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.

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

```
#include <stdio.h>
#include <stdlib.h>
#include "cxtimers.h"

inline float sinsum(float x, int terms)
{
  // sin(x) = x - x^3/3! + x^5/5! ...
  float term = x;    // first term of series
  float sum  = term; // sum of terms so far
  float x2   = x*x;
  for(int n = 1; n < terms; n++){
    term *= -x2 / (float)(2*n*(2*n+1));
    sum += term;
  }
  return sum;
}

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

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

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

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

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 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 ~6×10^{-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"
```

```
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`,

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.

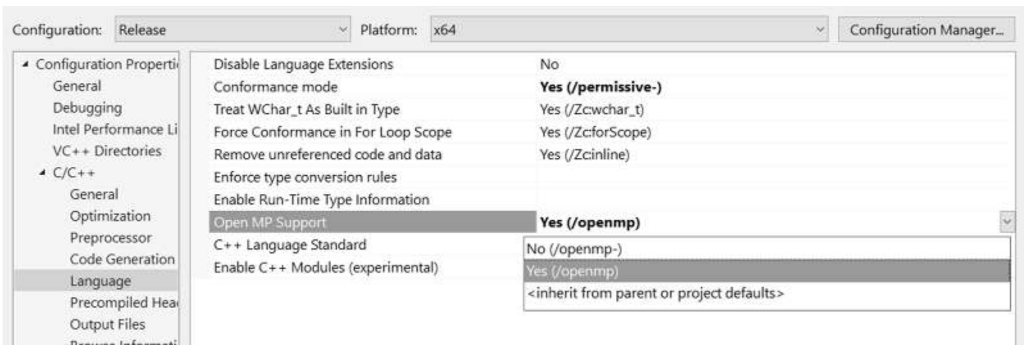
- Line 32: Here we have simply modified the existing `printf` to also output the value of `threads`.

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.



Configuration: Release Platform: x64

Disable Language Extensions	No
Conformance mode	**Yes (/permissive-)**
Treat WChar_t As Built in Type	Yes (/Zc:wchar_t)
Force Conformance in For Loop Scope	Yes (/Zc:forScope)
Remove unreferenced code and data	Yes (/Zc:inline)
Enforce type conversion rules	
Enable Run-Time Type Information	
Open MP Support	**Yes (/openmp)**
C++ Language Standard	No (/openmp-)
Enable C++ Modules (experimental)	Yes (/openmp)
	<inherit from parent or project defaults>

Figure 1.1 How to enable OpenMP in Visual Studio

Example 1.3 gpusum GPU calculation of a sin integral

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

__host__ __device__ inline float sinsum(float x, int terms)
{
  float x2 = x*x;
  float term = x;   // first term of series
  float sum = term; // sum of terms so far
  for(int n = 1; n < terms; n++){
    term *= -x2 / (2*n*(2*n+1));  // build factorial
    sum += term;
  }
  return sum;
}

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

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

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

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 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]

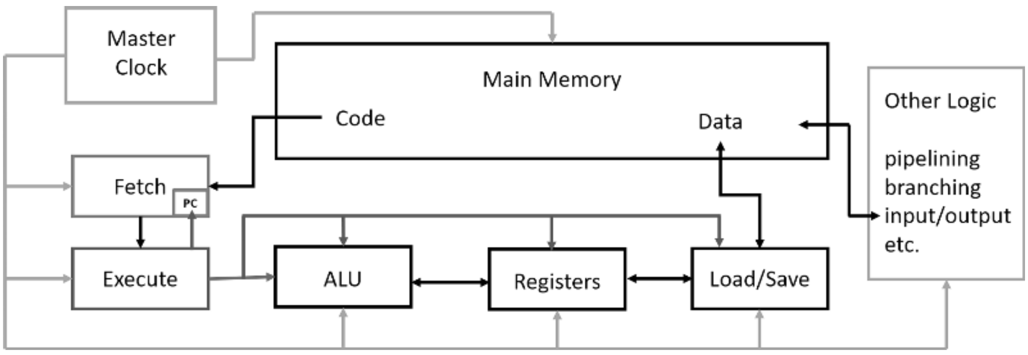


Figure 1.2 Simplified CPU architecture