1

Introduction to GPU Kernels and Hardware

This book aims to teach you how to use graphics processing units (GPUs) and Compute Unified Device Architecture (CUDA) to speed up your scientific or technical computing tasks. We know from personal experience that the best way to learn to speak a new language is to go to the relevant country and immerse yourself in the culture. Thus, we have chosen to start our book with a complete working example of an interesting problem. We present three versions of the code, firstly a standard C++ implementation for a single central processing unit (CPU) thread, and secondly a multithread CPU version suitable for running on one or two threads on each core for a multicore CPU, say between 4 and 16 threads. The third version uses CUDA to run with thousands of simultaneous threads. We don't expect readers to immediately grasp all the nuances in the CUDA code – that is what the rest of this book is for. Rather I hope you will see how similar the code is in all three versions and be encouraged that GPU programming is not difficult and that it brings huge rewards.

After discussing these introductory examples, we go on to briefly recap the architecture of traditional PCs and then introduce NVIDIA GPUs, introducing both their hardware features and the CUDA programming model.

1.1 Background

A modern PC processor now has two, four or more computing CPU cores. To get the best from such hardware, your code has to be able to run in parallel on all the resources available. In favourable cases, tools like OpenMP or the C++11 thread class defined in `<thread>` allow you to launch cooperating threads on each of the hardware cores to get a potential speed-up proportional to the number of cores. This approach can be extended to clusters of PCs using communication tools like Message Passing Interface (MPI) to manage the inter-PC communication. PC clusters are indeed now the dominant architecture in high-performance computing (HPC). A cluster of at least 25 PCs with 8-core CPUs would be needed to give a factor of 200 in performance. This is doable but expensive and incurs significant power and management overheads.

An alternative is to equip your PC with a modern, reasonably high-specification GPU. The examples in this book are based on an NVIDIA RTX 2070 GPU, which was bought for £480 in March 2019. With such a GPU and using NVIDIA's C++-like CUDA language, speed-ups of 200 and often much more can be obtained on a single PC with really quite modest effort. An additional advantage of the GPU is that its internal memory is about 10 times faster than that of a typical PC, which is extremely helpful for problems limited by memory bandwidth rather than CPU power.

https://doi.org/10.1017/9781108855273.002 Published online by Cambridge University Press

At the heart of any CUDA program are one or more *kernel* functions, which contain the code that actually runs on the GPU. These kernel functions are written in standard C++ with a small number of extensions and restrictions. We believe they offer an exceptionally clear and elegant way of expressing the parallel content of your programs. This is why we have chosen CUDA for this book on parallel programming. One feature that distinguishes the book from other books on CUDA is that we have taken great care to provide interesting real-world problems for our CUDA examples. We have also coded these examples using features of modern C++ to write straightforward but elegant and compact code. Most of the presently available online tutorials or textbooks on CUDA use examples heavily based on those provided by the NVIDIA Software Development Kit (SDK) examples. These examples are excellent for demonstrating CUDA features but are mostly coded in a verbose, outdated C style that often hides their underlying simplicity.[1]

To get the best from CUDA programs (and, indeed, any other programming language), it is necessary to have a basic understanding of the underlying hardware, and that is the main topic of this introductory chapter. But, before that, we start with an example of an actual CUDA program; this is to give you a foretaste of what is to come – the details of the code presented here are fully covered in later chapters.

1.2 First CUDA Example

Here is our first example showing what is possible with CUDA. The example uses the trapezoidal rule to evaluate the integral of `sin(x)` from 0 to π, based on the sum of a large number of equally spaced evaluations of the function in this range. The number of steps is represented by the variable `steps` in the code. We deliberately choose a simple but computationally expensive method to evaluate `sin(x)`, namely, by summing the Taylor series for a number of terms represented by the variable `terms`. The sum of the sin values is accumulated, adjusted for end points and then scaled to give an approximation to the integral, for which the expected answer is 2.0. The user can set the values of `steps` and `terms` from the command line, and for performance measurements very large values are used, typically 10^6 or 10^9 steps on the CPU or GPU, respectively, and 10^3 terms.

Example 1.1 `cpusum` single CPU calculation of a sin integral

```
02 #include <stdio.h>
03 #include <stdlib.h>
04 #include "cxtimers.h"

05 inline float sinsum(float x, int terms)
06 {
     // sin(x) = x - x^3/3! + x^5/5! ...
07   float term = x;    // first term of series
08   float sum  = term; // sum of terms so far
09   float x2   = x*x;
10   for(int n = 1; n < terms; n++){
```

https://doi.org/10.1017/9781108855273.002 Published online by Cambridge University Press

```
11      term *= -x2 / (float)(2*n*(2*n+1));
12      sum += term;
13    }
14    return sum;
15  }

16  int main(int argc, char *argv[])
17  {
18    int steps = (argc >1) ? atoi(argv[1]) : 10000000;
19    int terms = (argc >2) ? atoi(argv[2]) : 1000;

20    double pi = 3.14159265358979323;
21    double step_size = pi/(steps-1); // n-1 steps

22    cx::timer tim;
23    double cpu_sum = 0.0;
24    for(int step = 0; step < steps; step++){
25      float x = step_size*step;
26      cpu_sum += sinsum(x, terms);    // sum of Taylor series
27    }
28    double cpu_time = tim.lap_ms(); // elapsed time

29    // Trapezoidal Rule correction
30    cpu_sum -= 0.5*(sinsum(0.0,terms)+sinsum(pi, terms));
31    cpu_sum *= step_size;
32    printf("cpu sum = %.10f,steps %d terms %d time %.3f ms\n",
                              cpu_sum, steps, terms, cpu_time);
33    return 0;
34  }

D:\ >cpusum.exe 1000000 1000
cpu sum = 1.9999999974,steps 1000000 terms 1000 time 1818.959 ms
```

We will show three versions of this example. The first version, `cpusum`, is shown in Example 1.1 and is written in straightforward C++ to run on a single thread on the host PC. The second version, `ompsum`, shown in Example 1.2 adds two OpenMP directives to the first version, which shares the loop over `steps` between multiple CPU threads shared equally by all the host CPU cores; this illustrates the best we can do on a multicore PC without using the GPU. The third version, `gpusum`, in Example 1.3 uses CUDA to share the work between 10^9 threads running on the GPU.

Description of Example 1.1

This is a complete listing of the `cpusum` program; most of our subsequent listings will omit standard headers to save space. Notice that we chose to use 4-byte floats rather than 8-byte doubles for the critical function `sinsum`. The reasons for this choice are discussed later in this chapter, but briefly we

https://doi.org/10.1017/9781108855273.002 Published online by Cambridge University Press

wish to exploit limited memory bandwidth and to improve calculation speed. For scientific work, the final results rarely need to be accurate to more than a few parts in 10^{-8} (a single bit error in an IEEE 4-byte float corresponds to a fractional error of 2^{-24} or $\sim 6 \times 10^{-8}$). But, of course, we must be careful that errors do not propagate as calculations progress; as a precaution the variable `cpusum` in the main routine is an 8-byte double.

- Lines 2–4: Include standard headers; the header `cxtimers.h` is part of our cx utilities and provides portable timers based on the C++11 `chrono.h` library.
- Lines 5–15: This is the `sinsum` function, which evaluates sin(x) using the standard Taylor series. The value of x in radians is given by the first input argument `x`, and the number of terms to be used is given by the second input argument `terms`.
- Lines 7–9: Initialise some working variables; `term` is the value of the current term in the Taylor series, `sum` is the sum of terms so far, and `x2` is x^2.
- Lines 10–13: This is the heart of our calculation, with a loop where successive terms are calculated in line 11 and added to `sum` in line 12. Note that line 11 is the single line where all the time-consuming calculations happen.

The main function of the remining code, in lines 16–35, is to organise the calculation in a straightforward way.

- Lines 18–19: Set the parameters `steps` and `terms` from optional user input.
- Line 21: Set the step size required to cover the interval between 0 and π using `steps` steps.
- Line 22: Declare and start the timer `tim`.
- Lines 23–27: A `for` loop to call the function `sinsum` `steps` times while incrementing `x` in to cover the desired range. The results are accumulated in `double cpusum`.
- Line 28: Store the elapsed (wall clock) time since line 22 in `cpu_time`. This member function also resets the timer.
- Lines 30–31: To get the integral of sin(x), we perform end-point corrections to `cpusum` and scale by `step_size` (i.e. `dx`).
- Line 31: Print result, including time, is `ms`. Note that the result is accurate to nine significant figures in spite of using floats in the function `sinsum`.

The example shows a typical command line launch requesting 10^6 steps and 10^3 terms in each step. The result is accurate to nine significant figures. Lines 11 and 12 are executed 10^9 times in 1.8 seconds, equivalent to a few GFlops/sec.

In the second version, Example 1.2, we use the readily available OpenMP library to share the calculation between several threads running simultaneously on the cores of our host CPU.

Example 1.2 `ompsum` OMP CPU calculation of a sin integral

```
02   #include <stdio.h>
03   #include <stdlib.h>
03.5 #include <omp.h>
04   #include "cxtimers.h"
```

https://doi.org/10.1017/9781108855273.002 Published online by Cambridge University Press

```
05   float sinsum(float x, int terms)
06   {
     . . . same as (a)
15   }

16   int main(int argc, char *argv[])
     . . .
19.5 int threads = (argc >3) ? atoi(argv[3]) : 4;
     . . .
23.5 omp_set_num_threads(threads);                    // OpenMP
23.6 #pragma omp parallel for reduction (+:omp_sum)   // OpenMP
24   for(int step = 0; step < steps; step++){
     . . .
32   printf("omp sum = %.10f,steps %d terms %d
         time %.3f ms\n", omp_sum,steps,terms,cpu_time);
33   return 0;
34 }

D:\ >ompsum.exe 1000000 1000 4   (4 threads)
omp sum = 1.9999999978, steps 1000000 terms 1000 time 508.635 ms
D:\ >ompsum.exe 1000000 1000 8   (8 threads)
omp sum = 1.9999999978, steps 1000000 terms 1000 time 477.961 ms
```

Description of Example 1.2

We just need to add three lines of code to the previous Example 1.1.

- Line 3.5: An extra line to include the header file `omp.h`. This has all the necessary definitions required to use OpenMP.
- Line 19.5: An extra line to add the user-settable variable `threads`, which sets the number of CPU threads used by OpenMP.
- Line 23.5: This is actually just a function call that tells `openMP` how many parallel threads to use. If omitted, the number of hardware cores is used as a default. This function can be called more than once if you want to use different numbers of cores in different parts of your code. The variable `threads` is used here.
- Line 23.6: This line sets up the parallel calculation. It is a compile time directive (or pragma) telling the compiler that the immediately following `for` loop is to be split into a number of sub-loops, the range of each sub-loop being an appropriate part of the total range. Each sub-loop is executed in parallel on different CPU threads. For this to work, each sub-loop will get a separate set of the loop variables, `x` and `omp_sum` (N.B.: We use `omp_sum` instead of `cpu_sum` in this section of the code). The variable `x` is set on each pass through the loop with no dependencies on previous passes, so parallel execution is not problematic. However, that is not the case for the variable `omp_sum`,

https://doi.org/10.1017/9781108855273.002 Published online by Cambridge University Press

which is supposed to accumulate the sum of all the `sin(x)` values. This means the sub-loops have to cooperate in some way. In fact, the operation of summing a large number of variables, held either in an array or during loop execution, occurs frequently and is called a *reduce* operation. Reduce is an example of a *parallel primitive*, which is a topic we discuss in detail in Chapter 2. The key point is that the final sum does not depend on the order of the additions; thus, each sub-loop can accumulate its own partial sum, and these partial sums can then be added together to calculate the final value of the `sum_host` variable after the parallel `for`. The last part of the pragma tells OpenMP that the loop is indeed a reduction operation (using addition) on the variable `omp_sum`. OpenMP will add the partial sums accumulated by each thread's copy of `omp_sum` and place the final result into the `omp_sum` variable in our code at the end of the loop.

- Line 32: Here we have simply modified the existing `printf` to also output the value of `threads`.

Two command line launches are shown at the end of this example, the first using four OMP threads and the second using eight OMP threads.

The results of running `ompsum` on an Intel quad-core processor with hyper-threading are shown at the bottom of the example using either four or eight threads. For eight threads the speed-up is a factor of 3.8 which is a good return for little effort. Note using eight cores instead of four for our PC means running two threads on each core which is supported by Intel hyper-threading on this CPU; we see a modest gain but nothing like a factor of 2.

In Visual Studio C++, we also have to tell the compiler that we are using OpenMP using the properties dialog, as shown in Figure 1.1.

In the third version, Example 1.3, we use a GPU and CUDA, and again we parallelise the code by using multiple threads for the loop in lines 24–27, but this time we use a separate thread for each iteration of the loop, a total of 10^9 threads for the case shown here. The code changes for the GPU computation are a bit more extensive than was required for OpenMP, but as an incentive to continue reading, we will find that the speed-up is now a factor of 960 rather than 3.8! This dramatic gain is an example of why GPUs are routinely used in HPC systems.

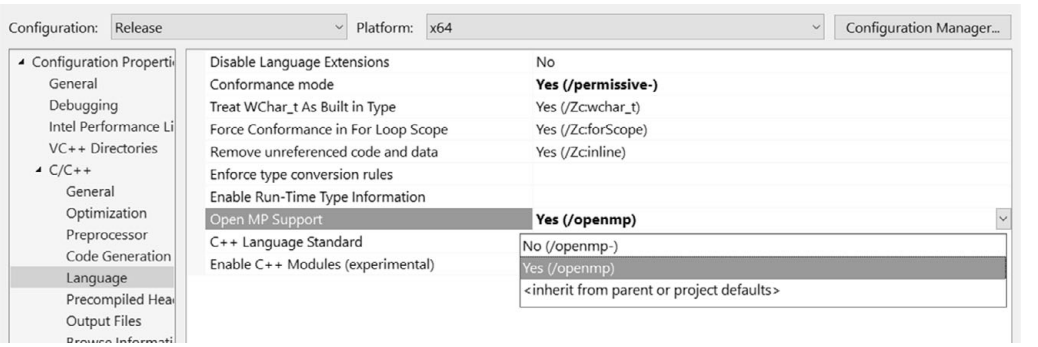

Figure 1.1 How to enable OpenMP in Visual Studio

https://doi.org/10.1017/9781108855273.002 Published online by Cambridge University Press

Example 1.3 gpusum GPU calculation of a sin integral

```
01   // call sinsum steps times using parallel threads on GPU
02   #include <stdio.h>
03   #include <stdlib.h>
04   #include "cxtimers.h"              // cx timers
04.1 #include "cuda_runtime.h"          // cuda basic
04.2 #include "thrust/device_vector.h"  // thrust device vectors

05   __host__ __device__ inline float sinsum(float x, int terms)
06   {
07     float x2 = x*x;
08     float term = x;   // first term of series
09     float sum = term; // sum of terms so far
10     for(int n = 1; n < terms; n++){
11       term *= -x2 / (2*n*(2*n+1));  // build factorial
12       sum += term;
13     }
14     return sum;
15   }

15.1 __global__ void gpu_sin(float *sums, int steps, int terms,
      float step_size)
15.2 {
       // unique thread ID
15.3   int step = blockIdx.x*blockDim.x+threadIdx.x;
15.4   if(step<steps){
15.5     float x = step_size*step;
15.6     sums[step] = sinsum(x, terms);  // store sums
15.7   }
15.8 }

16   int main(int argc, char *argv[])
17   {
       // get command line arguments
18     int steps = (argc >1) ? atoi(argv[1]) : 10000000;
19     int terms = (argc >2) ? atoi(argv[2]) : 1000;
19.1   int threads = 256;
19.2   int blocks = (steps+threads-1)/threads;  // round up

20     double pi = 3.14159265358979323;
21     double step_size = pi / (steps-1); // NB n-1
       // allocate GPU buffer and get pointer
21.1   thrust::device_vector<float> dsums(steps);
21.2   float *dptr = thrust::raw_pointer_cast(&dsums[0]);
22     cx::timer tim;
```

https://doi.org/10.1017/9781108855273.002 Published online by Cambridge University Press

```
22.1 gpu_sin<<<blocks,threads>>>(dptr,steps,terms,
                          (float)step_size);
22.2 double gpu_sum =
            thrust::reduce(dsums.begin(),dsums.end());
28   double gpu_time = tim.lap_ms(); // get elapsed time
29   // Trapezoidal Rule Correction
30   gpu_sum -= 0.5*(sinsum(0.0f,terms)+sinsum(pi, terms));
31   gpu_sum *= step_size;
32   printf("gpusum %.10f steps %d terms %d
          time %.3f ms\n",gpu_sum,steps,terms,gpu_time);
33   return 0;
34  }

D:\ >gpusum.exe 1000000000 1000
gpusum = 2.0000000134 steps 1000000000 terms 1000 time 1882.707 ms
```

Description of Example 1.3

This description is here for the sake of completeness. If you already know a bit of CUDA, it will make sense. If you are new to CUDA, skip this description for now and come back later when you have read our introduction to CUDA. At this point, the message to take away is that potentially massive speed-ups can be achieved and that, to my eyes at least, the code is elegant, expressive and compact and the coding effort is small.

The details of the CUDA methods used here are fully described later. However, for now you should notice that much of the code is unchanged. CUDA is written in C++ with a few extra keywords; there is no assembly to learn. All the details of the calculation are visible in the code. In this listing, line numbers without dots are exactly the same lines in Example 1.1, although we use `gpu` instead of `cpu` in some of the variable names.

- Lines 1–4: These include statements are the same as in Example 1.1.
- Line 4.1: This is the standard include file needed for all CUDA programs. A simple CUDA program just needs this, but there are others that will be introduced when needed.
- Line 4.2: This include file is part of the Thrust library and provides support for thrust vectors on the GPU. Thrust vector objects are similar to the `std::vector` objects in C++, but note that CUDA has separate classes for thrust vectors in CPU memory and in device memory.
- Lines 5–15: This is the same `sinsum` function used in Example 1.1; the only difference is that in line 5 we have decorated the function declaration with `__host__` and `__device__`, which tell the compiler to make two versions of the function, one suitable for code running on the CPU (as before) and one for code running on the GPU. This is a brilliant feature of CUDA: literally the same code can be used on both the host and device, removing a major source of bugs.[2]
- Lines 15.1–15.8: These define the CUDA kernel function `gpu_sin` that replaces the loop over `steps` in lines 24–27 of the original program. Whereas OpenMP uses a small number of host threads, CUDA uses a very large number of GPU threads. In this case we use 10^9 threads, a separate thread for each value of `step` in the original for loop. Kernel functions are declared with the keyword `__global__` and are launched by the host code. Kernel functions can receive arguments from the host but cannot return values – hence they must be declared as `void`. Arguments can either

https://doi.org/10.1017/9781108855273.002 Published online by Cambridge University Press

be passed to kernels by value (good for single numbers) or as pointers to previously allocated device memory. Arguments cannot be passed by reference, as in general the GPU cannot directly access host memory.

Line 15.3 of the kernel function is especially noteworthy, as it encapsulates the essence of parallel programming in both CUDA and MPI. You have to imagine that the code of the kernel function is running simultaneously for all threads. Line 15.3 contains the magic formula used by each particular instance of an executing thread to figure out which particular value of the index `step` that it needs to use. The details of this formula will be discussed later in Table 1.1. Line 15.4 is an out-of-range check, necessary because the number of threads launched has been rounded up to a multiple of 256.

- Lines 15.5 and 15.6 of the kernel: These correspond to the body of the for loop (i.e. lines 25–26 in Example 1.1). One important difference is that the results are stored in parallel to a large array in the global GPU memory, instead of being summed sequentially to a unique variable. This is a common tactic used to avoid serial bottlenecks in parallel code.
- Lines 16–19 of `main`: These are identical to the corresponding lines in Example 1.1.
- Lines 19.1–19.2: Here we define two new variables, `threads` and `blocks`; we will meet these variables in every CUDA program we write. NVIDIA GPUs process threads in blocks. Our variables define the number of threads in each block (`threads`) and the number of thread blocks (`blocks`). The value of threads should be a multiple of 32, and the number of blocks can be very large.
- Lines 20–21: These are the same as in Example 1.1.
- Line 21.1: Here we allocate an array `dsum` in GPU memory of size `steps`. This works like `std::vector` except we use the CUDA thrust class. The array will be initialised to zero on the device.
- Line 21.2: Here we create a pointer `dptr` to the memory of the `dsum` vector. This is a suitable argument for kernel functions.
- Lines 22.1–22.2: These two lines replace the `for` loop in lines 23–27 of Example 1.1, which called `sinsum steps` times sequentially. Here line 22.1 launches the kernel `gpu_sin`, which uses `steps` separate GPU threads to call `sinsum` for all the required `x` values in parallel. The individual results are stored in the device array `dsums`. In line 22.2 we call the reduce function from the thrust library to add all the values stored in `dsums`, and then copy the result from the GPU back to the host variable `dsum`.[3]
- Lines 28–34: These remaining lines are identical to Example 1.1; notice that the host version of our `sinsum` function is used in line 30.

As a final comment we notice that the result from the CUDA version is a little less accurate than either of the host versions. This is because the CUDA version uses 4-byte floats throughout the calculation, including the final reduction step, whereas the host versions use an 8-byte double to accumulate the final result sum over 10^6 steps. Nevertheless, the CUDA result is accurate to eight significant figures, which is more than enough for most scientific applications.

The `sinsum` example is designed to require lots of calculation while needing very little memory access. Since reading and writing to memory are typically much slower than performing calculations, we expect both the host CPU and the GPU to perform at their best efficiencies in this example. In Chapter 10, when we discuss profiling, we will see that the GPU is delivering several TFlops/sec in the example. While the `sinsum` function used in this example is not particularly interesting, the brute force integration method used here could be

https://doi.org/10.1017/9781108855273.002 Published online by Cambridge University Press

used for any calculable function spanned by a suitable grid of points. Here we used 10^9 points, which is enough to sample a function on a 3D Cartesian grid with 1000 points along each of the three coordinate axes. Being able to easily scale up to 3D versions of problems that can only be reasonably done in 2D on a normal PC is another great reason to learn about CUDA.

In order to write effective programs for your GPU (or CPU), it is necessary to have some feeling for the capabilities of the underlying hardware, and that is our next topic. So, after this quick look at CUDA code and what it can do, it is time to go back to the beginning and remind ourselves of the basics of computer hardware.

1.3 CPU Architecture

Correct computer code can be written by simply following the formal rules of the particular language being used. However, compiled code actually runs on physical hardware, so it is helpful to have some insights into hardware constraints when designing high-performance code. This section provides a brief overview of the important features in conventional CPUs and GPUs. Figure 1.2 shows a simplified sketch of the architecture of a traditional CPU.

Briefly the blocks shown are:

- Master Clock: The clock acts like the conductor of an orchestra, but it plays a very boring tune. Clock-pulses at a fixed frequency are sent to each unit causing that unit to execute its next step. The CPU processing speed is directly proportional to this frequency. The first IBM PCs were launched in 1981 with a clock-frequency of 2.2 MHz; the frequency then doubled every three years or so peaking at 4 GHz in 2002. It turned out that 4 GHz was the fastest that Intel was able to produce reliability, because the power requirement (and hence heat generated) is proportional to frequency. Current Intel CPUs typically run at ~3.5 GHz with a turbo boost to 4 GHz for short periods.
- Memory: The main memory holds both the program data and the machine code instructions output by the compiler from your high-level code. In other words, your program code is treated as just another form of data. Data from memory can be read from memory by either the load/save unit or the program fetch unit but normally only the load/save unit can write data back to the main memory.[4]

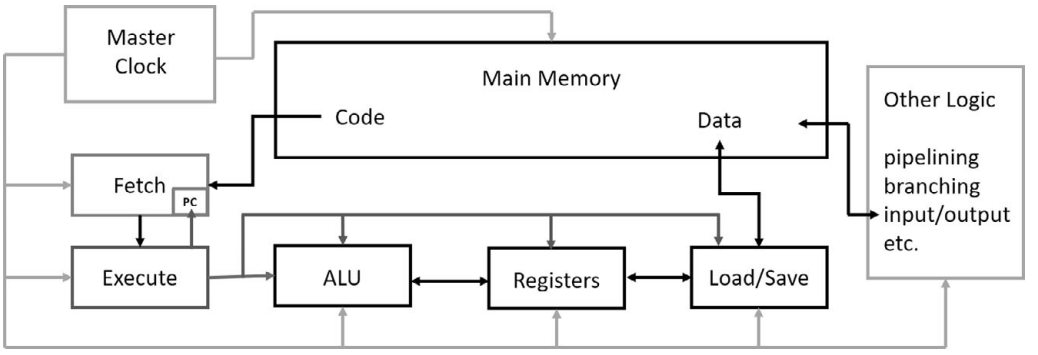

Figure 1.2 Simplified CPU architecture

https://doi.org/10.1017/9781108855273.002 Published online by Cambridge University Press

- Load/Save: This unit reads data from and sends data to the main memory. The unit is controlled by the Execute logic which at each step specifies if data is to be read or written to main memory. The data in question is transferred to or from one of the registers in the register file.
- Register File: This is the heart of a CPU; data must be stored in one or more registers in order to be operated on by the ALU.
- ALU or Arithmetic Logic Unit: This device performs arithmetic and logical operations in data stored in registers; at each step the required operation is specified by the execute unit. This is the piece of hardware that actually computes!
- Execute: This unit decodes the instruction sent by the instruction fetch unit, organises the transfer of input data to the register file, instructs ALU to perform the required operation on the data and then finally organises the transfer of results back to memory.
- Fetch: The fetch unit fetches instructions from main memory and passes them to the execute unit. The unit contains a register holding the program counter (PC)[5] that contains the address of the current instruction. The PC is normally incremented by one instruction at each step so that the instructions are executed sequentially. However, if a branch is required, the fetch unit changes the PC to point to the instruction at the branch point.

1.4 CPU Compute Power

The computing power of CPUs has increased spectacularly over time as shown in Figure 1.3.

The transistor count per chip has followed Moore's law's exponential growth from 1970. The CPU frequency stopped rising in 2002. The performance per core continues to rise

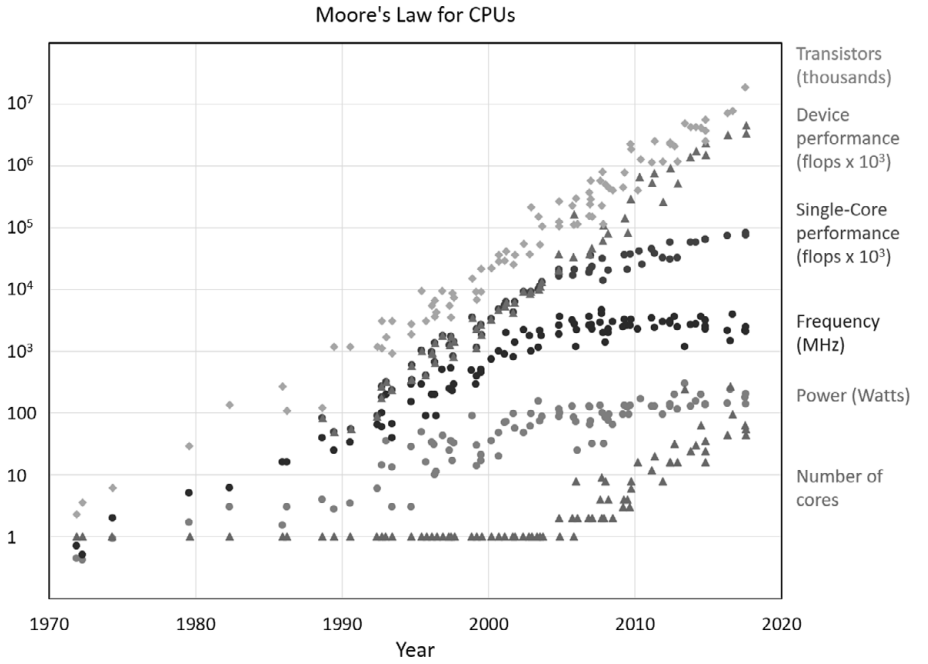

Figure 1.3 Moore's law for CPUs

https://doi.org/10.1017/9781108855273.002 Published online by Cambridge University Press

slowly but recent growth has been due to innovations in design rather than increase in frequency. The major contribution to performance per chip since 2002 has been from multicore technology. GPUs are excluded from this plot although recent Intel Xeon-phi designs with hundreds of cores are becoming quite GPU-like. Remarkably the power used by a single device has also not increased since 2002. The data are from https://github.com/karlrupp/microprocessor-trend-data.

It is really worth taking a moment to contemplate the story told in this figure. The compute power of individual devices has increased by a factor of more than 10^6 in the last 30 years and also the number of devices has grown at an even faster rate. Many people now own numerous devices in their smartphones, laptops, cars and household gadgets while the internet giants run vast server farms each of which must have a processor count running into millions. Likewise, for scientific computing the most powerful systems in the recent TOP500 list of supercomputers also have millions of processing cores. These developments, particularly in the last 15 years or so, have transformed society – and the process has only just begun, there is no sign that the trends shown in the figure are about to stop.

One hope I have in writing this book, is that learning about GPUs and parallel programming will be a small help in keeping up with the changes to come.

1.5 CPU Memory Management: Latency Hiding Using Caches

Data and instructions do not move instantly between the blocks shown in Figure 1.1; rather they progress clock-step by clock-step through various hardware registers from source to destination so that there is a *latency* between issuing a request for data and the arrival of that data. This intrinsic latency is typically tens of clock-cycles on a CPU and hundreds of clock-cycles on a GPU. Fortunately, the potentially disastrous performance implications of latency can be largely hidden using a combination of caching and pipelining. The basic idea is to exploit the fact that data stored in sequential physical memory locations are mostly processed sequentially in computer code (e.g. while looping through successive array elements). Thus, when one element of data is requested by the memory load unit, the hardware actually sends this element and a number of adjacent elements on successive clock-ticks. Thus, although the initially requested element may arrive with some latency, thereafter successive elements are available on successive clock-ticks. Thinking of data flows along the lines shown in Figure 1.1 just like water flows through pipes in your house is a valid comparison – when you turn on a hot tap, it takes a little while for the hot water to arrive (latency) and thereafter it stays hot.

In practice PCs employ a number of memory cache units to buffer the data streaming from multiple places in main memory as shown in Figure 1.4.

Note that in Figure 1.4 all three caches are on-chip and there are separate L1 caches for data and instructions. The main memory is off chip.

Program instructions are also streamed in a pipeline from main memory to the instruction fetch unit. This pipeline will be broken if a branch instruction occurs, and PC hardware uses sophisticated tricks such as *speculative execution* to minimise the effects. In speculative execution the PC executes instructions on one or more of the paths following the branch before the result of the branch is known and then uses or discards results once the branch path is known. Needless to say, the hardware needed for such tricks is complex.

https://doi.org/10.1017/9781108855273.002 Published online by Cambridge University Press

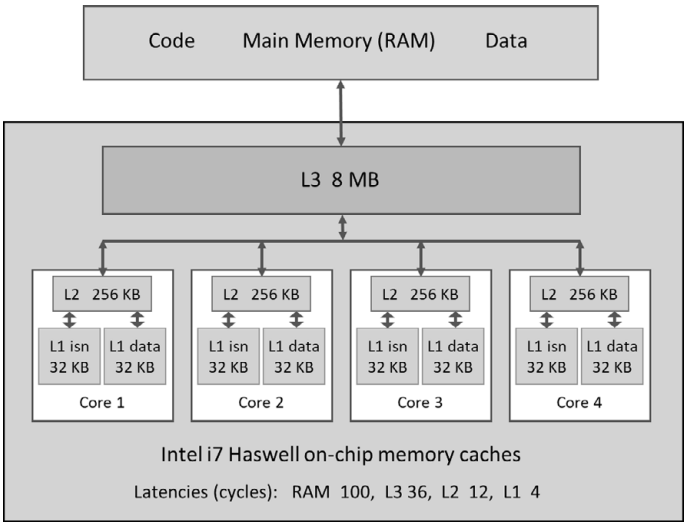

Figure 1.4 Memory caching on 4-core Intel Haswell CPU

The caching scheme shown in Figure 1.4 is typical for modern CPU multi-core chips. There are three levels of caching memories, all integrated into the CPU chip. First the level-3 (L3) cache is large at 8 MB and is shared by the 4 CPU cores. Each core also has progressively faster L2 and L1 caches, with separate L1 caches for data and instructions.

The hardware transfers cache data in packets called *cache lines* of typically 64 or 128 bytes. On current Intel processors the cache line size is 64 bytes, but the hardware usually transfers two adjacent lines at once giving an effective size of 128 bytes. The data in a cache line corresponds a contiguous region of the main memory beginning at an address that is a multiple of the cache line size.

1.6 CPU: Parallel Instruction Set

Before discussing the powerful parallel capabilities of GPUs, it is worth noting that Intel CPUs also have some interesting parallel capabilities in the form of vector instructions. These first appeared around 1999 with the Pentium III SSE instruction set. These instructions used eight new 128-bit registers each capable of holding 4 4-byte floats. If sets of 4 such floats were stored in sequential memory locations and aligned on 128-byte memory boundaries they can be treated as a 4-element vector and these vectors can be loaded to and stored from the SSE registers in a single clock-cycle and likewise enhanced ALUs could perform vector arithmetic in single clock-cycles. Thus, using SSE could potentially speed up some floating-point calculations by a factor of four. Over the years SSE has evolved and many recent Intel CPUs support AVX2 which uses 256-bit registers and can support many data types. Currently Intel's most recent version is AVX-512 which uses 512-byte registers capable of holding vectors of up to 16 floats or 8 doubles. The use of AVX on Intel CPUs is discussed in detail in Appendix D.

https://doi.org/10.1017/9781108855273.002 Published online by Cambridge University Press

1.7 GPU Architecture

In this section we give more details of the evolution of GPU computing in general and specifically on NVIDIA hardware.

1.7.1 First a Little Bit of History

GPUs were designed for high-performance computer graphics; in modern gaming a screen size of say 1920×1080 pixels refreshed at 60 Hz is normal. Each pixel needs to be computed from the instantaneous state of the game and the player's viewpoint – about 1.25×10^8 pixel calculations per second. While this might just about be possible with a modern processor, it certainly was not possible one or two decades ago when interest in PC gaming emerged. Note this is a massively *parallel* computing problem as the pixel values can all be calculated independently. Gaming cards emerged as dedicated hardware with a large number of simple processors to perform the pixel calculations. An important technical detail is that the pixel array representing the image is stored in a 2D array in a digital frame buffer as data, with typically 3-bytes per pixel representing the red, green and blue (RGB coding) intensities of the pixel. The frame buffer is specialised computer memory (video ram) that can be read or written as normal but has an additional port allowing dedicated video hardware to independently scan the data and send suitable signals to a monitor.

It was quickly noticed that an inexpensive card doing powerful parallel calculations and sending the results to computer memory might have applications beyond gaming and in around 2001 the acronym GPGPU (general purpose computing on graphics processing units) and website gpgpu.org were born. In 2007 NVIDIA launched their GPU programming toolkit which changed GPU programming from being a difficult niche activity to mainstream.

1.7.2 NIVIDA GPU Models

NVIDIA produces three classes of GPU:

1. The GeForce GTX, GeForce RTX or Titan branded models, e.g. GeForce GTX 1080; these are the least expensive and are aimed at the gaming market. Typically, these models have less FP64 support than the equivalent scientific versions and do not use EEC memory. However, for FP32 calculations their performance can match or exceed the scientific card. The Titan branded models are the most powerful and may have more capable GPUs. The RTX 3090 has the highest FP32 performance of any NVIDIA card released up to March 2021.
2. The Tesla branded models, e.g. Tesla P100; these are aimed at the high-end scientific computing market, have good FP64 support and use EEC memory. Tesla cards have no video output ports and cannot be used for gaming. These cards are suitable for deployment in server farms.
3. The Quadro branded GPUs, e.g. Quadro GP100; these are essentially Tesla model GPUs with added graphics capabilities and are aimed at the high-end desktop workstation market.

Between 2007 and the time of writing in 2020 NVIDIA have introduced 8 different GPU generations with each successive generation having more software features. The generations are named after famous scientists and within each generation there are usually several models which

https://doi.org/10.1017/9781108855273.002 Published online by Cambridge University Press

themselves may differ in software features. The specific capability of a particular GPU is known as its Compute Capability or CC, which is specified by a monotonically increasing number. The most recent generation is Ampere with a CC value of 8.0. The examples in the book were developed using a Turing RX 2070 GPU with a CC of 7.5. A fuller account can be found in Appendix A.

1.8 Pascal Architecture

NVIDIA GPUs are built up in a hierarchical manner from a large number of basic compute-cores, the arrangement for the Pascal generation GTX1080 is shown in Figure 1.4:

1. The basic unit is a simple compute-core; it is capable of performing basic 32-bit floating point and integer operations. These cores do not have individual program counters.
2. Typically, groups of 32 cores are clustered together form what NVIDIA call "32-core processing blocks" – I prefer to use the term "warp-engine". This is because, as explained in the software sections, in a CUDA kernel program the executing threads are grouped into 32-thread groups which NVIDIA calls "warps" which can be considered as the basic execution unit in CUDA kernel programs. All threads in a given warp run in lock-step executing the same instruction at every clock-cycle. In fact, all threads in a given warp are run on the same warp-engine which maintains a single program counter that is used to send a common instruction sequence to all the cores belonging to that warp-engine.[6]

   Importantly, the warp-engine adds additional compute resources shared by its cores. These include special function units (SFUs) which are used for fast evaluation of transcendental functions such as `sin` or `exp`, and double precision floating point units (FP64). In Pascal GPUs, warp-engines have eight SFUs and either 16 or one FP64 unit.
3. Warp-engines are themselves grouped together to form what NVIDIA calls symmetric multiprocessors or SMs. In the Pascal generation a GPU has either two (Tesla GP100 only) or four warp-engines (all others). Thus, an SM has typically 128 compute cores. The threads in a CUDA kernel program are grouped into a number of fixed size thread blocks. Each thread block in fact runs on a single SM, but different thread blocks may run on different SMs. This explains why threads in the same thread block can communicate with each other, for example using shared memory or `__syncthreads()`, but threads in different thread blocks cannot.

   The SM also adds texture units and various on chip memory resources shared equally between warp-engines, including a register file of 64K 32-bit words, shared memory of 96 KB and L1/texture cache of 24 KB or 48 KB.
4. Finally, a number of SMs are grouped together to make the final GPU. For example, the GTX 1080 has 20 SMs containing a total of $20 \times 128 = 2560$ compute-cores. Note all these cores are on the same hardware chip. An L2 cache of either 4 GB or 2 GB shared by all SMs is also provided at this level.

For the gaming market NVIDIA make a range of GPUs which differ in the number of SM units on the chip, for example the economy GTX 1030 has just 3 SM units. Gaming cards may also have different clock-speeds and memory sizes.

In Figure 1.5 a single compute core is shown on the left; it is capable of performing 32-bit floating point or integer operations on data which flows through the core. The hardware first

https://doi.org/10.1017/9781108855273.002 Published online by Cambridge University Press

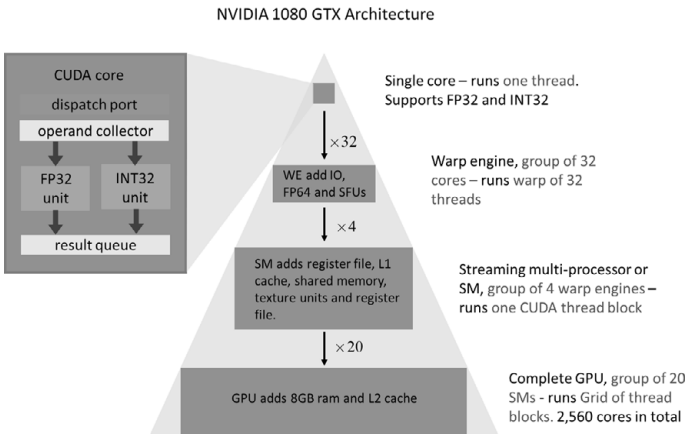

Figure 1.5 Hierarchical arrangement of compute cores in an NVIDIA GTX1080

groups the cores into "warps" of 32 cores processed by a "Warp Engine" (WE) which adds shared resources including IO, double precision units (either 1 or 16), and eight special function units (SFUs). The SFUs evaluate 32-bit transcendental functions such as sin or exp. All the threads in any particular CUDA warp run together on a single warp engine. Small groups of WEs, usually two or four, are then grouped together into one SM unit. Finally, multiple SMs are grouped together to make the GPU.

1.9 GPU Memory Types

GPU memory is also organised in a hierarchical fashion similar to cores, with the main GPU memory at the bottom of the pyramid and various specialised memories and caches above. This is also indicated in Figure 1.5. The memory types are as follows:

- **Main memory:** This is analogous to the main memory of a CPU; the program itself and all data resides here. The CPU can write and read data to and from the GPU main memory. These exchanges go via the PCI bus and are relatively slow and therefore CUDA programs should be designed to minimise data transfers. Helpfully data in the GPU main memory is preserved between kernel calls so that it can be reused by successive kernel calls without the need to reload. It is also possible for memory transfers to proceed in parallel with kernel execution (i.e. asynchronous transfers), this can be helpful in data processing tasks such as manipulating frames of a movie, where the next frame can be transferring to the GPU while the present frame is being processed. Notice that because texture and constant data are stored in the GPU main memory they can be written to by the CPU even though they are read only on the GPU.
- **Constant Memory:** A dedicated 64 KB of GPU main memory is reserved for constant data. Constant memory has a dedicated cache which bypasses the L2 cache so if all threads from a warp read the same memory location this can be as fast as if the data was in a register. In practice recent versions of the NVCC compiler can often detect this situation in your code and automatically place variables in constant memory. The use of `const` and `restrict` where appropriate will be helpful. It is probably not worthwhile worrying too

https://doi.org/10.1017/9781108855273.002 Published online by Cambridge University Press

much about explicitly using constant memory. Notice also that this memory is quite limited and therefore not useful for large tables of parameters.

- **Texture memory:** This feature is directly related to the graphics processing origins of GPUs. Texture memory is used to store arrays of up to three dimensions and is optimised for local addressing of 2D arrays. They are read only and have their own dedicated caches. Textures are accessed by the special lookup functions, `tex1D`, `tex2D` and `tex3D`. These functions are capable of performing very fast 1D interpolation, 2D bilinear interpolation or 3D trilinear interpolation for arbitrary input positions within the texture. This is a massive help for image processing tasks and I highly recommend using texture lookup wherever helpful. A number of our examples will illustrate how to use textures.

Recent versions of CUDA support additional texture functionality, including layered textures (indexable stacks of 1D or 2D textures) and surfaces which can be written to by the GPU.

- **Local memory:** These are memory blocks private to each individual executing thread; they are used as overflow storage for local variables in intermediate temporary results when the registers available to a thread are insufficient. The compiler handles the allocation and use of this resource. Local memory is cached via the L2 and L1 caches just like other data.
- **Register file:** Each SM has 64K 32-bit registers which are shared equally by the thread blocks concurrently executing on the SM. This can be regarded as a very important memory resource. In fact there is a limit of 64 on the maximum number of concurrent warps (equivalent to 2K threads) that can execute on a given SM. This means that if the compiler allows a thread to use more than 32 registers, the maximum number of thread blocks running on the SM (i.e. occupancy) is reduced, potentially harming performance. The NVCC compiler has a switch, `--maxrregcount <number>`, that can be used to tune overall performance by trading occupancy against thread computational performance.
- **Shared memory:** Each SM provides between 32 KB and 64 KB of shared memory.[7] If a kernel requires shared memory the size required can be declared either at kernel launch time or at compile time. Each concurrently executing thread block on an SM gets the same size memory block. Thus if your kernel requires more than half of the maximum amount of shared memory, the SM occupancy will be reduced to a single thread block per SM. Realistic kernels would usually ask for no more than half that available memory.

Shared memory is important because it is very fast and because it provides the best way for threads within a thread block to *communicate* with each other. Many of our examples feature the use of shared memory.

Many early CUDA examples emphasise the faster memory access provided by shared memory compared to the poor performance of the (then poorly cached) main memory. Recent GPUs have much better memory caching so using shared memory for faster access is less important. It is important to balance the performance gain from using shared memory against reduced SM occupancy when large amounts of shared memory are needed.

Current GPUs use their L1 and L2 caches together with high occupancy to effectively hide the latency of main memory accesses. The caches work most effectively if the 32 threads in a warp access 32-bit variables in up to 32 adjacent memory locations and the starting location is aligned on a 32-word memory boundary. Using such memory addressing patterns is called *memory coalescing* in CUDA documentation. Early documentation places great emphasis on

https://doi.org/10.1017/9781108855273.002 Published online by Cambridge University Press

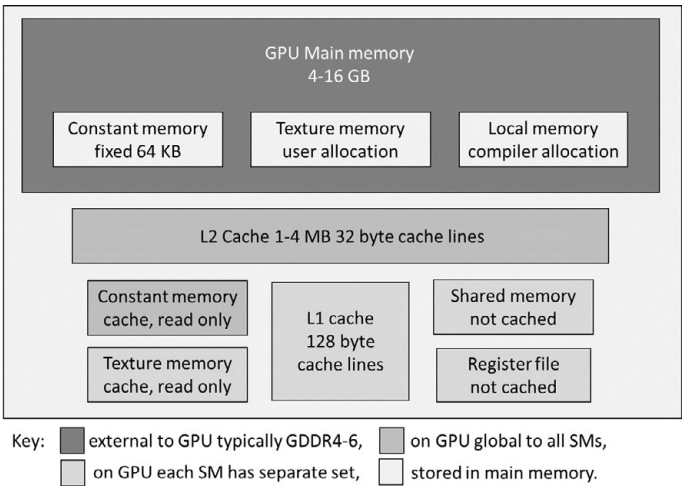

Figure 1.6 GPU memory types and caches

this topic because early GPUs had poor or no caching. Modern GPUs are more forgiving so you probably just need to stick to the golden rule: adjacent threads (or indices on CPUs) address adjacent memory locations. The memory arrangement is shown in Figure 1.6.

1.10 Warps and Waves

The GPU architecture is reflected in the way a CUDA kernel is designed and launched by host software. Designing good kernels to match particular problems requires skill and experience and is essentially what the rest of this book is about. When you get it right, it is very satisfying to see your code speed up by a large factor. First you have to decide how many threads, `Nthreads`, to use. Choosing a good value for `Nthreads` is one of the most important choices you make when designing CUDA kernels. The choice of `Nthreads` is of course problem specific. For our example for the `gpusum` program in Example 1.3 we used a value of `Nthreads` equal to the number of steps. So for 10^9 steps we used 10^9 threads which is huge compared to the eight that can be run in parallel on our CPU. In later chapters we will say a lot about image processing. To process a 2D image having `nx` × `ny` pixels, a good choice is to put `Nthreads = nx` × ny. The key point is that `Nthreads` should be *big*, arguably as big as possible.

If you are new to CUDA you might expect that setting `Nthreads` equal to `Ncores`, the number of cores in your GPU, would be enough to keep the GPU fully occupied. In fact, this is far from correct; one of the very neat features on NVIDIA GPUs is that the hardware can hide the latencies in memory accesses or other hardware pipelines by rapidly switching between threads and use the data for particular threads as soon as it becomes available.

To be specific the GPU used for most of our examples is a RTX 2070 which has 36 SM units (`Nsm` = 36) and each SM has hardware to process two warps of 32 threads (`Nwarp` = 2). Thus, for this GPU `Ncores = Nsm × Nwarp × 32` = 2304. What is less obvious is that during kernel processing each SM unit has a large number of *resident* threads, `Nres`, for

https://doi.org/10.1017/9781108855273.002 Published online by Cambridge University Press

the RTX 2070; `Nres` =1024 equivalent to 32 warps. At any given instant two of these 32 warps will be *active* and the remainder will be suspended, possibly waiting for pending memory requests to be satisfied. This is how the latency hiding is implemented in NVIDIA hardware. When we launch a kernel with say 10^9 threads these threads are run in *waves* of `Nwave = Nres × Nsm` threads; this is actually 27127 waves on the 2070 GPU with `Nwave` = 36864, with the last wave being incomplete. Ideally the minimum number of threads in any kernel launch should be `Nwave`, and if more threads are possible it should be a multiple of `Nwave`.

Note that although the Turing generation of GPUs have `Nres` = 1024 this is unusual; all the other recent NVIDIA GPU generations have `Nres` = 2048, twice the Turing value. Since for these GPUs `Nwarp = 2` is the same as for Turing, `Nwaves` will be twice the Turing value. Note that for any particular GPU generation `Nsm` will vary with model number, e.g. `Nsm` = 46 for the RTX 2080 GPU, but `Nwarp` will not. Thus, `Nwave` will vary like `Nsm` with GPU model for a given GPU generation.

1.11 Blocks and Grids

In CUDA the thread block is a key concept; it is a group of threads that are batched together and run on the same SM. The size of the thread block should be a multiple of the warp size (currently 32 for all NVIDIA GPUs) up to the hardware maximum size of 1024. In kernel code, threads within the same thread block can communicate with each other using shared or global device memory and can synchronise with each other where necessary. Threads in different thread blocks cannot communicate during kernel execution and the system cannot synchronise threads in different thread blocks. Note it is common for the thread block size to be a sub multiple of 1024; often it is 256. In that case warps from up to four (or eight for non-Turing GPUs) different thread blocks will coexist on the SMs during kernel execution. Even though these blocks may coexist on the same SM they will still not be able to communicate.

When we launch a CUDA kernel we specify the *launch configuration* with two values, the thread block size and the number of thread blocks. The CUDA documentation refers to this as launching a *grid* of thread blocks and the grid-size is just the number of thread blocks. In our examples we will consistently use the variable names `threads` and `blocks` for these two values. Thus, the total number of threads is specified implicitly as `Nthreads = threads × blocks`. Notice `Nthreads` must be a multiple of `threads` the thread block size. If the number of threads you actually want our kernel to run is N, then `blocks` must be large enough so that `Nthreads ≥ N`. This is the purpose of line 19.2 in Example 1.3; it is to round up `blocks` to ensure there will be at least `steps` threads. Because of rounding up we must include an out-of-range check in the kernel code to prevent threads of rank `≥ N` from running. That is the purpose of the test in line 15.4 of Example 1.3.

The CUDA documentation does not explicitly talk about waves very much; about the only reference we found was in the 2014 blog post by Julien Demouth at https://developer.nvidia.com/blog/cuda-pro-tip-minimize-the-tail-effect/. This is interesting because it implies that threads are dispatched to SMs for execution in complete waves when possible, which in turn means that it is important for `blocks` to be a multiple of the number of SMs on the GPU being used.

https://doi.org/10.1017/9781108855273.002 Published online by Cambridge University Press

Table 1.1 *CUDA built-in variables. NB the first 4 are structs containing x, y and z members*

variable	comment
`threadIdx`	`id = threadIdx.x` is thread rank in thread block `id = blockDim.x*blockIdx.x+threadIdx.x` is thread rank in grid
`blockIdx`	`blockIdx.x` is block rank in grid of blocks
`blockDim`	`blockDim.x` is number of threads in one block
`gridDim`	`gridDim.x` is number of blocks in the grid `threads = gridDim.x*blockDim.x` is total number of threads in launch
`warpSize`	Number of threads in warp, set to 32 on all current GPUs

1.12 Occupancy

NVIDIA define occupancy as the ratio of the number of threads actually resident in the SM units compared to the maximum value `Nres`. Occupancy is usually expressed as a percentage. Full occupancy of 100 per cent is the same as saying that complete waves are running on the SMs of the GPU.

Even if we launch a kernel with sufficient threads to achieve 100 per cent occupancy we might not actually achieve full occupancy. The reason for this is that each SM has a limited total shared memory size and a limited number of registers. If our thread block size is 256 then full occupancy will only be achieved if four (or eight) threads bocks are resident on each SM which reduces the resources available to each thread block by the same factor. NVIDIA GPUs have enough registers for each thread to use up to 32 registers while maintaining full occupancy. Shared memory is more difficult as it is typically limited to 64 or 96 KB per SM which is equivalent to only 32 or 48 bytes per thread at full occupancy for non-Turing GPUs. On the latest Ampere GPUs this is increased to 80 bytes.

Less than full occupancy is not necessarily bad for performance, especially if the kernel is compute bound rather than memory bound, you may have to accept lower occupancy if your kernel needs significant amounts of shared memory. Experimentation may be necessary in these cases; using global memory instead of shared memory and relying on L1 caching for speed may by a good compromise on modern GPUs.

Kernel code can use the built-in variables shown in Table 1.1 to determine the rank of a thread in its thread block and in the overall grid. Only the 1D case is shown in this table the 2D and 3D cases are discussed in the next chapter.

This is the end of our introductory chapter. In Chapter 2 we introduce the more general ideas behind parallel programming on SIMD machines and GPUs. We then give some more detailed examples, including the classic problem of parallel reduction. We also discuss kernel launches in more detail.

Endnotes Chapter 1

1 For example, the Nvidia plugin for Visual Studio C++ helpfully generates a sample program to get you started but unfortunately that program is full of `goto` statements. The use of `goto` has of course been deprecated since the early 1980s.

2 Both the Nvidia documentation and their example code mostly refer to the CPU as the *host* and to the GPU as the *device*. We often but not always follow this convention.

https://doi.org/10.1017/9781108855273.002 Published online by Cambridge University Press

3 Later we will discuss reduce operation in some detail as an example of a parallel primitive. Our best CUDA code for this operation will turn out to be a bit faster than using thrust.
4 Computers which store instructions and data in a common memory are said to have *von Neumann architecture*, after the physicist John von Neumann who worked on the design of the early 1945 ADVC machine. Arguably the idea of treating computer instructions as data can also be credited to Ada Lovelace in the 1840s. The alternative *Harvard architecture* uses separate hardware to store data and instructions. Examples include the Colossus computers, used at Bletchley Park from 1943, which were programmed using switches and plugs. Paper tape could also be used to hold instructions. In a curious case of nominative determinism, the English mathematician, Max Newman played an important role in the design of Colossus – truly these "new men" ushered in our digital age. Today Harvard architecture is still used for specialised applications, e.g. embedded systems running fixed programs stored in read only memory units (ROMs).
5 Unfortunately, the acronym PC for program counter is the same as for personal computer. Actually, the former use predates the introduction of personal computers by at least 20 years, thus we will use PC for both. This should not be confusing as we will rarely use PC for program counter.
6 Recently the Volta/Turing generation of GPUs launched in 2017 has relaxed this restriction somewhat. This is discussed later as an advanced topic.
7 This depends on the compute capability of a particular device. Most devices can be configured to have at least 48 KB.

https://doi.org/10.1017/9781108855273.002 Published online by Cambridge University Press