of 128 cores, each multiprocessor takes 2 blocks and each core executes 8 threads for each block. Before we start programming GPU kernels, it is good to know the GPU architecture, so our next section introduces hardware generations.

Hardware generation (chip)	Time frame	Number of multiprocessors	Number of cores per multiprocessor	Total number of cores
Tesla (G80)	2006-09	16	8	128
Fermi (GF100)	2010-11	16	32	512
Kepler (GK110)	2012-13	15	192	2880
Maxwell (GM200)	2014-15	24	128	3072
Pascal (GP100)	2016-17	56	64	3584
Volta (GV100)	2018-20	80	64	5120
+Turing (TU102)	2019-20	72	64	4608
Ampere (A100)	2020-?	108	64	6912

TABLE 1.1: The GPU evolution in number of 32-bit cores through hardware generations.

Hardware feature	Tesla V100 (Volta)	Titan RTX (Turing)	A100 (Ampere)
GPU (chip)	GV100	TU102	GA100
fp32 cores	5120	4608	6912
fp64 cores	2560	144	4096
Frequency (base-boost)	1370-1455 MHz	1440-1770 MHz	1410 MHz
TFLOPS (fp16, fp32, fp64)	30, 15, 7.5	32.6, 16.3, 0.51	78, 19.5, 9.7
Memory interface	HBM2 4096 bits	GDDR6 384 bits	HBM2 5 stacks
Memory bandwidth	900 GB/s.	672 GB/s	1555 GB/s
Video memory	16 or 32 GB	24 GB	48 GB
L2 cache	6 MB	6 MB	40 MB
Shared memory / SM	Up to 96 KB	Up to 64 KB	Up to 164 KB
Register file / SM	65536 (32-bit)	65536 (32-bit)	65536 (32-bit)

TABLE 1.2: Computational and memory resources available in 3 flagship GPUs.

SECTION 1.2

The GPU Architecture

The GPU has evolved through 7 generations since 2007. Table 1.1 summarizes this evolution through number of cores executing 32-bit integer/floating-point computation. Later, we incorporated double-precision in Fermi, half-precision in Pascal and Tensor cores in Volta. Latest cores are effectively used via libraries like cuTensor [1].

evolution

1.2.1 ▶ Computational units

The SM (Streaming Multiprocessor) is the cornerstone of every GPU chip. Figure 1.2 shows its basic structure and how software entities use its memory hierarchy.

warps

Threads are organized in groups of 32 called *warps*, which are SIMT lanes progressing simultaneously and sharing the instruction issue from the SIMT control unit. You may see warps like the mouth of the SM: The same way the human mouth can use its 32 teeth to bite 32 noodles in parallel, the GPU executes and controls 32 threads together. Pros are hardware simplicity. Cons are *warp divergencies*, a situation that arises when those 32 threads execute a conditional statement which is true for X threads and false for 32-X threads. In that case, the control unit issues X threads first and then the remaining ones, with a performance penalty.

accuracy

Low-end GPUs contain a small number of SMs, high-end GPUs provide many more, and subsequently will execute CUDA blocks proportionally faster. Another performance factor is how fast blocks are executed, and programmers may affect this: For example, certain GPUs have SMs with similar number of fp32 and fp64 cores where numerical accuracy will barely affect performance, but in other cases differences are huge. In other words, replacing `float` by `double` data types on a CUDA C code has a different slowdown factor on each SM. Table 1.2 summarizes computational and memory resources available in three representative GPUs.

1.2.2 ▶ Memory units

In the CUDA memory model there is a hierarchy composed of three basic levels:

by thread

❶ **The register file.** Each SM provides around 64K registers, which will be split evenly among all the threads of the blocks assigned to it. Hence, the number of registers needed in the computation will affect the number of threads able to be executed simultaneously. For example, if a kernel consumes 64 registers/thread, only 1024 threads can be assigned to each SM, which can be executed by 1 block of 1024 threads, 2 blocks of 512 threads, and so on.

by block

❷ **The shared memory.** Each SM provides few kilobytes of shared memory visible to the programmer and where threads within a block can communicate each other. For example, for a shared memory of 128 Kbytes and a block consuming 16 kilobytes, the scheduler will assign 8 CUDA blocks to every SM.

by kernel

❸ **The global memory.** This is the video memory (HBM2/GDDR6), which is accessed from any SM in regular memory accesses. Texture and constant caches are secondary memory resources also mapped to video memory.

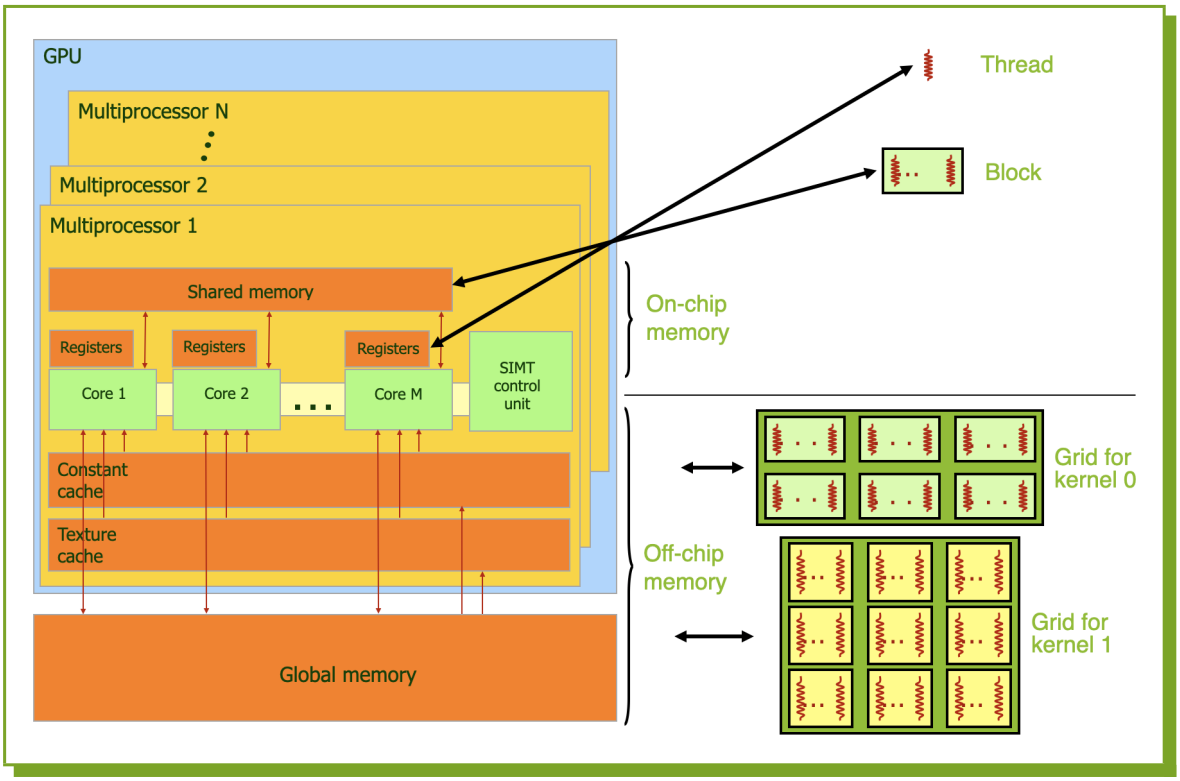

FIGURE 1.2: Basic structure of a GPU multiprocessor (SM) and mapping with software entities.

Trade off between computation and memory ◀ 1.2.3

In general, GPUs are more than one order of magnitude ahead of CPUs in peak performance (TFLOPS) and memory bandwidth (GB/s). Programmers are usually captivated by TFLOPS, but only 30% of scientific large-scale applications are compute-bound. The remaining 70% are memory-bound, which is why is so important to exploit the memory hierarchy wisely. As we have seen, the more registers and shared memory you use, the lower number of blocks are deployed on each multiprocessor. You should increase the degree of parallelism in compute-bound applications, and the use of fast memory in memory-bound applications. The register file and the shared memory are around 500 times faster than global memory, and their sizes are sensitive to each GPU generation (see Table 1.2).

memory vs. parallelism

Profiling tools are good to analyze if your kernel is compute- or memory-bound, but if you want to investigate that manually, here is a basic rule: calculate the operational intensity of your kernel as the fraction between the number of arithmetic operations performed on data structures and the number of bytes occupied by them. An operational intensity higher than 4-8 usually means a compute-bound kernel (which is not that likely to be found).

operational intensity

SECTION 1.3

Key Use Cases

data size

Today, millions of NVIDIA GPUs are accelerating many assorted types of computationally-intensive applications. They all have in common the use of large data structures, and the departure point for you to think about porting codes to the GPU is around ten million data. Otherwise, the travel is too short for the plane to amortize the cost of moving data back and forth, mainly because of latency.

forall

That said, every application contains sections of code which are sequential or minimally parallel, interleaved with others that are embarrasingly parallel and will be transformed into your CUDA kernels. How do you identify them? Our first candidates are large loops traversing data without carried-loop dependencies, which are called `forall` loops.

A way to recognize a `forall` loop is to start from the last iteration moving backwards, or first compute even iterations and then odd iterations, and final results always match. It you always get the same results, that is a `forall` loop.

Now we focus on typical cases which constitute the building blocks for GPU parallelism.

1.3.1 ▶ Streaming operators

A streaming operator traverses a loop to transform input arrays into output arrays applying a transformation which is independent for each loop iteration and output value.

For example, in image processing, you can convert a color image in RGB format into a grayscale version using the formula of the luminance for each pixel.

```
float r[Npixels], g[Npixels], b[Npixels], luminance[Npixels];
for (i=0; i<Npixels; i++)
  luminance[i] = 255 * (0.299 * r[i] + 0.587 * g[i] + 0.114 * b[i]);
```

This is the simplest expression of a `forall` loop. A CUDA implementation would extract parallelism by declaring as many threads as number of pixels, each taking care of a single loop iteration.

1.3.2 ▶ Array constructors

These are typical operations in linear algebra, where vector addition is a good example.

```
float x[N], y[N], z[N];
for (i=0; i<N; i++)
  x[i] = y[i] + z[i];
```

Again, this is a `forall` loop, and the CUDA kernel would be launched for N threads, each calculating an element of `x[]`. Like in the previous case, the CUDA kernel here contains a single instruction.

Stencil operators ◀ 1.3.3

Stencil codes are iterative methods to transform array elements according to a pattern (called the stencil) involving neighboring elements. They are most commonly found in computer simulations like fluid dynamics and modeling such as finite elements. A good exponent is the Jacobi kernel for solving the Laplace's differential equation on a square domain, regularly discretized [2].

```
int i, j, iter, N, Niters;
float in[N][N], out[N][N];
for (iter=0; iter<Niters; iter++) {
  for (i=1; i<N-1; i++)
    for (j=1; j<N-1; j++)
      out[i][j] = 0.2 * (in[i][j] + in[i-1][j] + in[i+1][j] + in[i][j-1] + in[i][j+1]);
  for (i=1; i<N-1; i++)
    for (j=1; j<N-1; j++)
      in[i][j] = out[i][j]; }
```

This code implements the following idea: Let us consider a body represented by a 2D array of particles, each with an initial value of temperature. This body is in contact with a fixed value of temperature on the four boundaries, and Laplace's equation is solved for all internal points to determine their temperature as the average of the four neighboring particles. Taking this task as the computational core, a number of iterations are performed over the data to recompute average temperatures repeatedly, and the values gradually converge to a finer solution until the desired accuracy is reached.

Note that iterations have to be serialized due to carried-loop dependencies, but parallelism is enabled within iterations because the computation at each particle is independent. Thus, the workload depends more on the number of iterations, and the amount of parallelism that can be extracted from the code relies more on the size of the 2D input matrix. Loops 2, 3, 4 and 5 are `forall` loops, but all threads have to be synchronized between the two assignments to prevent that any thread may update `in[]` while others are still reading its values.

1.3.4 ▶ Reduction operators

Reductions are associative operators (typically additions or products) to combine the elements of an array into a single result. For example:

```
float sum, x[N];
sum = 0;
for (i=0; i<N; i++)
  sum += x[i];
```

This code has carried-loop dependencies, but parallelism can exploit the associativity to compute partial results following the pattern of a binary tree to compute results in O(logN) steps. The challenge here is to use the pattern that better exploits the GPU memory hierarchy.

1.3.5 ▶ Histograms

A histogram is the discretized representation of the distribution of numerical data into as many bins or consecutive, non-overlapping intervals of a variable. A typical example is the histogram of a grayscale image representing how many pixels have each of the intensity levels (the bins).

```
int histo[Nbins], image[N][N];
sum = 0;
for (i=0; i<Nbins; i++)
  histo[Nbins] = 0;
for (i=0; i<N; i++)
  for (i=0; i<N; i++)
    histo[image[i][j]]++;
```

atomic ops.

We first have a forall loop, but short and lightweight to exploit parallelism. The following two nested loops have carried-loop dependencies, but `image` values can be read in parallel and assigned to CUDA threads. Then, CUDA provides a mechanism called *atomic operations* (`atomicInc(histo[image[i][j]])`) to serialize concurrent accesses to the `histo` vector, and can be used here to avoid what are called *race conditions*. In CUDA, atomic operations generally work for both shared and global memory.

1.3.6 ▶ Final remarks

Table 1.3 summarizes our findings after your first experience with data parallelism. Overall, we may conclude:

Code feature	Streaming Operator	Array Constructor	Stencil Operator	Reduction Operator	Histogram
Input size	3*N	2*N	N^2	N	N^2
Output size	N	N	N^2	1	No. bins
Computational weight	Low	Low	Heavy	Low	High
Computat. complexity	O(N)	O(N)	O(it*N^2)	O(N)	O(N^2)
Memory usage	Average	Average	Intensive	Low	High
Data reuse	None	None	High	Low	Low
Op. intensity (FLOP/byte)	0.375	0.083	0.625	0.25	0.25
Shared memory candidates	No	No	`in[]`	`sum`	`histo[]`
Estimated GPU speed-up	High	High	Fair	Low	Low

TABLE 1.3: Summary of features for our representative kernels to extract GPU parallelism.

1. The streaming operator will likely reach the higher GPU acceleration (expect more than 100x on large-scale images).
2. The stencil operator is the best candidate for using shared memory.
3. The reduction operator is the implementation where the speed-up will be more sensitive to CUDA programming skills. Try to see it as your first challenge and work on the best memory access pattern from parallel threads.
4. The histogram is the toughest CUDA implementation. In fact, our intention was to move from easier to hardest kernel along this section.

SECTION 1.4

Nvidia GPUs for General-Purpose Parallel Programming

Data parallelism and its SIMD (Single Instruction Multiple Data) execution model was proposed in the early 80's for supercomputers, and later in multimedia instructions for CPUs (MMX, SSE, ...). There are four basic reasons to understand why the SIMT CUDA model has been more successful:

1. **Control Unit is simplified.** The logic required to control a thread is amortized by another 31 within the warp, leaving more room to useful functional units.
2. **Scalability.** CUDA blocks and SMs define a sustainable parallelization model, as already explained in section 1.1.

❸ **Lightweight threads and context switch.** CUDA threads do not share the register file, but partition it in local spaces to avoid push/pop values to a memory stack.

❹ **Hide latencies.** When a warp is stalled (for example, due to a memory access), the GPU can immediately switch to another warp ready to continue. It is the main difference between the SIMT model with parallel lanes of warps decoupled and the SIMD model where all these parallel lanes are synchronized and cannot hide latencies from others because they all issue the same instruction.

energy

In addition, the GPU has evolved as a power-efficient processor leading the green500.org list, and the number of applications interested in GFLOPS per watt is growing exponentially. Figure 1.3 compares the CPU and the GPU models for general-purpose computing to finally characterize cars and planes.

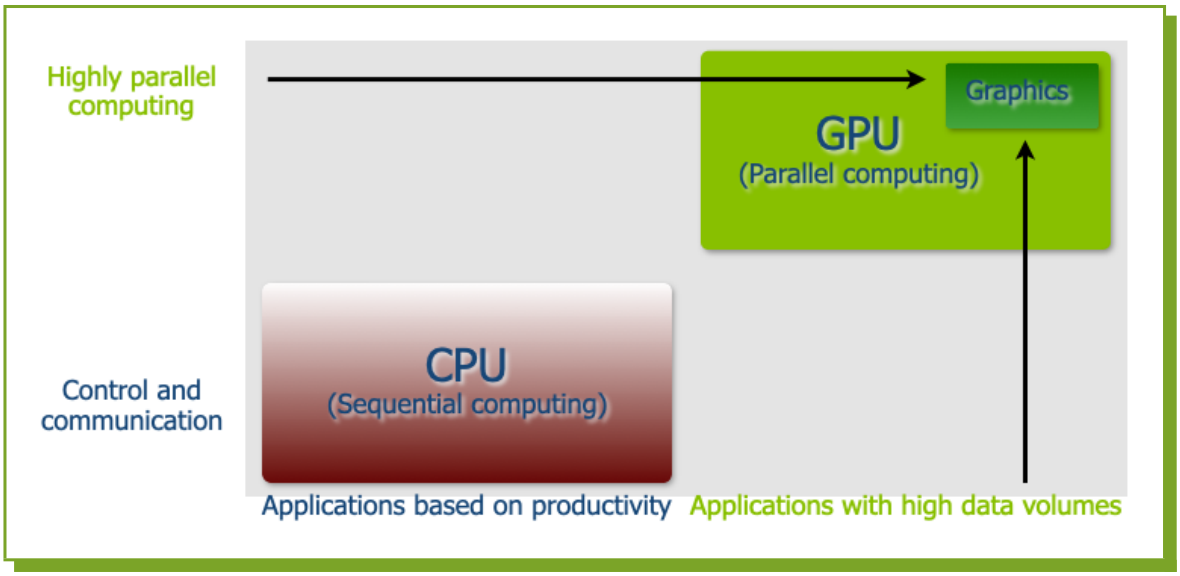

FIGURE 1.3: A comparison between the CPU and the GPU computational models.