2

Thinking and Coding in Parallel

Computers have always been very good at giving users the impression that they can perform multiple tasks at the same time. For example, even a single core PC will allow you to browse the web while running a lengthy calculation in the background. However, this is accomplished by fast task switching – the user gets CPU cycles when active, otherwise the CPU cycles are given to the calculation. This is resource sharing not true parallel programming.

If you have a more recent 4-core PC, you might launch four instances of your lengthy calculation with different parameter values to genuinely perform parallel computation without any extra programming effort. Problems that can be solved with this approach are sometimes called "*trivially parallel*". In spite of the name, this approach is perfectly valid if it gets your job done effectively and has the enormous advantage of requiring little extra programming effort. If the potential number of jobs is large, a simple script file might be useful to automate the launching of new jobs and collecting their results. A nice example is the CERN data centre which has more than 200,000 cores mostly running event data processing or Monte Carlo simulation using the same program but different event data or different random numbers.

Unfortunately, the trivial programming approach does not work on GPUs which have very simple processing cores designed to work together on a single task. True parallel programming requires just this – many processing cores working together to complete a single task. It turns out that writing effective parallel code is often rather straightforward – as hopefully we demonstrate in this book.

2.1 Flynn's Taxonomy

Computer scientists recognise a small number of serial and parallel computer architectures described by 4-letter acronyms summed up in Flynn's taxonomy shown in Table 2.1.[1]

The first, SISD case, represents a "normal" single processor running a single thread. Computers with this architecture can still be relatively fast and by employing rapid switching between tasks they can give human beings the illusion that they are multitasking but in fact they only execute one operation on one data item in a clock-cycle.

The second, SIMD case, covers architectures where the hardware can execute the same instruction on multiple data items at the same time. This can be achieved by have multiple ALUs fed with different data items but using a common instruction decoder. To overcome memory access bottlenecks the data items are fed to the ALUs in a vector format. Hence these architectures are often known as vector processors. The fondly remembered CRAY supercomputers, dating from the 1970s, were early and very effective examples of this approach. In 1999 Intel CPUs introduced their so-called Streaming SIMD Extensions (SSE)

https://doi.org/10.1017/9781108855273.003 Published online by Cambridge University Press

Table 2.1 *Flynn's taxonomy*

acronym	Name	comment
SISD	Single Instruction Single Data	Single-core system running one thread.
SIMD	Single Instruction Multiple Data	Multiple processors running the same task on multiple data streams.
MIMD	Multiple Instruction Multiple Data	Multi-core system with each core running a different task, also multiple connected systems.
MISD	Multiple Instructions Single Data	Rare, possible application in fault tolerant designs.
SIMT	Single Instruction Multiple Threads	Variation on SIMD implemented in CUDA.

instruction set into the Pentium III architecture; these instructions could perform operations on vectors of four 32-bit floating point numbers using what were effectively 128-bit registers. Over the years the capabilities of Intel SIMD operations have increased; currently Intel supports advanced vector extensions using 512-bit registers (AVX-512), enough for up to 16 32-bit numbers. We discuss this topic in more detail in Appendix D.

The third, MIMD, is effectively just a set of separate CPUs performing separate tasks. This case includes both modern multicore PCs running Linux or Windows and clusters of PCs connected by a network. In both cases suitable software, for example, MPI or OpenMP, can be used to allow the multiple independent processors to work together on a single computing task.

The fourth, MISD, is included for the sake of completeness and is rarely used. It might be used in specialised embedded systems requiring redundancy against failure; for example, satellites.

The final, SIMT, was introduced by NVIDIA as a variation of SIMD to describe their GPU architecture. Although both are used to tackle similar scientific computations, there are differences between them. In the SIMD model a relatively small number of threads use vector hardware to process data. In the SIMT model a large number of threads are used to process individual data items. If a common instruction is used by all threads then SIMD behaviour is replicated, but the SIMT architecture also permits threads to perform divergent operations which, while it may lead to a drop in performance, also allows for more versatile code. In the recent Volta and Turing generations of GPU, NVIDIA have extended the capabilities for programming individual threads.

It is the SIMD/T case that is of interest for parallel programming. We look for sections of our code where the same operation is performed on multiple data items – obvious candidates are `for` loops. However, if we want to share the computation of a loop across multiple threads it is important that there are no dependencies between passes through the loop, for example the order in which the loop traversals are executed should not matter.

Consider again the loop in the Example 1.1

```
double sum_host = 0.0;
for (int step = 0; step <= steps; step++){
   float x = step_size*step;
   sum_host += sinsum(x, terms);
}
```

The loop statements shown in lines 25–26 are independent of other passes through the loop, because the order in which we sum the values in the variable `sum_host` does affect the final result.[2] Thus, this loop is a good candidate for parallel code, particularly so if the evaluation of the function `sin_host` is computationally expensive. Before we proceed there is a subtle technical problem to resolve. Either the variable `sin_host` must be global – therefore, visible to all the threads participating in the parallel calculation or some other means must be found to get the correct final sum.

Making `sin_host` global to all threads, while straightforward to implement, introduces yet another complication – if two or more threads try to update the variable simultaneously the result will be undefined! With CUDA, one thread will succeed and the attempts by other threads at simultaneous update will be ignored, so the final answer will be wrong. There is a fix for this problem, which is actually a generic issue for all parallel computing platforms; it is to use so-called *Atomic* operations to perform the required operation serially. Atomic operations are usually implemented by calling platform specific functions, and their use in CUDA code is discussed in Appendix B. For now we note that using atomics might slow down a calculation and we choose an alternative approach, which is to simply store the individual values returned by the `sinsum` function in separate elements of a large array. The elements of the array will be summed together in a separate step once they have all been calculated. This is an example of parallel thinking; we separate a serial loop into a part that can be done in parallel – calling the `simsum` many times, and a part which cannot be done in parallel – the reduce operation of adding up all the individual stored values. These two steps are the only parts of the calculation that will be done on the GPU; code running on the CPU takes care of everything else.

Now it is time to look in more detail at our first CUDA program emphasising the steps necessary to convert the serial version in Example 1.1 to the parallel version in Example 1.3.

The first step is to add the header files needed for CUDA

```
#include "cuda_runtime.h"           // cuda basic
#include "thrust/device_vector.h"  // thrust device vectors
```

The headers added are `cuda_runtime.h` which provides basic support for CUDA and `thrust/device_vector.h` provides a container class like `std::vector` for 1D arrays in GPU memory. Most of the examples in this book use these headers.

We then convert the function `sinsum` into a function that can be used on both the CPU and the GPU. This is simply done be adding a CUDA keyword to the function declaration.

```
__host__ __device__ inline float sinsum(float x, int terms)
```

The keyword `__device__` tells the compiler to compile a version of the function that runs on the GPU and can be called by kernels and other functions running on the GPU. Likewise, `__host__` tells the compiler to create a version of the function for the CPU code to use. The inline keyword is part of standard C++ and tells the compiler to generate function code embedded in the caller's code, removing the overhead of a function call at the price of increasing the size of the final exe file. In CUDA inline is the default for __device__ functions. The __host__ keyword is only needed if the function is to be used on both the host and device; for device only functions just __device__ is needed. The entire body of the function in lines 6–15 is unchanged. This is a very powerful feature of CUDA.

It is also possible to have two different versions of a function, one declared with __device__ and one declared with __host__. The __host__ prefix could be omitted from the host version as this is the default, but we recommend using it to make your intentions clear. Obviously this prefix is not needed (or recommended) for functions which are only used by the host.

The __device__ version of sinsum is simply a GPU function and is not callable directly from the host. We need to write a separate CUDA kernel function which runs on the GPU and can be called from the host. CUDA kernels are declared using __global__ instead of __device__ this reflects their dual nature – callable by the host but running on the GPU. In the CUDA world people talk about "launching" kernels rather than "calling" them so that is what we shall do from now on. Our first kernel gpu_sum in lines 15.1–15.8 is all new code which replaces most of lines 23–27 in the original program.

```
15.1 __global__ void gpu_sin(float *sums, int steps, int terms,
                                          float step_size)
15.2 {
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 values
15.7   }
15.8 }
```

The kernel declaration in line 15.1 looks very much like a normal C++ declaration except for the prefix __global__. There are, however, some restrictions based on the fact that although the kernel is called from the host it cannot access any memory on the host. All kernels must be declared void and their arguments are restricted to scalar items or pointers to previously allocated regions of device memory. All kernel arguments are passed by value. In particular, references are not allowed. It is not a good idea to try and pass large C++ objects to kernels; this is because they will be passed by value and there may be significant copying overheads. Also any changes made by the kernel will not be reflected back in the host's copy after the kernel call. Additionally, any C++ classes or structs passed to a kernel must have __device__ versions of *all* their member functions.

- Line 15.3 declares a variable step equivalent to the for loop index variable of the same name in line 24 of Example 1.1. It is set to a value defined by the built-in variables blockDim.x, blockIdx.x and threadIdx.x. The values of these variables depend on the launch parameters used in the host call to the kernel as follows:

  ◦ `blockDim.x` will be set to `threads`, i.e. the thread block size used by the kernel.
  ◦ `blockIdx.x` will be set to the rank of the thread block to which the current thread belongs and will be in the range `[0,blocks-1]`.
  ◦ `threadIdx.x` will be set to the rank of the current thread within its thread block and will be in the range `[0,threads-1]`.
  ◦ `step = blockDimx.blockIdx.x+threadIdx.x` is in range `[0, threads × blocks - 1]`.

The key point is that the system will run `threads × blocks` instances of the kernel on the GPU covering all possible combinations of values for `threadIdx` and `blockIdx`. Thus, when we look at a kernel listing we must imagine that we are looking at the contents of a loop which is executed for all possible values of these built-in variables. In this case `step` takes all values in the range `[0,size-1]` where `size = threads × blocks`. When looking at a kernel code, you must imagine that the code is being run simultaneously by all the threads. Once you have mastered this concept, you will be a parallel programmer![3]

- Line 15.4: This is an out-of-range check on the value of `step`, the kernel will exit at this point for threads that fail the check.
- Line 15.5: Calculate the `x` value corresponding to `step`.
- Line 15.6: Call `sinsum` with the thread dependant value of `x`. The result is stored in the array `sums` using `step` as an index.
- Line 15.7: The kernel exits at here; recall that `return` statements are not required for `void` functions in C++.

The changes to the main routine are as follows:

```
19.1 int threads = 256;
19.2 int blocks = (steps+threads-1)/threads; // round up
```

- Lines 19.1–19.2: The two lines are added to define the kernel launch configuration parameters `threads` and `blocks`. In this our first example, we use a fixed value of 256 for threads and a calculated value for `blocks` which is set to be just big enough to get the total number of threads in the launch to satisfy `threads × blocks ≥ steps`.

```
21.1 thrust::device_vector<float> dsums(steps); // GPU buffer
21.2 float *dptr = thrust::raw_pointer_cast(&dsums[0]);
```

- Line 21.1: This line creates the array `dsums` of size `steps` in the device memory using the thrust `device_vector` class as a container. By default the array will be initialised to zeros on the device. This array is used by the `gpu_sin` kernel to hold the individual values returned by calls to the `sinsum` function.

- Line 21.2: We cannot pass `dsums` to the kernel directly as thrust was not designed to make this possible,[4] but we can pass a pointer to the memory array managed by the class. For `std::vector` objects, the member function `data()` does this job. While this function does work for thrust `host_vector` objects it does not work for `device_vector` objects. Therefore we have to use the more complicated `cast` shown in this line. As an alternative you could instead use the undocumented `data().get()` member function of `device_vectors`.

```
22.1 gpu_sin<<<blocks,threads>>>
                (dptr,steps,terms,(float)step_size);
```

- Line 22.1: This line shows our first CUDA kernel launch; this is basically just a function call with a weird extra bit `<<<blocks, threads>>>` inserted between the function name and its argument list. The `int` variables that appear here specify the number of threads in each thread block (`threads`) and the number of thread blocks (`blocks`). Note these values can be defined at *run time* and if you have multiple kernel launches in your code each launch can use different values. As discussed above, `threads` should be a multiple of 32 and has a maximum allowed value of 1024 for all current GPUs.[5] The second parameter `blocks` should be large. These values affect the performance of the kernel and "tuning" them to get the best performance involves trying different combinations and choosing the combination that runs the kernel fastest. To aid this in most of our subsequent example code we will make these launch parameters user settable command line parameters. For most kernels a good starting point is `<<<4*Nsm, 256>>>` where `Nsm` is the number of SMs on the target GPU.[6] In this book we often use `<<<288, 256>>>` as our GPU has 36 SM units. For testing or debugging purposes using `<<<1, 1>>>` is sometimes interesting and is allowed. That version has the effect of running just a single thread on one SM unit.

```
22.2 double gpu_sum =
                thrust::reduce(dsums.begin(),dsums.end());
```

- Line 22.2: Here we use the host callable reduce function in the thrust library to sum all the elements of the array `dsums` in GPU memory. This call involves two steps, firstly we perform the required additions on the GPU and secondly we copy the result from GPU memory to CPU memory. This is often referred to as a D2H (device to host) transfer.

That is the end of our detailed description of our first kernel. We have deliberately kept this code as simple as possible.

It is worth looking at line 15.1 of the code in more detail. For any particular thread at line 15.1 of the kernel function the CUDA system variable `blockIdx.x` is set to the number of the currently executing thread block and the variable `threadIdx.x` is set to the rank of that current thread within its thread block. Thus, in the present case where `steps` is set to 10^9 and threads is set to 256, `blocks` will be set to 3906251 and `blockIdx.x` will be in the range [0,3906250], `threadIdx.x` will be in [0,255] and `blockDim.x` will be 256. Thus, for each thread the calculation, line 15.1 produces a unique value for `step` in the range 0–10^9 - 1. This is exactly what we need to replicate the behaviour of the original `for` loop from Example 1.1. You will find something like line 15.1 in every CUDA kernel function – it allows each thread to determine what specific task it has to do. In fact, the variable `step` is nothing more than a unique thread id; this is often referred to as the *rank* of the thread. NVIDIA also use the term *lane* to refer to the rank of a thread within its particular 32-thread warp.

There is another important point to make about line 15.1. The GPU hardware allocates all the threads in any particular thread block to a single SM unit on the GPU, and these threads are run together very tightly on warp-engines as warps of 32 threads. The variable `threadIdx.x` is set so that threads in the *same* warp have *consecutive* values of this variable; specifically `threadIdx.x%32` is the rank or lane of a thread within its warp (range 0–31) and `threadIdx.x/32` is the rank of the warp within the thread block (range 0–7 in our case). Thus, in line 15.6 of the kernel where we store a value in `sums[step]`, the adjacent threads within a given warp have adjacent values of `step` and so they will address adjacent memory locations in the array `sums`. This is vital to make efficient use of the GPU memory caching. Had we not known this we might have used the formula:

```
step = threadIdx.x*gridDim.x+blockIdx.x;
```

in line 15.1. Since the variable `gridDim.x` is set to the number of thread blocks the alternative would have given us the same range of values for the variable `step` but now adjacent hardware threads have values of `step` separated by 3906251 – resulting in a serious loss of memory performance.

In case you were wondering about the .x decoration, these variables are actually all `dim3` structs defined by the CUDA SDK (in vector_types.h). The type `dim3` is a struct with three `const uint` members x, y and z. In the next example we will see how they are used in 3D grids of threads to best fit the needs of a particular problem. The CUDA SDK defines a number of structs like this with up to four elements. For example, `uint3` is similar to `dim3` except that `uint3` defines overloaded operators to support basic arithmetic operations whereas `dim3` does not. Also, `dim3` has a default constructor that initialises all unspecified values to one so that it is always safe to use all components of the variables even if the user has not explicitly set them. Although we used simple integers for our kernel launch, the compiler will silently and safely promote them to type `dim3` for the actual kernel launch.

The values of threads and blocks are defined in lines 19.1 and 19.2 of the example to ensure that there are at least `steps` threads so that we use a separate thread for each call to `sinsum`. This may well be optimal here because the `sinsum` function does a great deal of computation. But this code gives the user no chance to experiment to find out if that is true. A more general approach is to allow the user to specify values for threads and blocks and to modify the `gpu_sin` kernel to allow individual threads to make more than one call to `gpu_sin` if necessary. Both modifications are very straightforward as shown in Example 2.1.

Example 2.1 Modifications to Example 1.3 to implement thread-linear addressing

```
. . .
15.1 __global__ void gpu_sin(float *sums, int steps,
                             int terms, float step_size)
15.2 {
15.3   int step = blockIdx.x*blockDim.x+threadIdx.x; // ID
15.4   while(step<steps){
15.5     float x = step_size*step;
15.6     sums[step] = sinsum(x,terms);  // store value
15.65    step += gridDim.x*blockDim.x;  // grid size stride
15.7   }
15.8 }
. . .  // NB ternary operator (test) ? a : b used here
19.1   int threads = (argc > 3) ? atoi(argv[3]) : 256;
19.2   int blocks  = (argc > 4) ? atoi(argv[4]) :
                              (steps+threads-1)/threads;
       . . .
```

Our modifications to the kernel are changing the `if` in line 15.4 to `while` and inserting an extra line `15.65` at the end of the while loop. In line 15.65 we increment `step` by the total number of threads in the grid of thread blocks. The while loop will continue until steps values have been calculated for all (non-zero) user supplied values of `blocks` and `threads`. Moreover, and importantly for performance reasons, on each pass through the `while` loop adjacent threads always address adjacent memory locations. Other ways of traversing through the data could be devised but the one shown here is the simplest and best. This technique of using a while loop with indices having a grid-size stride between passes through the loop is called "thread-linear addressing" and is common in CUDA code. It should always be considered as an option when porting a loop in host code to CUDA.

The added lines 19.1 and 19.2 of the main routine now also use the C/C++ ternary operator `(?:)` and set the values of threads and blocks according to whether or not the user has supplied extra command line arguments. If the user does not specify these arguments then `argc` will be set to three or less and both tests will fail so both default values (after the :) will be used. If the user specifies just one extra argument argc will be set to four so the first test will succeed and `threads` will be set using the expression before the : which will be the user supplied value. The second test will still fail and a default value for `blocks` will be used in the calculation. Finally, if the user supplies both extra arguments then both `threads` and `blocks` will be set using the user's values.

We confess to being ambivalent about the `(?:)` operator; it is very terse and was introduced in the early days of C in the 1970s when it was desirable to minimise keystrokes on heavy mechanical teletype machines when inputting code. Careless use of this operator can make code hard to read. However, crucially it *returns a value* whereas `if` statements do not. Using `(?:)` allows us to declare and initialise a variable in the *same statement* which is in line with modern C++ RAII practices. In our view this trumps the terse syntax of the operator. We do use it in our examples when initialising a variable to one of two alternatives.

One drawback of this approach to reading command line arguments for setting program options is that the user has to know the order in which the options are defined and cannot set a given option without also specifying all the previous options. For production code we would of course recommend something better.

For thread-linear addressing it is possible to replace the `while` loop Example 2.1 with a `for` loop as shown in Example 2.2.

Example 2.2 `gpu_sin` kernel alternative version using a for loop

```
. . .
__global__ void gpu_sin(float *sums, int steps,
                        int terms, float step_size)
{
  for(int step = blockIdx.x*blockDim.x+threadIdx.x;
               step<steps; step += gridDim.x*blockDim.x){
    float x = step_size*step;
    sums[step] = sinsum(x,terms);  // store value
  }
}
```

Feel free to use either version but for me the first version using `while` seems clearer; there is so much going on in the `for` statement in the second version that I find the code harder to follow. A good compiler will generate identical code from either version.

2.2 Kernel Call Syntax

The general form of a call to a CUDA kernel uses up to four special arguments in the `<<< >>>` brackets and the kernel itself can have a number of function arguments. The four arguments inside the `<<< >>>` brackets in order are:

First: defines the dimensions of the grid of thread blocks used by the kernel. Either an integer (or unsigned integer) for linear block addressing or a `dim3` type defining a 2D or 3D grid of thread blocks.

Second: defines the number of threads in a single thread block. Either an integer (or unsigned integer) for thread-linear addressing within a block or a `dim3` type to define a 2D or 3D array structure for the threads within a thread block.

Third: An optional argument of type `size_t` (or `int`) defining the number of bytes of dynamically allocated shared memory used by each thread block of the kernel. No shared memory is reserved if this argument is omitted or set to zero. Note that as an alternative the kernel itself can declare static shared memory. The size of a static shared memory allocation must be known at compile time but the size of dynamically allocated shared memory can be determined at run time.

Fourth: An optional argument of type `cudaStream_t` specifying the CUDA stream in which to run the kernel. This option is only needed in advanced applications running multiple simultaneous kernels. CUDA streams are discussed in Chapter 7.

2.3 3D Kernel Launches

Only the first 2 arguments specified as simple integers have been used in our examples so far. Our next Example 2.3 shows the use of `dim3` variables to run a kernel on a 3D grid. This example is really just for illustrative purposes and most of the code is concerned with printing grid related quantities.

Example 2.3 `grid3D` using a 3D grid of thread blocks

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <stdlib.h>

__device__  int   a[256][512][512];  // file scope
__device__  float b[256][512][512];  // file scope

__global__ void grid3D(int nx, int ny, int nz, int id)
{
  int x = blockIdx.x*blockDim.x+threadIdx.x; // find
  int y = blockIdx.y*blockDim.y+threadIdx.y; // (x,y,z)
  int z = blockIdx.z*blockDim.z+threadIdx.z;
  if(x >=nx || y >=ny || z >=nz) return;     // range check

  int array_size = nx*ny*nz;
  int block_size = blockDim.x*blockDim.y*blockDim.z;
  int grid_size  = gridDim.x*gridDim.y*gridDim.z;
  int total_threads = block_size*grid_size;
  int thread_rank_in_block = (threadIdx.z*blockDim.y+
          threadIdx.y)*blockDim.x+threadIdx.x;
  int block_rank_in_grid = (blockIdx.z*gridDim.y+
          blockIdx.y)*gridDim.x+blockIdx.x;
  int thread_rank_in_grid = block_rank_in_grid*block_size+
          thread_rank_in_block;
  // do some work here
  a[z][y][x] = thread_rank_in_grid;
  b[z][y][x] = sqrtf((float)a[z][y][x]);
  if(thread_rank_in_grid == id) {
    printf("array size   %3d x %3d x %3d = %d\n",
          nx,ny,nz, array_size);
    printf("thread block %3d x %3d x %3d = %d\n",
          blockDim.x, blockDim.y, blockDim.z, block_size);
    printf("thread  grid %3d x %3d x %3d = %d\n",
          gridDim.x,  gridDim.y, gridDim.z, grid_size);
    printf("total number of threads in grid %d\n",
          total_threads);
28      printf("a[%d][%d][%d] = %i and b[%d][%d][%d] = %f\n",
               z, y, x, a[z][y][x], z, y,  x, b[z][y][x]);
29      printf("for thread with 3D-rank %d 1D-rank %d
              block rank in grid %d\n", thread_rank_in_grid,
              thread_rank_in_block,block_rank_in_grid);
30    }
31  }

32  int main(int argc, char *argv[])
33  {
34    int id = (argc > 1) ? atoi(argv[1]) : 12345;
35    dim3 thread3d(32,  8,  2); // 32*8*2     = 512
36    dim3  block3d(16, 64,128); // 16*64*128 = 131072
37    grid3D<<<block3d,thread3d>>>(512,512,256,id);
38    return 0;
39  }
```

Description of Example 2.3

- Lines 1–4: Basic include statements for CUDA.
- Lines 5–6: Declare two large 3D arrays which have file scope and so can be used by any of the functions declared later in the same file. This is standard C/C++ but with an extra CUDA feature. By declaring the arrays with the `__device__` prefix we are telling the compiler to allocate these arrays in the GPU memory not in the host memory. Thus, the arrays `a` and `b` are usable by kernel functions but not host functions. Notice the array dimensions are in order z, y, x going from left to right, where memory is allocated so the adjacent x values are adjacent in memory. This is standard in C/C++ but opposite to Fortran which uses x, y, z order. Apart from array subscripts we will use "natural" x, y, z ordering in our code. This follows CUDA practice where for example a `float4` variable `a` has members `a.x, a.y, a.z, a.w` which are ordered from `x` to `w` in memory.

Using a global declaration is actually an easy way to create GPU arrays of known size, but we will rarely use it – there are two important disadvantages. Firstly, the array dimensions must be set at compile time not run time and secondly declaring variables with file scope is a deeply depreciated programming style because it leads to unstructured code where functions can easily cause unwanted side effects. In our subsequent examples we will allocate arrays in code and then pass them as pointer arguments to called functions as necessary.

- Line 7: The kernel grid3D is declared with four arguments which are the array dimensions and `id` which specifies the thread whose information will be printed.
- Lines 9–11: Here we calculate the thread's x, y and z coordinates within its thread block. The launch parameters defined in lines 35–36 set the block dimensions to 32, 8 and 2 and the grid dimensions to 16, 64 and 128 for x, y and z respectively. This means that in line 9 the built-in variables `blockDim.x` and `gridDim.x` are set to 32 and 16 respectively. Thus `threadIdx.x` and `blockIdx.x` will have ranges `[0,31]` and `[0,16]` and the desired coordinate `x` will have the range `[0,511]` which is required to index the global arrays `a` and `b`. Similarly, `y` and `z` have ranges of `[0,511]` and `[0,255]`. Within any particular thread block the `threadIdx` values will have ranges of `[0,31]`, `[0,7]` and `[0,1]` for `x`, `y` and `z`; note the `x` range corresponds to one

complete warp of threads; this is a design choice not chance. Having decided to use an x range of 32 we are restricted to smaller ranges for `y` and `z` as the product of all three is the thread block size which is limited by hardware to a maximum of 1024.

- Line 12: This is an out-of-range check on the calculated indices. This check is not strictly necessary here as we have carefully crafted the launch parameters to exactly fit the array dimensions. In general, this will not always be possible and it is good practice to always include range checks in kernel code.
- Lines 13–19: Calculate some values derived from the launch parameters. Most of these values would not be needed in a real-world problem, but we want to print them to illustrate the detail of 3D addressing in kernels.
  - Lines 13–15: The 3D array, thread block and grid sizes are simply calculated as the product of their dimensions.
  - Line 16: Similarly the total number of threads is the product of the thread block size `block_size` and the number of thread blocks `grid_size`.
  - Line 17: The rank of the thread within its 3D thread block is calculated using the standard 3D addressing rank formula:

    3D Rank Formula

    ```
    rank = (z*dim_y + y)*dim_x + x
    ```

    for a 3D array of dimensions `(dim_x, dim_y, dim_z)` laid out sequentially in memory with the `x` values adjacent, the `y` values are separated by stride of `dim_x` and the `z` values are separated by a stride of `dim_x*dim_y`. We will use versions of this formula very often in our examples, often encapsulated in a lambda function.
  - Line 18: Here we also use the rank formula to calculate the rank of the thread block within the grid of thread blocks.
  - Line 19: Here we use the 2D version of the rank formula to calculate the rank of the thread within the entire thread grid.
  - Lines 21–22: Here we actually do some real work storing values into the array `a` and `b` using indices derived from the threads position in the 3D thread grid.
  - Lines 23–29: Here, for one thread, chosen by the user, we print some of the calculated quantities.
- Lines 32–39: Here is the complete short main routine. Basically, we get a user settable value for `id` in line 34, set the kernel launch parameters in lines 35–36 and launch the kernel in line 37.

The results of running Example 2.3 are shown in the box below. There are 2 cases shown.

- Case `id=511`: This is the last thread in the first block which spans the range: `[0-31,0-7,0-1]` and the last point in this range is `(31,7,1)` which is shown correctly as the index `[1][7][31]` in the figure.
- Case `id=1234567`: To understand this we need to realise that a set of 16 blocks will span the complete x range for eight consecutive y and two consecutive z values. Hence the first 1024 blocks will span the range `[0-511,0-511,0-1]` which is two complete x-y slices of the array, The next 1024 blocks will span the slices with z in range `[2-3]` and so on. Since `1234567 = 512*2411+135` we have picked the 135th thread in the 2412th block. The first 4 x-y slices account for 2048 blocks so our pick is in the 364th block in the

4–5 slice pair. Next since `364 = 22*16 + 12` we conclude that our thread is in the 12th block in the set of 16 blocks that spans the index range `[0-511,168-175,5-6]`. This 12th block spans `[352-383,176-183,5-6]` and since the 135th thread is offset by `[7,4,0]` from this position we find an index set of `[359,180,5]` or a C/C++ 3D vector index address of `[4][180][359]`.

Case 1 Last thread in first thread block:

```
D:\ > grid3D.exe 511
array size   512 x 512 x 256 = 67108864
thread block  32 x   8 x   2 = 512
thread  grid  16 x  64 x 128 = 131072
total number of threads in grid 67108864
a[1][7][31] = 511 and b[1][7][31] = 22.605309
rank_in_block = 511 rank_in_grid = 511 rank of block_rank_in_grid = 0
```

Case 2 Thread 135 in block 2411

```
D:\ grid3d.exe 1234567
array size   512 x 512 x 256 = 67108864
thread block  32 x   8 x   2 = 512
thread  grid  16 x  64 x 128 = 131072
total number of threads in grid 67108864
a[4][180][359] = 1234567 and b[4][180][359] = 1111.110718
rank_in_block = 135 rank_in_grid = 1234567 rank of
  block_rank_in_grid = 2411
```

Results from running grid3D

As our second case illustrates 3D thread blocks are somewhat complicated to visualise but their unique selling point is that they group threads spanning 3D subregions of the array into a single SM unit where the threads can cooperate. In many volume processing applications, for example, automatic anatomical segmentation of 3D MRI scans, this is a key advantage. In practice, addressing such a subregion directly from the GPU main memory is often inefficient due to the large strides between successive y and z values. In such cases caching a 3D subregion in shared memory on the SM is an important optimisation.

However, if threads in your kernel only process individual elements of the array with little collaboration between threads then 1D thread-linear address is simpler to implement and offers more scope for tuning the launch configuration. Example 2.4 shows a version of the grid3D kernel with 1D thread-linear addressing.

Example 2.4 `grid3D_linear` thread-linear processing of 3D array

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
```

```
#include <stdlib.h>

__device__  int   a[256][512][512];  // file scope
__device__  float b[256][512][512];  // file scope

__global__ void grid3D_linear(int nx, int ny, int nz, int id)
{
  int tid = blockIdx.x*blockDim.x+threadIdx.x;

  int array_size = nx*ny*nz;
  int total_threads = gridDim.x*blockDim.x;
  int tid_start = tid;
  int pass = 0;

  while (tid < array_size){   // linear tid => (x, y, z)
    int x =  tid%nx;          // tid    modulo nx
    int y = (tid/nx)%ny;      // tid/nx modulo ny
    int z =  tid/(nx*ny);     // tid/(x-y slice size)
    // do some work here
    a[z][y][x] = tid;
    b[z][y][x] = sqrtf((float)a[z][y][x]);
    if(tid == id) {
      printf("array size   %3d x %3d x %3d = %d\n",
             nx,ny,nz,array_size);
      printf("thread block %3d\n",blockDim.x);
      printf("thread  grid %3d\n",gridDim.x);
      printf("total number of threads in grid %d\n",
             total_threads);
      printf("a[%d][%d][%d] = %i and b[%d][%d][%d] = %f\n",
             z,y,x,a[z][y][x],z,y,x,b[z][y][x]);
      printf("rank_in_block = %d rank_in_grid = %d
             pass %d tid offset %d\n", threadIdx.x,
             tid_start, pass, tid-tid_start);
    }
    tid += gridDim.x*blockDim.x;
    pass++;
  }         // end while
}

int main(int argc, char *argv[])
{
  int id      = (argc > 1) ? atoi(argv[1]) : 12345;
  int blocks  = (argc > 2) ? atoi(argv[2]) : 288;
  int threads = (argc > 3) ? atoi(argv[3]) : 256;
  grid3D_linear<<<blocks,threads>>>(512,512,256,id);
  return 0;
}
```

```
D:\ > grid3d_linear.exe 1234567 288 256
array size   512 x 512 x 256 = 67108864
thread block 256
thread  grid 288
total number of threads in grid 73728
a[4][363][135] = 1234567 and b[4][363][135] = 1111.110718
rank_in_block = 135 rank_in_grid = 54919 rank of
 block_rank_in_grid = 214 pass 16
```

Results from example 2.4 using the `grid3D_linear` kernel to process 3D arrays with thread-linear-addressing. The displayed array element has different 3D indices as compared to example 2.2 even though its linear index is the same as used in that example.

Description of Example 2.4

- Lines 1–8: These lines are unchanged from Example 2.3 except we have renamed the kernel.
- Line 9: The variable `tid` is set to the current thread's rank in the grid of threads blocks using the standard formula for 1D thread and grid-blocks.
- Lines 10–13: We set some variables used in the later print statements here. Note the formula for the `total_threads` is now the simple 1D case.
- Lines 14–31: These lines are the `while` loop used for thread-linear addressing.
- Lines 15–17: These lines show how to convert a thread-linear address into 3D coordinates with x the most rapidly varying coordinate and z the least rapidly varying. Note the division and modulus (%) operators are expensive and could be replaced by masking and shifting operations if `nx` and `ny` are known powers of 2. This gives better performance at the price of a less general kernel. For the case where both `nx` and `ny` are 512 we could use:

```
int x =  tid & 0x01ff;          // x = bits 0-8
int y = (tid >> 9) & 0x01ff;    // y = bits 9-17
int z =  tid >> 18;             // z = bits 18 and above
```

Calculation of 3D coordinates by extraction of bit fields from thread-linear address. The two 9-bit masks are for case where both nx and ny are equal to 512.

The formulae used to convert between a 3D `(x,y,z)` index triad and a linear index are shown in the box. Lines 15–17 here are an example of this:

Relations Between Linear and 3D indices

```
index = (z*ny+y)*nx+x

x = index % nx
y = (index / nx) % ny
z = index / (nx*ny)
```

- Lines 19–28: This is similar to the previous example except we are using the variable name `tid` instead of `thread_rank_in_grid`.
- Line 29: Here we increment `tid` using a stride equal to the length of the entire thread-grid
- Line 30: Here we increment a counter `pass` and continue to the next pass of the while loop. The variable `pass` is only used as part of the information printed. The actual linear address being used by a given `tid` within the `while` loop is `rank_in_grid+pass*total_threads`.
- Lines 33–40: The `main` routine now accepts two additional user arguments `blocks` and `threads`, which define the kernel launch parameters.

The results for the thread with a linear index of 1234567, the same value as used in Example 2.2, shows that this linear index corresponds to a 3D element [4][363][135] whereas in Example 2.2 using 3D grid and thread blocks it corresponded to the element [4][180][359]. Neither result is "wrong". The difference merely reflects the different order in which elements of the arrays are encountered.

Next we return to the CUDA topic of occupancy.

2.4 Latency Hiding and Occupancy

When a new kernel begins execution, a 32-thread warp will begin execution on each of the warp-engines on the GPU. These warps become *active-warps* and will remain *resident* until all warps in their thread block are complete. A likely early step will be loading an item of data from global memory but loading this data from global memory has a latency of several hundred clock-cycles. Thus, the active threads have to wait for their data to arrive before they can proceed – while waiting they are termed *stalled* threads. In fact, the hardware is quite sophisticated and the threads will not stall until an instruction that actually uses the pending data is reached. This means that a programmer can hide some of the latency by doing work independent of the data between requesting that data and using it. While useful, there may be limited scope for this in practice, and writing instructions in an unnatural order makes code harder to debug and maintain – probably it's best to rely on the compiler to do this job for you.

A key feature of the GPU design is that each warp engine can process several active warps in an interleaved fashion, if one of its active warps stalls, a warp-engine will switch to another active warp capable of running with no loss of cycles. Efficient switching between warps is possible because each thread in an active warp maintains its own state and set of registers. This is in dramatic contrast to the standard CPU architecture where a single set of registers is shared by all active threads making task switching an expensive operation. Thus, a warp-engine can continue executing instructions until all its active-warps have stalled waiting for memory access. Such a multiple stall may occur once at the start of a kernel but, provided each thread does "enough" computation between each global memory access, any further stalls are avoided. Note, enough computation is the number of instructions that could be executed in the time taken for a "typical" memory access. This can be several hundred instructions on a modern GPU. Latency hiding is illustrated in Figure 2.1.

Figure 2.1 shows four thread blocks, T1–T4. On the left they each run briefly and then stall waiting for a memory access. The access requests are shown as vertical arrows. The dispatch of the requested data is shown by the slanted dashed lines. Most of the latency

Table 2.2 *Kernel launch configurations for maximum occupancy*

Thread Block Size	Blocks per SM	Blocks per Grid if GPU has 20 SMs	Registers per Thread	Bytes of shared memory per Thread Block
32	64	1280	32	1 KB
64	32	640	32	2 KB
128	16	320	32	4 KB
256	8	160	32	8 KB
512	4	80	32	16 KB
1024	2	40	32	32 KB

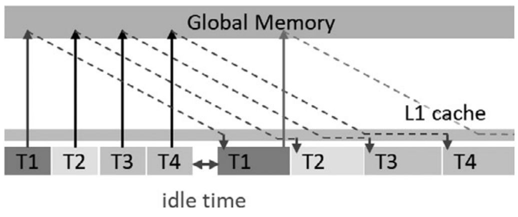

Figure 2.1 Latency hiding on GPUs

from T1's first memory request is hidden by the initial running of T2–T4. After a small amount of additional idle time T1's data arrives in L1 cache and T1 resumes execution using this data, T1 then stalls again with a second memory access shown as the grey vertical line. However, now there is no further idle time as data will now reach the L1 cache before it is needed.

On the Pascal architecture each SM has four warp-engines each capable of running up to 16 active warps, or equivalently 64 warps or 2048 threads on the SM. Thus, if the global memory latency is 400 cycles, each thread would need to do only 25 cycles worth of computation between memory accesses to fully hide this latency. This is a best-case situation because the kernel launch configuration may restrict the maximum number of active warps to less than 16. The *occupancy* of a kernel is defined as:

$$\text{occupancy} = \frac{\text{Actual number of active warps}}{\text{Maximum number of active warps}}.$$

The factors which may limit occupancy are the thread block size, the number of thread blocks, the number of registers used by each thread and the amount of shared memory used by a thread block. The number of thread blocks should be an integer multiple of the number of SMs in the GPU sufficient to give 2048 threads per block. There are also limits of 64K 32-bit registers and 64 KB of shared memory per SM. These limits are illustrated in Table 2.2 for a Pascal GPU with 20 SMs. Note, the number of registers allocated to each thread is determined by the compiler and depends on the complexity of the code. For full occupancy, the hardware has sufficient registers for 32 registers per thread. This value can be inspected and/or overridden by using NVCC compiler switches `--maxrregcount` and `--resource-usage`.

Full occupancy is more important for memory bound kernels than it is for compute bound kernels but it is always a good idea to keep your kernel code compact and straightforward as this will allow the compiler to allocate registers more effectively. It is also a good idea to split long calculations into stages and use separate kernels for each stage. Remember that the contents of GPU global memory is preserved between kernel launches.

It is now time to move on to more interesting GPU code where threads have to actively cooperate to perform a calculation.

2.5 Parallel Patterns

Parallel programming for GPUs running vast numbers of threads does require some rethinking of your approach to coding. Methods that work well in single CPUs or codes running a small number of independent threads may need to be rethought.

2.5.1 Avoid If Statements

One big difference is that branch statements are problematic in CUDA code, for example consider a CUDA kernel containing the following code:

```
if (flag == 0)  function1(a1,a2,...);
else            function2(b1,b2,...);
```

If all the 32 threads in a particular warp have `flag=0`, then all threads will call `function1` and there is very little performance loss, the same is true if none of the 32 threads have flag set to zero. However, if even just one of these threads has `flag` set to a non-zero value while the other 31 threads have `flag=0` then we get a so-called *branch-divergence*. The system handles this by serializing the calls to the two functions, that is, the subset of threads in the warp with `flag=0` execute the call to `function1` while the threads having `flag` non-zero stay idle. Then, when the function has returned for all active threads, the `else` clause calling `function2` is executed by the previously idle threads while previously active threads are now idle.[7]

CUDA Coding tip

- Avoid diverging if statements
- But only within 32-thread warps
- Modest conditional execution, e.g.:

  ```
  if(flag==0) x= x+1;
  ```

  is less harmful.

If the functions concerned are modest and require only a small fraction of the kernel's execution time, no great harm is done, but otherwise there can be up to a factor two drop in performance. If the called functions also have branch divergences the performance penalty is even worse.

If you have not encountered parallel programming on GPUs before, the need to remove all `if` statements from your code may seem like a deal breaker – but as we shall see in our examples this can be achieved quite straightforwardly in many cases. I also have to confess that I enjoy the intellectual challenge of designing good GPU code.

2.6 Parallel Reduce

The parallel reduce operation, mentioned in Chapter 1, is a good place to begin our discussion of parallel coding patterns. This is a good example of the more general problem of performing the same operation on a large set of numbers. Reduce itself involves finding the arithmetic sum of the numbers, but other operations such as max or min would require similar code.

As a specific case, consider the problem of summing N floating point numbers stored in the GPUs global memory. The first point to recognise is that each data item just requires a single add; thus we will be limited by memory access speed not arithmetic performance. This is the exact opposite to the situation in Example 1.3. We want to use as many threads as possible in order to hide memory latency efficiency so our basic algorithm is as shown in the box and illustrated in Figure 2.2.

Reduce Algorithm 1: Parallel sum of N numbers

- Use N/2 threads to get N/2 pairwise sums
- Set N = N/2 and iterate till N=1

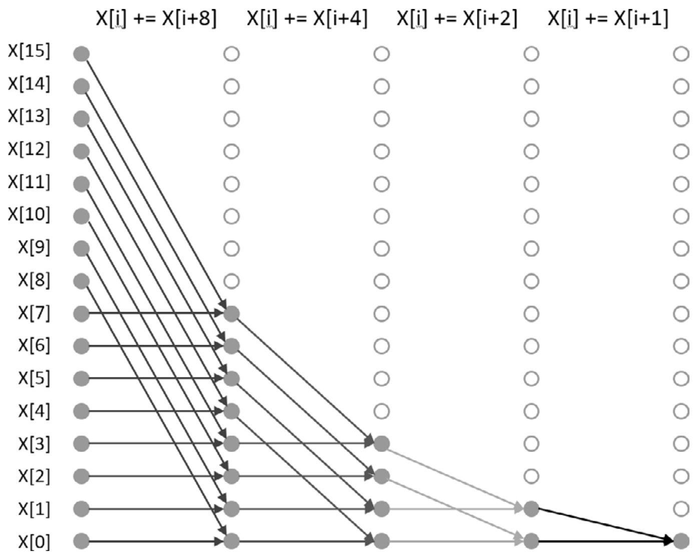

Figure 2.2 Pairwise reduction for the last 16 elements of x

The GPU implementation of this algorithm is shown in Example 2.5. The host code which initialises the data and manages most of the calculation is also shown and discussed in this example.

Description of Example 2.5

- Lines 1–3: All the necessary includes are here. The file `cx.h` is part of our example set and contains all the standard includes and some helpful definitions; it is fully described in Appendix G. The header `cxtimers.h` defines a portable C++ based timer object.
- Lines 4–8: Show the `reduce0` kernel which is very simple; each thread finds its rank, `tid`, in the grid and, making the tacit assumption that `tid` is in the range `0` to `m-1`, adds the appropriate element from the top half of the array to the bottom half. At this point, it is worth pausing to admire the simplicity of the kernel code. We have been able to directly express the idea for implementing a parallel reduction with a 2-line kernel. The method is illustrated in Figure 2.2 and shows the last few steps. Another thing to think about when looking at kernel code is the sheer power of the GPU; line 7 which does the additions will be executed in parallel by all the cores on the GPU, potentially delivering one or more operations for each core on each clock-cycle. My RTX 2070 GPU has 2304 cores running at about 1.1 GHz and can deliver several 10^{12} operations per second.
- Line 11: Here we set the array size N to a user supplied value or a default of 2^{24}.
- Lines 12–13: Here we allocate thrust host and device vectors `x` and `dev_x` to hold the data.
- Lines 15–17: These lines initialise a C++ random number generator and use it to fill `x`. The use of generators from `<random>` is much preferred over the deprecated `rand()` function from ancient C.

Example 2.5 `reduce0` kernel and associated host code

```
01  #include "cx.h"
02  #include "cxtimers.h"
03  #include <random>

04  __global__ void reduce0(float *x, int m)
05  {
06    int tid = blockDim.x*blockIdx.x+threadIdx.x;
07    x[tid] += x[tid+m];
08  }

09  int main(int argc, char *argv[])
10  {
11    int N = (argc >1) ? atoi(argv[1]) : 1 << 24; // 2^24

12    thrust::host_vector<float>       x(N);
13    thrust::device_vector<float> dev_x(N);

14    // initialise x with random numbers and copy to dx
15    std::default_random_engine gen(12345678);
16    std::uniform_real_distribution<float> fran(0.0,1.0);
17    for(int k = 0; k<N; k++) x[k] = fran(gen);
18     dx = x;   // H2D copy (N words)

19     cx::timer tim;
20     double host_sum = 0.0;    // host reduce!
21     for(int k = 0; k<N; k++) host_sum += x[k];
22     double t1 = tim.lap_ms();

23     // simple GPU reduce for N = power of 2
24     tim.reset();
25     for(int m = N/2; m>0; m /= 2) {
26       int threads = std::min(256,m);
27       int blocks =  std::max(m/256,1);
28       reduce0<<<blocks, threads>>>(dev_x.data().get(), m);
29     }
30     cudaDeviceSynchronize();
31     double t2 = tim.lap_ms();

32     double gpu_sum = dev_x[0];   // D2H copy (1 word)
33     printf("sum of %d random numbers: host %.1f %.3f ms,
             GPU %.1f %.3f \n", N, host_sum, t1, gpu_sum, t2);
34     return 0;
35   }

D:\ > reduce0.exe
sum of 16777216 random numbers: host 8388314.9 14.012 ms
                                 GPU 8388315.0  0.535 ms
```

- Line 18: The contents of `x` are copied from the host to `dev_x` on the GPU. The details of the transfer are handled by thrust.
- Lines 19–22: A timed loop to perform the reduction on the host using a simple `for` loop.
- Lines 24–31: Implement the GPU-based parallel iteration of Algorithm 1. For each pass through the `for` loop the `reduce0` kernel called in line 28 causes the top half of the array `dev_x` to be "folded" down to an array of size `m` by adding the top `m` elements to the bottom `m` elements. The last pass through the loop has `m=1` and leaves the final sum in `dev_x[0]`; this value is copied back to the host in line 35. Lines 28–29: Within the `for` loop the kernel launch parameters `blocks` and `threads` are set so that the total number of threads in the grid is exactly `m`. This code will fail if N is not a power of 2 due to rounding down errors at one or more steps in the process.

In CUDA programs a kernel launch such as that used in line 28 will not block the host which will proceed to the next line of the host program without waiting for the kernel call to finish. In this case that means all the kernel calls (23 in all for $N=2^{24}$) will be rapidly queued to run successively on the GPU. In principle the host can do other CPU work while these kernels are running on the GPU. In this case we just want to measure the duration of the reduction operation so before making the time measurement we must use a `cudaDeviceSynchronize` call in line 30 which causes the host to wait for all pending GPU operations to complete before continuing. This kind of synchronisation issue often occurs in parallel code.

- Lines 32–33: Here we copy the final sum in the `dev_x[0]` back to the host, again using thrust, and print results.

The bottom line shows the results obtained running this program with the default value of 2^{24} for the number of values to be summed. Note the kernel execution time of 0.535 ms is too short a single measurement to be reliable. The values shown in these reduce examples were in fact obtained as averages of 10,000 runs using a `for` loop around kernel calls. An alternative method would be to use the Nsight Compute profiling tool, but our simple host-based method using `cx::timer` is a good starting point.

An interesting feature of the results obtained is that the host calculation uses a 64-bit `double` variable to accumulate the sum of the x values but the GPU does not. However, the results differ by only 1.1 parts in 10^{-8} – this is about the best that can be expected from a 32-bit floating point calculation; rounding errors have not accumulated in the GPU calculation. On the other hand, if we change the variable `host_sum` (line 23 of the host calculation in Example 2.4) to a `float` instead of a `double` the accuracy of the host calculation falls to only about 3 parts in 10^{-5} thus rounding errors do accumulate in the host calculation. This difference is due to the fact that the GPU accumulates many intermediate partial sums and thus tends to be always adding numbers of similar sizes. Although this improvement is data dependent, this is encouraging to see, as we plan to use 32-bit floats in our GPU calculations whenever possible.

While accurate, our kernel is very inefficient and unlike the compute bound problem in Chapter 1, reduction is a memory bound problem and the `reduce0` kernel does not handle this well.

Firstly, the only calculation done by each thread is a single addition, and secondly the statement:

```
x[tid] += x[tid+m],
```

triggers three global memory operations, namely loading both the values stored in `x[tid]` and `x[tid+m]` into GPU registers and then storing the sum of these values back into `x[tid]`. If we could accumulate partial sums in local registers, that would reduce the number of global memory accesses needed for each addition down to one, which offers a speed-up by a potential factor of three.

Secondly, the host calls the kernel iteratively, halving the array size at each step to complete the reduction process, leaving the final sum in the first array element. The effect of this is to double the number of times the `x[tid] += x[tid+m]` statement is performed. If we could instead perform the iteration inside the kernel that could also reduce the number of memory accesses required.

Finally, the kernel of Example 2.5 is not general enough, the array size must be a power of 2 and the host has to make multiple calls to the kernel using a carefully crafted sequence of launch parameters. A better solution is to use thread-linear addressing, with user defined values of blocks and threads to get something like Example 2.6:

Example 2.6 `reduce1` kernel using thread-linear addressing

```
04   __global__ void reduce1(float *x, int N)
05   {
06     int tid = blockDim.x*blockIdx.x+threadIdx.x;
06.1   float tsum = 0.0f;
06.2   for(int k=tid; k<N; k += gridDim.x*blockDim.x)
                                          tsum += x[k];

07     x[tid] = tsum; // store partial sums in first
08   }                // gridDim.x*blockDim.x elements of x
       . . .
24     tim.reset();
25     reduce1<<< blocks, threads >>>(dx.data().get(),N);
26     reduce1<<< 1, threads >>>(dx.data().get(),blocks*threads);
27     reduce1<<< 1,1 >>>(dx.data().get(),threads);
30     cudaDeviceSynchronize();
31     double t2 = tim.lap_ms();
32     double gpu_sum = dx[0];  //D2H copy (1 word)

D:\ > reduce1.exe
sum of 16777216 numbers: host 8388889.0 14.012 ms
                          GPU 8388315.5  0.267 ms
```

Description of Example 2.6

- Lines 5–10: This is the reduce1 kernel, now 4 lines long. We use thread-linear addressing to sum all the N values contained in `x` into lower `block*threads` elements. Each thread accumulates its own partial sum in its copy of the register variable `tsum` and then stores the final result in `x[tid]` where `tid` is the thread's unique rank in the grid. In this example we have used a `for` loop instead of a `while` clause to keep the code compact.

  Note line 7, where we change the value of an element of `x`, requires thought. Not all threads actually run at the same time so using the same array for a kernel's input and output is always potentially dangerous. Can we be sure no thread other than `tid` needs the original value in `x[tid]`? If the answer is no, then the kernel would have a *race condition* and the results would be undefined. In the present case we can be sure because every thread uses a separate disjoint subset of the elements of `x`. If in doubt you should use different arrays for kernel input and output.
- Lines 28–30: Replace the for loop in lines 28–32 of the original. Now we make three calls to `reduce1`, the first uses the full thread-grid defined by the user set variables blocks and threads. After this call the lower `blocks*threads` elements of `x` contain partial sums. The second kernel call uses just 1 thread block of size threads, after this call the partial sums are in the first `threads` elements of x. The third call uses just 1 thread to sum `threads` elements of `dx` and leave to total sum in the first element. Clearly the last two kernel calls do not make efficient use of the GPU.

The bottom line shows the time required for the `reduce1` kernel; the host time is unchanged but the GPU time is about half that required for `reduce0`.

Table 2.3 *Features of GPU generations from Kepler to Ampere*

GPU Generation	Kepler		Maxwell			Pascal			Volta	Turing	Ampere
Compute Capability	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0	7.5	8.0
Shared Mem per SM (KB)	48	112	64	96	64	64	96	64	96	64	164
Max Resident Threads per SM					2048					1024	2048
Shared Mem per Thread (bytes)	24	56	32	48	32	32	48	32	48	64	82
4-byte Registers per Thread	32	64	32	32	32	32	32	32	32	64	32

The `reduce1` is about twice as fast as `reduce0` which is not a bad start but we can do more. Our `reduce1` kernel is also much more user friendly, it can cope with any value of the input array size `N` and the user is free to tune the launch configuration parameters `blocks` and `threads`.

Notice that in the last reduce step, line 27 of Example 2.6, we used a single thread running alone to sum `threads` values stored in x. We can do better than this by getting the threads in each thread block to cooperate with each other.

A key feature of NVIDIA GPUs is shared memory which allows the threads within a thread block to cooperate efficiently. A thread block running on the GPU can reserve an allocation of shared memory, and all threads in the thread block can then read from and write to that memory. Threads in different thread blocks of a kernel get different allocations of shared-memory and cannot read or write to each other's allocations. Each SM unit has a pool of shared memory which is divided between all the currently resident thread blocks that request shared memory. Of course, device global memory is also visible to all threads on all thread blocks but accessing shared memory is much faster and can be as fast as using registers.

NVIDIA GPUs have at least 48 KB of shared memory per SM unit and more recently 64K. The precise amount depends on the CC level and is summarised in Table 2.3. It is clear from the table that at the thread level shared memory is a scarce resource; there is enough for only 8 or 16 4-byte words per thread. If more is required one can try using more shared memory per thread at the expense of lower occupancy. The runtime system will automatically run fewer kernels per SM if necessary. If a kernel requests more shared memory than the total available on an SM then the kernel launch will fail. Shared memory featured very prominently in early CUDA tutorials and books because these GPUs had little or no caching capability for global memory accesses. More recent GPUs have good L1 and L2 caching and in particular L1 caching can sometimes work well as an alternative to using shared memory. Interestingly devices with CC 7.0 and above have a single fast memory resource that can be shared between L1 and shared memory partitions in different proportions for different kernels.

In our next Example 2.7 we use shared memory to enable the threads in each thread block to sum their individual accumulated totals and then write a single word with the block-sum to external memory. The scheme of Figure 2.2 is again used for this intra-block reduction.

Example 2.7 `reduce2` kernel showing use of shared memory

```
__global__ void reduce2(float *y, float *x, int N)
{
  extern __shared__ float tsum[]; // Dynamic Shared Mem

  int id = threadIdx.x;
  int tid = blockDim.x*blockIdx.x+threadIdx.x;
  int stride = gridDim.x*blockDim.x;
  tsum[id] = 0.0f;
  for(int k=tid;k<N;k+=stride) tsum[id] += x[k];
  __syncthreads();
  // power of 2 reduction loop
  for(int k=blockDim.x/2; k>0; k /= 2){
    if(id<k) tsum[id] += tsum[id+k];
    __syncthreads();
  }
  // store one value per thread block
  if(id==0) y[blockIdx.x] = tsum[0];
}

int main(int argc, char *argv[])
{
  int N = (argc > 1) ? atoi(argv[1]) : 1 << 24;
  int blocks  = (argc > 2) ? atoi(argv[2]) : 256;
  int threads = (argc > 3) ? atoi(argv[3]) : 256;
  thrust::host_vector<float>    x(N);
  thrust::device_vector<float>  dx(N);
  thrust::device_vector<float>  dy(blocks);

  // initialise x with random numbers
  std::default_random_engine gen(12345678);
  std::uniform_real_distribution<float> fran(0.0,1.0);
  for(int k = 0; k<N; k++) x[k] = fran(gen);
  dx = x;  // H2D copy (N words)

  cx::timer tim;
  double host_sum = 0.0;    // host reduce!
  for(int k = 0; k<N; k++) host_sum += x[k];
  double t1 = tim.lap_ms();

  // simple GPU reduce for any value of N
  tim.reset();
  reduce2<<<blocks,threads,threads*sizeof(float)>>>
               (dy.data().get(),dx.data().get(),N);
  reduce2<<<1, blocks, blocks*sizeof(float)>>>
               (dx.data().get(),dy.data().get(),blocks);
    cudaDeviceSynchronize();
    double t2 = tim.lap_ms();
    double gpu_sum = dx[0];  // D2H copy (1 word)
    printf("sum of %d numbers: host %.1f %.3f ms
         GPU %.1f %.3f ms\n",N,host_sum,t1,gpu_sum,t2);
    return 0;
  }

D:\ > reduce2.exe 16777216 256 256
sum of 16777216 numbers: host 8388314.9 14.012 ms
                         GPU 8388314.5  0.202 ms
```

Description of Example 2.7

- Lines 1–9: This is the reduce2 kernel which uses shared memory.
- Line 1: This kernel uses `y` as an output array and `x` as the input array with N elements. The previous `reduce1` kernel used `x` for both input and output.
- Line 3: Here we declare the float array `tsum` to be a shared memory array of size determined by the host at kernel launch time. Shared memory is on-chip and very fast. Each SM has its own block of shared memory which has to be shared by all the active thread blocks on that SM. All threads in any given thread block share `tsum` and can read or write to any of its elements. Inter-block communication is not possible using `tsum` because each thread block has a separate allocation for its `tsum`. For this kernel, an array size of `blockDim.x` is assumed for `y` and it is up to the host code to ensure that the correct amount has been reserved. Incorrectly specified kernel launches could cause hard-to-find bugs.
- Lines 4–6: To prepare for thread-linear addressing we set `id` to the rank of the current thread in its thread block, `tid` to the rank of the current thread in the whole grid and `stride` to the number of threads in the whole grid.
- Line 7: Each thread "owns" one element of `tsum`, `tsum[id]` for this part of the calculation. Here we set the element to zero.
- Line 8: Here each thread sums the subset of elements of x corresponding to `x[id+n*stride]` for all valid integers $n \geq 0$. Although there is a large stride between successive elements, this is a *parallel* calculation and adjacent threads will simultaneously be reading adjacent elements of `x` so this arrangement is maximally efficient for reading GPU main memory. Note that for large arrays, most of the kernel's execution time is used on this statement and very little calculation is done per memory access.
- Line 9: The next step of the algorithm requires threads to read elements of `tsum` that have been updated by *different* threads in the thread block. Technically that's fine – this is what shared memory is for. However, not all threads in the thread block run at the same time and we must be sure that *all* threads in the thread block have completed line 8 before *any* of the threads proceed. The CUDA function `__syncthreads()` does exactly this; it acts as a barrier, all (non-exited) threads in the thread block must reach line 9 before any of them can proceed. Note that `__syncthreads` only synchronises threads in a single thread block. This is in contrast to the host function `cudaDeviceSynchronize()` which ensures that all pending CUDA kernels and memory transfers have completed before allowing the host to continue. If you want to ensure that all threads in all thread blocks have reached a particular point in a kernel then in most cases your only option is to split the kernel into two separate kernels and use `cudaDeviceSynchronize()` between their launches.[8]

- Lines 10–13: This is the implementation of the power of 2 reduction scheme of Figure 2.2 implemented to sum the values in `tsum` on a thread block. This section of code assumes that `blockDim.x` is a power of 2. Note that the number of active threads reduces by a factor of 2 on each pass through the for `loop`. Older tutorials tend to dwell on further optimisation of this loop by explicitly unrolling and exploiting synchronicity within 32-thread warps. This will be discussed in the next chapter on cooperative groups. For now, note further optimisation of this loop is only important for smaller datasets.
- Line 14: The final block sum accumulated in `tsum[0]` is stored in the output array `y` using `blockIdx.x` as an index.
- Lines 16–45: This is the main routine; much of it is similar to the previous example and here we will just mention differences.
- Lines 18–20: Here we give the user the option to set the array size `N` and the launch parameters `blocks` and `threads`. Note `blocks` needs to be a power of 2 for the `reduce2` kernel to work properly.
- Line 23: We now allocate a device array `dy` having dimension `blocks`. This new array will hold the individual block wide reduction sums.
- Line 35: Here we call the `reduce2` kernel for the first time to process the whole `dx` array with the block sums being stored in the output array `dy`. Note the third kernel argument requesting a shared memory allocation of `threads` 4-byte floats for each active thread block. A large value here may result in reduced occupancy.
- Line 36: Here we call `reduce2` again but with the array arguments swapped round. This has the result of causing the values stored in `y` by the previous kernel call, to themselves be summed with the total placed in `x[0]`. This requires a launch configuration of a single thread block of size `blocks` threads.

The result at the end of the listing shows that `reduce2` is about 2.65 times faster than `reduce0`.

A worthwhile optimisation of the `reduce2` kernel would be to drop the restriction that blocks must be a power of 2. This is because in many GPUs the number of SM units is not a power of 2. For example, my GPU has 36 SMs so to keep all SMs equally busy it is better to use 288 rather than 256 for the number of user set value of `blocks`. We can do this by replacing `blockDim.x` in line 10 of the `reduce2` kernel by the smallest power of 2 greater than or equal to `blocks`. For `blocks` = 288 this would be 512. The effect of doing this is that in the first pass when k=256, threads with rank 0 to 31 will add values from `tsum[256]` to `tsum[287]` to their `tsum` values. We also have to add an out-of-range check to prevent threads `32-255` from attempting out-of-range additions. The modified reduce3 kernel is shown in Example 2.8.

Example 2.8 `reduce3` kernel permitting non-power of two thread blocks

```
01  __global__ void reduce3(float *y,float *x,int N)
02  {
03    extern __shared__ float tsum[];

04    int id = threadIdx.x;
05    int tid = blockDim.x*blockIdx.x+threadIdx.x;
06    int stride = gridDim.x*blockDim.x;
07    tsum[id] = 0.0f;
08    for(int k=tid;k<N;k+=stride) tsum[id] += x[k];
09    __syncthreads();
```

```
     // next higher power of 2
10.1 int block2 = cx::pow2ceil(blockDim.x);
     // power of 2 reduction loop
10.2 for(int k=block2/2; k>0; k >>= 1){
11     if(id<k && id+k < blockDim.x) tsum[id] += tsum[id+k];
12     __syncthreads();
13   }
     // store one value per block
14   if(id==0) y[blockIdx.x] = tsum[0];
15 }

D:\ >reduce3.exe 16777216 288 256
sum of 16777216 numbers: host 8388314.9 14.012 ms
                          GPU 8388314.5  0.196 ms
```

Description of Example 2.8

In this example the kernel and main routine (not shown) are the same as in Example 2.7 except for lines 10 and 11 of the kernel.

- Line 10.1: Here we add a new variable `block2` which is set the value of `blockDim.x` rounded up to the lowest power of 2 greater than or equal to `blockDim.x`. We use the `cx` utility function `pow2ceil` for this. That function is implemented using the NVIDIA intrinsic function `__clz(int n)` which returns the number of the most significant non-zero bit in `n`. This is a device-only function.
- Line 10.2: This is the same as line 10 in `reduce2` except we use the rounded up `block2/2` as the starting value of `k`.
- Line 11: This corresponds to line 11 of `reduce2` with an added out-of-range check on `id+k`.

In the last line we see that launching this kernel with exactly 8 thread blocks per SM gives a speed-up of 2.73 compared to `reduce0`, slightly better than `reduce2`.

The `reduce3` kernel is about 70 times faster than the single core host version. While this is not quite as spectacular as our Chapter 1 result for a CPU bound calculation, reduction is a memory bandwidth bound calculation with just one add per read of 4-bytes of memory so we expect reduced performance. Given that the GPU memory bandwidth is only about 10 times that of the CPU the factor 70 improvement shows that other GPU features including the latency hiding are helping speed up this memory bound problem. The last trick to try is explicitly unrolling the loop in lines 10–13.

Example 2.9 `reduce4` kernel with explicit loop unrolling

```
01  __global__ void reduce4(float * y, float * x,int N)
02  {
03    extern __shared__ float tsum[];

04    int id = threadIdx.x;
05     int tid = blockDim.x*blockIdx.x+threadIdx.x;
06     int stride = gridDim.x*blockDim.x;
07     tsum[id] = 0.0f;
08     for(int k=tid;k<N;k+=stride) tsum[id] += x[k];
09     __syncthreads();

10     if(id<256 && id+256 < blockDim.x)
                  tsum[id] += tsum[id+256]; __syncthreads();
11     if(id<128) tsum[id] += tsum[id+128]; __syncthreads();
12     if(id< 64) tsum[id] += tsum[id+ 64]; __syncthreads();
13     if(id< 32) tsum[id] += tsum[id+ 32]; __syncthreads();

14     // only warp 0 array elements used from here
15     if(id< 16) tsum[id] += tsum[id+16]; __syncwarp();
16     if(id< 8)  tsum[id] += tsum[id+ 8]; __syncwarp();
17     if(id< 4)  tsum[id] += tsum[id+ 4]; __syncwarp();
18     if(id< 2)  tsum[id] += tsum[id+ 2]; __syncwarp();
19     if(id==0)  y[blockIdx.x] = tsum[0]+tsum[1];
20   }

D:\ >reduce4.exe 16777216 288 256
sum of 16777216 numbers: host 8388314.9 14.012 ms
                         GPU 8388314.5   0.195 ms
```

Description of Example 2.9

- Line 1–9: These are the same as in the previous example.
- Lines 10–19: These replace the for loop and last line of the previous example. Here we have unrolled the loop on the explicit assumption that the number of threads per block, `blockkDim.x`, is in the range `[256,511]`. In practice we used 256 threads and 288 blocks for the first call to `reduce4` and 288 threads and 1 block for the second call to `reduce4`. This kernel could easily be generalised to work with a larger range of thread block sizes, for example, by making the thread block size a template parameter. You can find such generalisations in many tutorials; for example, the very early blog by Mark Harris: "Optimizing Parallel Reduction in CUDA, November 2 2007" downloadable from https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.
- Line 10: This is the first step in the parallel reduction chain; values in `tsum[256-511]` (if any) are added to those in `tsum[0-255]`. This line is needed for second kernel call with `blockDim.x=288`. The if statement is then necessary to avoid out of range errors for threads `32-255`.
- Lines 11–13: These lines are the next three steps in the parallel reduction. No out-of-range checks are needed here on the assumption `blockDim.x` is at least 256.

Note there is a `__syncthreads` after each step in lines 10–13. These calls are necessary to ensure that all threads in the thread block have completed their addition before any of them proceed to the next step.

- Lines 15–19: These lines are the final five steps in the parallel reduction tree. In these lines only the first 32 threads participate. These threads are all in the same warp so we can replace __syncthreads with the much faster __syncwarp. For devices of CC < 7 all threads in the same warp act in strict lockstep so here it is possible to rely on implicit warp synchronisation and omit the __syncwarp calls entirely. You will find this done in early (now deprecated) tutorials. Even if you only have access to older devices, we strongly recommend that you always use syncwarp where it would be necessary on newer devices to maintain code portability.

The result shown in the last line shows at best a tiny improvement compared to reduce3.

The performance difference between reduce3 and reduce4 is small but reduce4 has introduced us to warp level programming. We will return to the reduce problem in the next chapter and show how warp-based programming can be taken much further.

Next we will discuss shared memory in more detail and then explore another application, namely matrix multiplication.

2.7 Shared Memory

Shared memory is a fast access memory pool of size typically 64 KB available on each SM. Kernels can elect to use shared memory by declaring one or more array or scalar variables as prefixed with the decoration __shared__.

Each thread block running on an SM gets a separate allocation of the required size from the shared memory pool. If a kernel has thread blocks requiring more than 32 KB (i.e. more than half the total available) then only one thread block can run on the SM at once which severely reduces occupancy. Thus, shared memory use is one of the factors to be considered when optimising occupancy.

As the name implies, shared memory is shared by all the threads in a thread block, i.e. any part of it can be read or written by any thread in the thread block. Note that, while this is extremely useful in many situations, the sharing is local – shared memory is not shared between thread blocks. The contents of the shared memory belonging to a given thread block are lost when that thread block exits.

Shared memory allocation can be either *static* or *dynamic*. For static shared memory allocation the required sizes are declared in the kernel code using values known a compile time, for example using:

```
__shared__ float data[256];
```

This is arguably the simplest method but lacks flexibility. Shared memory array sizes typically depend on the number of threads in the thread block, thus if they are fixed at compile time then so is the size of the thread block.

Dynamic shared memory allocation is an alternative where the kernel does not specify the size of an array but declares an externally allocated shared pointer, for example:

`extern __shared__ float *data;` or equivalently `extern __shared__ float data[];`

In this case the actually required memory size is specified by the host at kernel launch time using the value (in bytes) as third kernel launch third parameter. Since this value can be a variable determined during program execution this method of memory allocation known as dynamic. Examples 2.4 and 2.5 use this method.

Static memory declarations are usually placed at the start of your kernel code, but this is not mandatory; obviously like all variables, their declaration needs to precede their use.[9] More than one shared array or variable can be declared in a single kernel, the static allocation case is straightforward, for example:

```
__shared__ float sx[256];
__shared__ unsigned short su[256];
```

will work as expected creating separate arrays with a total memory requirement of `1024+512` bytes. However, the corresponding dynamic allocation in kernel code:

```
extern __shared__ float sx[];
extern __shared__ unsigned short su[];
```

will compile successfully without warnings but both arrays will start at the *same* address – namely the starting address of the reserved block of shared memory that is allocated when the kernel is runs. Although annoying and bug prone, this is the only possible thing the compiler can do since it does not know the array sizes at compile time. Thus, during execution, writing to either array will write to both, leading to kernel failure or (worse) hard to find bugs. In order to fix this problem, we need to modify the kernel code so that only one array is declared `extern` and all other arrays are declared as pointers with appropriately calculated offsets from the `extern` array as shown in Example 2.10.

Example 2.10 `shared_example` kernel showing multiple array allocations

```
__global__ void shared_example(float *x,float *y,int m)
{
  // notice order of declarations,
  // longest variable type first
  // shortest variable type last
  // NB sx is a pointer to the start of the shared
  extern __shared__ float sx[]; // memory pool

  ushort* su = (ushort *)(&sx[blockDim.x]); // start after sx
  char*   sc =   (char *)(&su[blockDim.x]); // start after su

  int id = threadIdx.x;

  sx[id] = 3.1459*x[id];
  su[id] = id*id;
  sc[id] = id%128;
  // do useful work here
  . . .
  int threads = (argc >1) ? atoi(argv[1]) : 256;
  int blocks =  (size+threads-1)/threads;
  int shared = threads*(sizeof(float) +
                     sizeof(ushort) + sizeof(char));
  shared_example<<< blocks, threads, shared >>>
                          (dx_ptr,dy_ptr,size);
```

Description of Example 2.10

The start of the kernel using three dynamic shared memory arrays is shown in lines 1–12 of this example. Here we will assume that the required size of each array is the number of threads in the thread block, i.e. `threadDim.x` for 1D thread grids.

- In line 6: A single dynamically allocated shared memory array `sx` of type `float` is declared. Note that sx is just a C style pointer to an array of floats. We could have used "`float *sx;`" instead of "`float sx[]`"
- Lines 7–8: Here pointers to two additional arrays, `su` and `sc`, are declared using pointer arithmetic to calculate their offsets from the start of sx. In line 7 the `su` pointer is set to the address after `blockDim.x` floating point elements of the array `sx` and then cast to the `ushort` pointer type. Similarly, in line 8 the `sc` pointer is set to the address after `blockDim.x` `ushort` elements of the array `su` and then cast to the `char` type.
- Lines 9–12: Here we demonstrate use of the arrays, the variable `id` is set to the current threads's rank in the thread block and then used normally to index the three arrays.
- Lines 30–33: These show a fragment of the corresponding host code containing the kernel.
  - Line 30: The launch parameter `threads` is set using an optional user supplied value.
  - Line 31: The parameter `blocks` is then set as usual. in lines 30–31.
  - Line 32: A third launch parameter `shared` is set in line 32. The value stored in `shared` is calculated as the total number of *bytes* necessary for the three arrays.
  - Line 33: This shows the kernel launch using three parameters in the launch configuration.

One subtle detail of this example is that the calculation made in line 32 makes no allowance for memory "gaps" between the arrays that might be needed for natural alignment of each array on memory boundaries. However, because the declarations and assignments in lines 5–8 of the kernel go from the longest variable type (4-byte `floats`) to the shortest variable type (1-byte `chars`), natural alignment will be achieved for all three arrays without the compiler needing to introduce gaps.[10]

Simple variables can also appear in dynamically allocated shared memory, but since their size, namely `sizeof(variable type),` is known at compile time, static allocation is the best choice. If the variable is intended to contain some parameter which is read but not changed by the threads, then using `constant` memory might be a better choice. Note that `constant` memory will be automatically used for most kernel arguments.

2.8 Matrix Multiplication

Our next example is a naturally 2D problem – matrix multiplication. Matrix multiplication has in fact been featured as an early example of the use of shared memory in all releases of the CUDA SDK and our final version is closely based on the that SDK code. A matrix **M** is simply a 2D rectangular array of numbers, if the matrix **M** has n rows and m columns we say it is an n by m (or n $\times$ m) matrix and M_{ij} denotes the element in row i and column j. Note the order of the suffices is significant, in general M_{ij} and M_{ji} are different values in different positions in the matrix. In the special case where $M_{ij} = M_{ji}$ for all values of i and j the matrix is *symmetric* and *square* meaning that its dimensions n and m are the same. Matrices are multiplied using the formula:

Formula for Matrix Multiplication

If **A** is n × m and **B** is m × p then **C** = **A**×**B** is a n × p matrix with elements given by:

$$C_{ij} = \sum_{k=1}^{m} A_{jk} B_{kj}.$$

Note the number of columns of **A** must be equal to the number of rows of **B**

As mentioned above 2D arrays can be implemented in numerous ways, but the most efficient method is to use a single contiguous memory block and address elements using the 2D version of the rank formula given above, namely:

2D linear addressing for matrices

The expression:

`A[i*Ncols+j]`

gives the element A_{ij} of the matrix held in the 1D array **A**, `Ncols` is the number of columns in **A**. The number of rows `Nrows` is not explicit here but the size of **A** must be at least `Nrows*Ncols`. This notation is correct if **A** is either a thrust or std vector or a simple pointer.

In this scheme the column index `j` is the "hot" index, `j` and `j+1` refer to adjacent memory locations whereas i and `i+1` refer to memory locations separated by a stride `Ncols`. In most of our examples we will use thrust as a container class for both host and device arrays.

We are fully aware that C++ provides nice tools to create beautiful containers for vectors and matrices with support for more elegant addressing schemes such as the Fortran like `A(i,j)` or the C multidimensional `A[i][j]` style. Indeed we had intended to adopt one of these wrappers when we began this project. However at the time of writing any attempt to pass any object other than a bare pointer to a CUDA kernel prevents any optimisations based on using `__restrict` and since our goal is fast code we will stick with the simple explicit address calculation as shown in the box.[11] Example 2.11 shows a straightforward implementation of matrix multiplication on the host.

Example 2.11 `hostmult0` matrix multiplication on host CPU

```
01   #include "thrust/host_vector.h"
02   #include "cxtimers.h"
03   #include <random>

04   int hostmult0(float *C, float A, float * B, int Ay,
                                              int Ax, int Bx)
05   {
06     // compute C=A*B for matrices (assumes Ax = By)
07     for(int i=0;i<Ay;i++) for(int j=0;j<Bx;j++){
08       C[i*Bx+j] = 0.0;       // row.col dot product
09        for(int k=0;k<Ax;k++)C[i*Bx+j] += A[i*Ax+k]*B[k*Bx+j];
10      }
11      return 0;
12    }

13    int main(int argc, char *argv[])
14    {
15      int Arow = (argc > 1) ? atoi(argv[1]) : 1024;
16      int Acol = (argc > 2) ? atoi(argv[2]) : Arow;
17      int Brow = Acol;
18      int Bcol = (argc > 3) ? atoi(argv[3]) : Brow;
19      int Crow = Arow;
20      int Ccol = Bcol;

21      thrust::host_vector<float> A(Arow*Acol);
22      thrust::host_vector<float> B(Brow*Bcol);
23      thrust::host_vector<float> C(Crow*Ccol);

24      // initialise A and B with random numbers
25      std::default_random_engine gen(12345678);
26      std::uniform_real_distribution<float> fran(0.0,1.0);
27      for(int k = 0; k<Arow*Acol; k++) A[k] = fran(gen);
28      for(int k = 0; k<Brow*Bcol; k++) B[k] = fran(gen);
29      cx::timer tim;

30      hostmult0(C.data(),A.data(),B.data(),Arow,Acol,Bcol);
31      double t1 = tim.lap_ms();
32      double flops = 2.0*(double)Arow*(double)Acol*
                                        (double)Bcol;
33      double gflops= flops/(t1*1000000.0);
34      double gbytes = gflops*6.0; // 12 bytes per term
35      printf("A %d x %d B %d x %d host time %.3f ms
               Gflops/sec %.3f\n",Arow,Acol,Brow,Bcol,t1,gflops);
36      return 0;
37    }
```

```
D:\ >hostmult0.exe
A 1024 x 1024 B 1024 x 1024 host time 2121.046 ms
                     GFlops 1.013 GBytes 6.076
```

Description of Example 2.11

- Line 1: Here we include the thrust `host_vector` header, `std::vector` could have been used instead for this host only code.
- Lines 4–12: This is the host matrix multiply function `hostmult0`, it takes standard pointers to the data for the matrices `C`, `A` and `B` as the first three arguments. The next three arguments define the

sizes of all three matrices. Note we use y and x instead of row and col to denote the first and second dimensions of the matrices. Thus, `A` is `ay` × `ax`, `B` is `ax` × `bx` and `C` is `ay` × `bx`, we infer the first dimension of `B` and both dimensions of `C` from the properties of matrix multiplication.
- Lines 7: These `for` loops over `i` and `j` cover all the elements of the desired product C.
- Line 9: The inner loop over `k` implements the summation from the standard formula. You can think of this summation as a dot product between the ith row of A and the jth column of B. Notice how the array indices vary with the `for` loop index `k`. The factor `A[i*Ax+k]` behaves "nicely" because as `k` increments, it addresses elements of `A` which are adjacent in memory, this is optimal for caching. On the other hand, the factor `B[k*Bx+j]` addresses memory with a stride of `Bx` words between successive values of `k`, which gives poor cache performance. This problem is inherent in matrix multiplication and has no simple fix.

  Notice also that a triple for loop is needed for matrix multiplication. If the matrices have dimensions of 10^3 then a total of 2×10^9 arithmetic operations are required – multiplication of big matrices is slow!

  You might worry that the expressions like `i*Bx+j` used for the array indices add a significant computational load for each step through the loop. In fact, this sort of index expression is so common that compilers are very good at generating the best possible code for indexing such arrays efficiently.

- Lines 13–36: This is the main routine:
  - Lines 15–20: Here we set the matrix sizes using optional user inputs for the dimensions of `A` (`Arow & Acol`) and the number of columns of `B` (`Bcol`). The dimensions of `C` and number of rows of `B` are set to be compatible with matrix multiplication.
  - Lines 21–23: Here we allocate thrust vectors to hold the matrices.
  - Lines 24–28: Here `A` and `B` are initialised with random numbers.
  - Lines 29–31: A timed call the `hostmult0` to perform the multiplication.
  - Lines 32–35: Print some results, the performance in GFlops/sec assumes two operations per iteration in line 9 and ignores all overheads.

The timing result in the last line shows that this calculation runs at about 1 GFlops/sec and is clearly memory bound. The memory bandwidth achieved is about 6 GBytes/sec (8 bytes read and 4 bytes written per term).

The performance of this code is quite poor but we can improve it significantly by adding the C++11 `__restrict` keyword to the pointer argument declarations in line 9. This is shown in Example 2.12 where only line 9 from Example 2.11 has been changed.

Example 2.12 `hostmult1` showing use of restrict keyword

```
     . . .
04   int hostmult1(float * __restrict C, float * __restrict A,
                   float * __restrict B, int Ay, int Ax, int Bx)
     . . .
D:\ > hostmult1.exe
A 1024 x 1024 B 1024 x 1024 host time 1468.845 ms
                          GFlops 1.462 GBytes 8.772
```

We have improved the performance by 44% making this simple change! If you are not familiar with the history of C/C++ this requires some explanation. When a function is declared with a simple pointer argument, the compiler has no way of being certain that there are no other pointers to the same memory location elsewhere in the code. This is called pointer aliasing and in the early days of C when computer memory was a scarce resource, people would deliberately use pointer aliasing to use the same piece of memory for different purposes at different stages in the program. Needless to say, this practice often resulted in hard-to-find bugs. On modern systems with 64-bit memory addressing pointer aliasing is completely unnecessary yet the memory of old practice lingers on in modern compilers which are still reluctant to fully optimise code involving simple pointers. Specifically, they will tend to unnecessarily store intermediate results back to main memory rather than using registers. Adding the `restrict` qualifier to a pointer declaration tells the compiler that the pointer is not aliased and aggressive optimisation is safe.[12]

The CUDA NVCC compiler also supports `restrict` and the performance of many kernels does indeed improve when it is used. Thus, we come to the conclusion shown in the box:

Always use **`restrict`** with pointer arguments.

As mentioned above, in practice C++ compiler support for restrict is quite shallow; if the restrict pointer is passed as a function argument wrapped in even a simple C++ class then restrict has no effect. In fact, while restrict is officially part of modern C11, it is not part of any C++ standard up to C++17. Fortunately, most recent C++ compilers including Visual Studio and g++ do support restrict, albeit in a shallow form. Another issue is that while the C standard uses `restrict` without decoration, C++ compilers may use `_restrict` or `__restrict` instead.

At this point we should mention another qualifier, `const`; many books on C++ get very excited about this. We discuss it in more detail in our C++ coding appendix. In principle use of `const` can allow the compiler to further optimise code. In practice we find using `const` does not usually give much or any performance gain; its use is actually more important as a safeguard to prevent accidental overwriting of variables and to make a programmer's intentions clear. In the case of pointers there are four possibilities shown in Table 2.4.

Table 2.4 *Possible combinations of const and restrict for pointer arguments*

declaration	cx wrapper	effect
`float * __restrict A`	`r_Ptr<float> A`	pointer variable, data variable
`const float * __restrict A`	`cr_Ptr<float> A`	pointer variable, data constant
`float * const __restrict A`	`cvr_Ptr<float> A`	pointer constant, data variable
`const float * const __restrict A`	`ccr_Ptr<float> A`	pointer constant, data constant

Example
```
int hostmult1(r_Ptr<float> C, cr_Ptr<float> A, cr_Ptr<float> B, int Ay,
int Ax, int Bx)
```

The middle column of the table shows templated wrappers defined in `cx.h` that can be used to hide the gory details in the first column. An example of their use is shown in the bottom row. These wrappers can be used in both host and kernel code and the first two will be used in most of our examples from now on.

It is now time to look at a GPU version of matrix multiply; to get the best performance is actually quite complicated but we will start with a simple approach as shown in Example 2.13.

Example 2.13 `gpumult0` kernel simple matrix multiplication on the GPU

```
__global__ void gpumult0(float * C, const float * A,
                    const float * B, int Ay, int Ax, int Bx)
{
  int tx = blockIdx.x*blockDim.x + threadIdx.x;  // col j
  int ty = blockIdx.y*blockDim.y + threadIdx.y;  // row i
  if(ty >= Ay || tx >= Bx) return;
  C[ty*Bx+tx] = 0.0;
  for(int k=0;k<Ax;k++) C[ty*Bx+tx] +=
                              A[ty*Bx+k]*B[k*Bx+tx];
}
. . .
int main(int argc, char *argv[])
{
  int Arow = (argc > 1) ? atoi(argv[2]) : 1024;
  int Acol = (argc > 2) ? atoi(argv[3]) : Arow;
  int Brow = Acol;
  int Bcol = (argc > 3) ? atoi(argv[4]) : Brow;
  int Crow = Arow;
  int Ccol = Bcol;
  uint tilex = (argc > 4) ? atoi(argv[5]) : 32;  // tile x
  uint tiley = (argc > 5) ? atoi(argv[6]) : 8;   // tile y

  thrust::host_vector<float>       A(Arow*Acol);
  thrust::host_vector<float>       B(Brow*Bcol);
  thrust::host_vector<float>       C(Crow*Ccol);
  thrust::device_vector<float> dev_C(Crow*Ccol);
  thrust::device_vector<float> dev_A(Arow*Acol);
  thrust::device_vector<float> dev_B(Brow*Bcol);

  // initialise A and B with random numbers
  std::default_random_engine gen(12345678);
  std::uniform_real_distribution<float> fran(0.0,1.0);
  for(int k = 0; k<Arow*Acol; k++) A[k] = fran(gen);
  for(int k = 0; k<Brow*Bcol; k++) B[k] = fran(gen);
  dev_A = A;  // H2D copy
  dev_B = B;  // H2D copy

  dim3 threads ={tilex, tiley, 1};
28.4   dim3 blocks ={(Bcol+threads.x-1)/threads.x,
                     (Arow+threads.y-1)/threads.y, 1};
29     cx::timer tim;
30     gpumult0<<<blocks,threads>>>(dev_C.data().get(),
                dev_A.data().get(), dev_B.data().get(),
                Arow, Acol, Bcol);
30.1   cudaDeviceSynchronize();  // wait for kernel
31     double t2 = tim.lap_ms();
31.1   C = dev_C;               // D2H copy

32     double flops = 2.0*Arow*Acol*Bcol;
33     double gflops = flops/(t2*1000000.0);
34     double gbytes = gflops*6.0; // 12 bytes per term
35     printf("A %d x %d B %d x %d gpu time %.3f ms
               GFlops %.3f GBytes %.3f\n",Arow,Acol,Brow,
               Bcol,t2,gflops,gbytes);
36     return 0;
37   }

D:\ >gpumult0.exe 1024 1024 1024 32 32
A 1024 x 1024 B 1024 x 1024 gpu time 6.685 ms
            GFlops 321.233 GBytes 1927.400
```

Description of Example 2.13

Much of the code in this example is identical to the host version shown in Example 2.11. Here we comment on the differences.

- Lines 4–11: The GPU kernel `gpumult0` replaces the previous `hostmult0` function here. The kernel is designed to use one thread to calculate one element of the matrix product. The kernel expects to be called with a 2D grid of thread blocks with sufficient threads in the x and y dimensions to span all the elements of C. As before `x` is the column index and `y` is the row index.
- Lines 6–7: Here we set `tx` and `ty` from the built-in variables to determine which element of C this thread will calculate. These lines effectively replace the loops over `i` and `j` used in the host version, we can think of the kernel as effectively calculating *all* the elements of C in parallel.
- Line 8: This is an out-of-range check on `tx` and `ty`. It is necessary because the dimensions of each thread block may have been rounded up.
- Lines 9–10: Here we calculate one element of C using the standard formula. Notice the factor `B[k*Bx+tx]` in line 10 still uses a memory stride of `Bx` words on successive passes through the `for` loop over `k`. But now in this parallel kernel *adjacent threads* will use adjacent elements of `B` because adjacent threads have adjacent values of `tx`. Thus L1 caching will be efficient for both factors in the multiplication – this is an interesting example of how parallel CUDA code can provide efficient memory access in situations where single threaded code struggles.
- Lines 20.1–20.2: We add two additional user settable parameters `tilex` and `tiley` which define the x and y dimensions of the thread blocks used by the kernel launch. These are equivalent to the `threads` and `blocks` parameters we use in many 1D examples.

- Lines 23.1–23.3: Here we allocate device arrays to hold copies of the matrices `A`, `B` and `C`.
- Lines 28.1–28.2: Copy `A` and `B` to the device.
- Line 28.3: Set `threads` to a `dim3` triad representing a 2D tile on the matrix `C`.
- Line 28.4: Set blocks as a `dim3` with x and y dimensions sufficient for the thread block tiles in threads to span the matrix `C`. Notice the integer rounding up for cases where the dimensions of `C` are not exact multiples of `tilex` and `tiley`. The out-of-range test in line 8 is necessary for cases where rounding up was needed. Rounding up and consequent testing in kernels are very common in CUDA code written to process general cases where not everything is a power of 2.
- Lines 29–31: This timed loop is similar to that of Example 2.9 but performs a kernel launch instead of a host function call. The use of `cudaDeviceSynchronize` is necessary for timing purposes.
- Line 31.1: Here we copy the result back to the host. Although C is not used in the code shown here, it would obviously be used in real-world code. Indeed, we have used C to compare the results from the host and GPU versions and find the calculated C_{ij} agree to about 6 significant figures.

The timing result in the last line shows that there is an impressive speed-up of about 220 times compared to the host calculation in Example 2.12.

If we change the `gpumult0` declaration in line 4 to use restrict we get the `gpumult1` kernel declaration shown in Example 2.14. This example shows two alternative methods of declaring restrict arrays in kernel code; the first method just uses C++ keywords and is quite verbose; the second method uses cx defined abbreviations for the same result. Note that `cr_Ptr` and `r_Ptr` are defined with templated `using` statements in `cx.h`. We will use the abbreviated versions in all our later examples.

Example 2.14 `gpumult1` kernel using restrict keyword on array arguments

```
       . . .
04     __global__ void gpumult1(float * __restrict C,
         const float * __restrict A, const float * __restrict B,
         int Ay, int Ax, int Bx)
or:
04    __global__ void gpumult1(r_Ptr<float> C, cr_Ptr<float> A,
           cr_Ptr<float> B, int Ay, int Ax, int Bx)

D:\ > gpumult1.exe
A 1024 x 1024 B 1024 x 1024 gpu time 2.536 ms
            GFlops 846.813 GBytes 5080.878
```

We can see that simply using restrict on our GPU matrix multiply code gives a dramatic speed-up of more than a factor of 2.6 (compared to about 1.4 for host code). The effective memory bandwidth is also much greater than the hardware limit of about 400 GBytes/sec separately for read and write, demonstrating that memory caching is playing an important role.

Finally, if you really hate the explicit address calculations in line 10 of Example 2.13 they can be hidden using a lambda function as shown in Example 2.15.

Example 2.15 `gpumult2` kernel using lambda function for 2D array indexing

```
04 __global__ void gpumult2(r_Ptr<float> C, cr_Ptr<float> A,
           cr_Ptr<float> B, int Ay, int Ax, int Bx)
05 {
06   int j = blockIdx.x*blockDim.x + threadIdx.x;  // col j
07   int i = blockIdx.y*blockDim.y + threadIdx.y;  // row i
08   if(i >= Ay || j >= Bx) return;

     // lambda function
09   auto idx = [&Bx](int i,int j){ return i*Bx+j; };

10   C[idx(i,j)] = 0.0;
11   for(int k=0;k<Ax;k++)
            C[idx(i,j)]+= A[idx(i,k)]*B[idx(k,j)];
12 }
```

In Example 2.15 we have added a new line 9 which defines a local function `idx` that performs the standard 2D address calculation needed for this function. The span needed step to successive rows of B and columns of C is `Bx`, the number of columns of B and (necessarily) also the number of rows of C. The value is captured by the lambda function using the `[&Bx]` syntax to indicate that the variable `Bx` used in the body of the lambda function is the same variable as used in the main body of the surrounding function. Moreover, by prefixing `&` to `Bx` we indicate that the variable is to be used by reference with no copy required. This should lead to the compiler generating code identical to Example 2.14, and indeed we find no performance difference between these two versions. Using a lambda function in this way is modernising the old trick of using a macro; in this case:

```
#define idx(i,j) (i)*Bx+(j)
```

which achieves the same effect. However, we deeply deprecate the use of macros in this way because even if the macro occurs inside a function its definition will persist throughout the code which risks hard-to-find bugs and greatly complicates code where different 2D spans are needed in different parts of the code. Note also the precautionary brackets around `i` and `j` in the macro definition, these are need for correctness if say `i` is passed as `i+1`.

2.9 Tiled Matrix Multiplication

Matrix multiplication is often used to demonstrate the use of shared memory in many CUDA tutorials and books. This is based on the observation that A_{ik} and B_{kj} for fixed k are read from main memory many times for each possible combination of the values of i and j. This was a major problem on early GPUs which had poor memory caching and it is still an issue on recent GPUs which have much better caching.

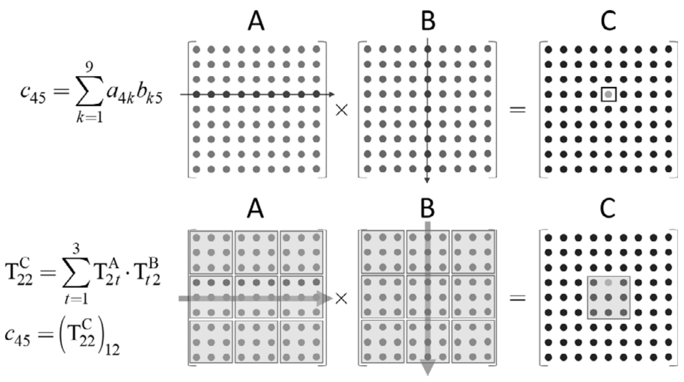
A
B
C
$c_{45} = \sum_{k=1}^{9} a_{4k} b_{k5}$
$T^{C}_{22} = \sum_{t=1}^{3} T^{A}_{2t} \cdot T^{B}_{t2}$
$c_{45} = \left(T^{C}_{22}\right)_{12}$

Figure 2.3 Tiled matrix multiplication

To exploit shared memory, we can use each thread in a thread block to store different elements of A and B in shared memory and then let all the threads in the block use all the cached values to calculate contributions to the product. This is relatively straightforward because matrix multiplication can be represented as a sum over products of 2D tiles defined over the matrices as shown in the following equation:

$$\mathbf{T}^{C}_{I,J} = \sum_{t=1}^{M_T} \mathbf{T}^{A}_{I,t} \cdot \mathbf{T}^{B}_{t,J},$$

where the **T**s are rectangular tiles defined over the matrices and · represents matrix multiplication between tiles. The summation index, t, represents a summation over contributing M_t pairs of tiles; the matrix product implies further summations over the individual matrix elements within the tiles. The idea is shown in Figure 2.3 where we use a set of 3 × 3 tiles to perform the multiplication of 9 × 9 matrices.

Tiled matrix multiplication can be readily implemented in CUDA kernels by using a 16 × 16 or 32 × 32 thread blocks to represent a pair of tiles from A and B. Each thread then first copies one element of its thread block's allocated A and B tiles into shared memory arrays. Once this process is complete, the same threads can then compute the elements of the tiled matrix multiplication to obtain that tile-pair's contribution to a tile in C. In this way each element of A and B is only read once from external memory instead of 16 or 32 times. Our implementation is the `gputiled` kernel shown in Example 2.16. In Figure 2.3 the element c_{45} is shown calculated conventionally in the top row and by tiled matrix multiplication in the bottom row.

Example 2.16 `gputiled` kernel: tiled matrix multiplication using shared memory

```
01  #include "cx.h"
02  #include "cxtimers.h"
03  #include <random>
    . . .
```

```
int main(int argc, char *argv[])
{
  . . .
  dim3 threads = {tilex,tilex,1};  // square
  dim3 blocks = {(Bcol+threads.x-1)/threads.x,
                 (Arow+threads.y-1)/threads.y, 1};
  . . .
  cx::timer tim;
  if(tilex == 8) gputiled< 8><<<blocks,threads>>>(
       dev_C.data().get(),dev_A.data().get(),
       dev_B.data().get(),Arow,Acol,Bcol);
  else if(tilex == 16) gputiled<16><<<blocks,threads>>>(
       dev_C.data().get(),dev_A.data().get(),
       dev_B.data().get(),Arow,Acol,Bcol);
  else if(tilex == 32) gputiled<32><<<blocks,threads>>>(
       dev_C.data().get(),dev_A.data().get(),
       dev_B.data().get(),Arow,Acol,Bcol);
  cudaDeviceSynchronize();
  double t3 = tim.lap_ms()
  . . .
}

template <int TS> __global__ void gputiled(
      float * __restrict C, float * __restrict A,
      float * __restrict B, int Ay, int Ax, int Bx)
{
  __shared__ float Atile[TS][TS];   // tile A e.g. [16][16]
  __shared__ float Btile[TS][TS];   // tile B e.g. [16][16]
  int tx  = threadIdx.x;            // tile col index j
  int ty  = threadIdx.y;            // tile row index i
  int ocx = blockDim.x*blockIdx.x;  // tile x origin in C
  int ocy = blockDim.y*blockIdx.y;  // tile y origin in C

  int ax = tx;      // j or x in first tile on A
  int ay = ocy+ty;  // i or y in first tile on A and C
  int bx = ocx+tx;  // j or x in first tile on B and C
  int by = ty;      // i or y in first tile on B

  float csum = 0.0f;
  for(int t=0; t<gridDim.x; t++){
    Atile[ty][tx] = A[ay*Ax+ax];  // copy A to shared mem
    Btile[ty][tx] = B[by*Bx+bx];  // copy B to shared mem
    __syncthreads();
    for(int k=0;k<TS;k++) csum += Atile[ty][k]*Btile[k][tx];
    __syncthreads();
    ax += TS;         // step A tiles along rows of A
    by += TS;         // step B tiles down  cols of B
61      }
62      C[ay*Bx+bx] = csum; // store complete result
63    }

D:\  > gputiled.exe  1024 1024 1024 32
A 1024 x 1024 B 1024 x 1024 gpu time 1.945 ms
                GFlops 1104.284 GBytes 6625.701
```

Description of Example 2.13

The kernel is shown in full as here the changes are significant. For the main routine only changes from Example 2.13 are shown.

- Line 28.3: We use `tilex` to set both dimensions of the 2D thread blocks used to represent tiles. While it is possible to use non-square tiles, that would complicate the kernel code.
- Lines 29–31: As before this is the timed block that launches a kernel and waits for completion. The kernel launch itself is now changed because the `guptiled` kernel is written to use the value of `tilex` as a template parameter. Here we use a 3-way if-else tree to allow values of 32, 16 or 8 for this parameter. The kernel argument list is the same as before.
- Line 40: This is the start of our new `guptiled` kernel; the arguments are as before and we are now using the `restrict` keyword by default for all pointers. Note that this is a templated kernel; thus the tile size parameter `TS` is known at compile time.
- Lines 42–43: We declare two statically allocated shared memory arrays to hold square tiles copied from `A` and `B` to `Atile` and `Btile`.
- Lines 44–45: Here we set the position of the current thread in the local `TS x TS` tiles. This depends only on the thread block dimensions.
- Lines 46–47: Here we set `ocx` and `ocy` to the origin of the target tile in `C` using grid-block quantities. These values are the same for all threads in the thread block.
- Lines 48–51: In the first two lines we set `ax` and `ay` to the current thread's position in `A` based on the first tile to be used. Similarly, in the second pair of lines we set `bx` and `by` for matrix `B`. Notice that as we step to different tiles along the rows of `A` and down the columns of `B` `ay` and `bx` are constant whereas `ax` and `by` change. In fact `ay` and `bx` are the i and j values of the c_{ij} element being evaluated by the current thread.
- Line 51: The local variable `csum` is used to accumulate the current thread's c_{ij} value; here we set it to zero.
- Lines 53–61: Each pass through this loop performs matrix multiplication on one pair of tiles from `A` and `B` and accumulates the result in `csum`.
  - Lines 54–55: Here we copy the current tiles from A and B to shared memory. Each thread copies one element from A and one from B to `Atile` and `Btile` and will later read `TS` values back from these arrays.
  - Line 56: An essential `syncthreads` here; no thread in the block can safely proceed until all the elements of `Atile` and `Btile` have been set.
  - Line 57: Matrix multiplication of `Atile` and `Btile`; each thread computes one element of the product.
  - Line 58: A second essential `syncthreads`; no thread can proceed to the next pass through the for loop until all threads have reached this point.

  - Lines 59–60: Here we increment `ax` and `by` to point to the required position in the next tiles from `A` and `B`.
- Line 62: Here we store the final result in C.

The result in the last line shows that `gputiled` delivers more than 1 TFlop/sec of processing. A tile size of 32 × 32 works best on the RTX 2070 GPU used for this test.

We note that using shared memory as shown in Example 2.16 gives a significant performance boost of about 250 GFlops/sec amounting to about 1.1 TFlops/sec overall. Although not shown here, we did try running this example without using `restrict` in kernel arguments and found only a small drop in performance. This is presumably because we now read from `A` and `B` fewer times and hence the performance gain from using `restrict` on the pointers to these arguments is less important.

There is one last trick we can try to squeeze a bit more performance from our code and that is explicit loop unrolling:

Example 2.17 `gputiled1` kernel showing explicit loop unrolling

```
         . . .
52       float csum = 0.0f;
52.1     #pragma unroll 16
         // step A tiles along rows of A
53       for(int t=0;t<gridDim.x;t++){
         . . .

D:\ gputiled1.exe  1024 1024 1024 32
A 1024 x 1024 B 1024 x 1024 gpu time 1.765 ms
                  GFlops 1216.958 GBytes 7301.748
```

As shown in the last line of Example 2.17 we have gained another 100 GFlops/sec of performance by using loop unrolling. The optimal depth of unrolling can only be found by experiment; on our RTX 2070 the value 16 seems to give the best result. On other GPUs you may find a different optimum. Tuning GPU code always involves some experimentation. Note the NVCC compiler will often automatically perform loop unrolling and especially in cases where the number of passes is known at compile time. For this reason, making the loop counter a template parameter can be worthwhile. Here this is done for the inner loop over `TS` but not for the outer loop over `gridDim.x` which is therefore not known at compile time. Interestingly, we find that explicit unrolling over the outer loop helps but in experiments (not shown) we found explicit unrolling over the inner loop does not help.

2.10 BLAS

Matrix multiplication is a classic problem in computational linear algebra and the results of more than 50 years of development are encapsulated in the BLAS (basic linear algebra subprograms) function libraries that are available for all serious computing platforms. BLAS is used by calling appropriate functions to perform the desired operations. Matrix multiplication of float-4 matrices can be performed by calling the `sgemm` (single precision general matrix multiplication) routine which implements the saxpy-like operation $\mathbf{C} = \alpha\mathbf{AB} + \beta\mathbf{C}$ for matrices **A**, **B** and **C**. The good news is that BLAS is available for CUDA code. The NVIDIA cuBLAS library is a set of host callable routines that run BLAS functions on the GPU using vectors and matrices in GPU memory. In fact, although cuBLAS provides its own routines to allocate and transfer arrays between host and GPU memories it is also perfectly possible to use thrust (or any other method) to manage these arrays. Thus cuBLAS can be used in our matrix multiply example with just a few modifications. Example 2.18 shows how BLAS routines can be used in host code to replace the kernel calls used in our previous examples.

Example 2.18 Host code showing matrix multiplication using cuBLAS

```
. . .
#include "cublas_v2.h"
. . .
int main(int argc, char *argv[])
{
. . .
  thrust::host_vector<float>       A(Arow*Acol);
  thrust::host_vector<float>       B(Brow*Bcol);
  thrust::host_vector<float>       C(Crow*Ccol);
  thrust::device_vector<float> dev_A(Arow*Acol);
  thrust::device_vector<float> dev_B(Brow*Bcol);
  thrust::device_vector<float> dev_C(Crow*Ccol);
  thrust::device_vector<float> dev_D(Crow*Ccol);

  // initialise A and B with random numbers, clear C
  std::default_random_engine gen(12345678);
  std::uniform_real_distribution<float> fran(0.0,1.0);
  for(int k = 0; k<Arow*Acol; k++) A[k] = fran(gen);
  for(int k = 0; k<Brow*Bcol; k++) B[k] = fran(gen);
  for(int k = 0; k<Crow*Ccol; k++) C[k] = 0.0f;

  dev_A = A;  // H2D copy
  dev_B = B;  // H2D copy
  dev_C = C;  // clear

  float alpha = 1.0f;
  float beta  = 1.0f;
  cublasHandle_t handle; cublasCreate(&handle);
  // enable tensor cores
  cublasSetMathMode(handle,CUBLAS_TENSOR_OP_MATH);

  cx::timer tim;    // C = alpha*(A*B) + beta*C
41     cublasSgemm(handle, CUBLAS_OP_T, CUBLAS_OP_T,
              Crow, Ccol, Arow, alpha, dev_A.data().get(),
              Acol, dev_B.data().get(), Bcol, &beta,
              dev_C.data().get(), Crow);
42     beta = 0.0f;
       // D = transpose(C) from C = alpha*A+beta*B
43     cublasSgeam(handle, CUBLAS_OP_T, CUBLAS_OP_T,
              Crow, Ccol, &alpha, dev_C.data().get(),
              Crow, &beta, dev_C.data().get(),
              Crow, dev_D.data().get(), Ccol);
44     cudaDeviceSynchronize();
45     double t3 = tim.lap_ms()/(double)(nacc);

46     C = dev_D; // D2H copy
47     double flops = 2.0*(double)Arow*(double)Acol*(double)Bcol;
48     double gflops = flops/(t3*1000000.0);
49     double gbytes = gflops*6.0; // i.e 12 bytes per term
50     printf("A %d x %d B %d x %d gpu time %.3f ms
            GFlops %.3f GBbytes %.3f\n", Arow, Acol, Brow,
            Bcol, t3, gflops, gbytes);
51     return 0;
52   }

D:\ > blasmult.exe 1024
A 1024 x 1024 B 1024 x 1024 time 0.318 ms
            GFlops 6747.1 GBytes 40482.8 (no TC)
A 1024 x 1024 B 1024 x 1024 time 0.242 ms
            GFlops 8882.3 GBytes 53293.9 (with TC)
```

Description of Example 2.18

- Line 5: The include file `cublas_v2.h` is necessary to use the NVIDIA BLAS library. It is also necessary to use the include file `cublas.lib` in the linking stage on Windows. An older version `cublas.h` file is also supplied for backward compatibility but that version is now deprecated for new code.
- Lines 12–18 (not shown): Set the sizes of the `A`, `B` and `C` matrices using optional user supplied values as before.
- Lines 20–26: Create thrust containers for the matrices for the host and device. An additional device matrix `dev_D` the same size as the result matrix `C` is created here. It will be needed to hold the transpose of `dev_C` as explained below.
- Lines 27–35: Initialise the matrices `A` and `B` to random numbers and set `C` to zeros. (Clearing `C` is technically unnecessary as this is thrust's default allocation option, but we like to make our intentions clear). The host vectors are then copied to the corresponding device vectors.
- Lines 36–37: The parameters alpha and beta are declared and set to one.
- In line 38: A `cublasHandle_t` object `handle` is created; this is a necessary first argument to nearly all of the library functions. It is useful in multithreaded applications where separate threads use different handles.

- Line 39: Because we are using a GPU equipped with tensor cores (i.e. having a CC of 7.0 or above) we tell `cuBLAS` to use them where possible. Although originally realised as a tool for mixed precision operations on 2 and 4-byte floats, recent versions of `cublas` can also use tensor cores to speed up pure 4-byte float calculations. In a test we see a speed-up of about 30% using 1024 $\times$ 1024 matrices and speed-ups of up to a factor of two using larger matrices.
- Lines 40–45: This is the timed loop where the matrix product is calculated. The BLAS library is great for performance but the functions have a dreadful user interface essentially unchanged from their early Fortran origins in the 1950s. Also, they keep to the Fortran convention of expecting matrices in column major format (i.e. elements in the same column are stored in adjacent memory locations). This means that default C/C++ style matrices, in row major format, are treated as if they had been transposed. While this does not matter for simple operations such as addition, it does matter for matrix multiplication. Fortunately, the matrix functions such as `cublasSgemm` (the `cuBLAS` version of `sgemm`), used in line 41, have flag arguments specifying whether the input matrices `A` and `B` should be transposed before use. This results in correct matrix multiplication but the resulting matrix `C` is still left in column major format. We correct this in line 43 by calling the `cuBLAS` function `cublasSgeam` to transpose `C` back to row major format.
- Line 41: The call the `cubalsSgemm` function has many arguments as follows:
  1. The mandatory `cuBLAS` handle
  2. Transpose `A` if `CUBLAS_OP_T` or not if `CUBLAS_OP_N`
  3. Transpose `B` if `CUBLAS_OP_T` or not if `CUBLAS_OP_N`
  4. The number of rows of `A` (after transposition if done) and `C`; we use `Crow` here.
  5. The number of columns of `B` (after transposition if done) and `C`; we use `Ccol` here.
  6. The number of columns of `A` (after transposition if done) and rows of `B` (after transposition if done), we use `Arow` here. This is the index that is summed in matrix multiplication.
  7. Pointer to the scaling factor `alpha`.
  8. Pointer to the matrix `A`.
  9. Leading dimension of array used to hold `A`; we use `Acol` here.
  10. Pointer to the matrix `B`.
  11. Leading dimension of the array used to hold `B`; we use `Bcol` here.
  12. Pointer to the scaling factor `beta`.
  13. A pointer to the matrix `C`.
  14. Leading dimension of the array used to store `C`.

For square matrices all the dimensions are the same and the interface is relatively forgiving; in other cases significant care is required to get everything correct. We have allowed for the transposition of `A` and `B` in our choice for argument 6 but not arguments 9 and 11. We have tacitly assumed that `C` is in column major format for our choice of arguments 5, 6 and 14.

- Line 42: Set `beta` to zero before calling `cublasSgeam.`
- Line 43: Here we use the `cublasSgeam` function which evaluates $\mathbf{C} = \alpha\mathbf{A} + \beta\mathbf{B}$ to transpose `C`. This function is an NVIDIA extension to the standard set of BLAS functions. By setting $\alpha = 1$ and $\beta = 0$ we cause `A` to be copied to `C` with optional transposition of `A` if requested. The arguments for `cublasSgeam` are as follows:

  1–5. Same as `cublasSgemm`.
  6. Pointer to `alpha`.
  7. Pointer to the matrix `A`; we use `C` here.
  8. Leading dimension of array used to hold first matrix; we use `Crow` here.
  9. Pointer to `beta`. Note `beta` is set to zero in line 42.
  10. Pointer to the matrix `B`; we use `C` here.

11. Leading dimension of array used to hold `B` matrix; we use `Crow` here.
12. Pointer to the matrix `C`; we use `D` here.
13. Leading dimension of array used to hold `C` matrix; we use `Ccol` here. (This would be `Dcol` in cases where `C` and `D` had different sizes).

- Lines 44–52: These are similar to before.

The results at the end of the example show performances of 6.7 and 8.9 TFlops for the RTX2070 and 1024×1204 matrices. The latter result was obtained using the tensor core processors available on devices of CC ≥ 7.0.

The performance of the `cublasSgemm` is a factor of 6 or more better than our best kernel. Moreover, tensor cores, if available, can be used to give further impressive speed-ups of ~30% or more. Thus, while matrix multiplication is an excellent and much used calculation for demonstrating the use of shared memory in CUDA kernels, if you really need lots of fast matrix multiplication, use the NVIDA library not your own kernels. Similar advice applies to other standard problems such as FFT for which NVIDIA also has a good library. We discuss NVIDIA's full range of libraries in Appendix F.

Figure 2.4 shows how the performance of our matrix multiply routines varies as a function of matrix size. The peak performance for the largest matrix sizes is over 15 TFlops for `cublasSgemm` with tensor cores. The curve labelled kernel corresponds to the `gputiled1` kernel. The curves labelled blas and blas+TC correspond to the two BLAS routines. Note the peak performance achieved by the TC version of cuBLAS for the largest matrices is over 15 TFlops; this is an astonishing performance from a £400 PC card.

In Chapter 11 we show you how to write your own matrix multiply kernels using tensor cores; the shared memory version achieves about 5.6 TFlops compared to the 8.9 achieved by cuBLAS. This is actually not bad as the cuBLAS library routines will contain many

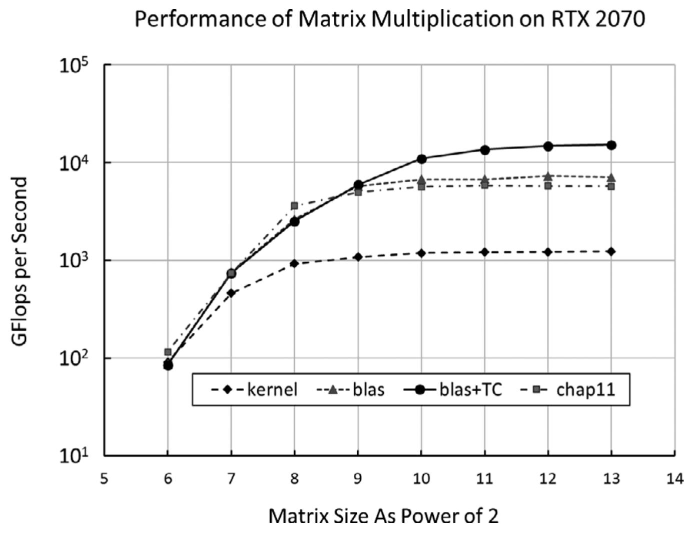

Figure 2.4 Performance of matrix multiplication on an RTX 2070 GPU

detailed optimisations. The performance of this kernel is shown as the chap11 curve on Figure 2.4.

The fact that fast libraries for standard calculations like matrix multiplication are available does not mean that learning to write your own kernels in CUDA is unnecessary; there are many situations where an out of the box solution is not readily available. For example, simulation is very important in many areas. Models of your particular problem often have to be hand-crafted and could gain enormously in speed if you are able to include code that exploits the raw power of GPUs. We also note from Figure 2.4 that our matrix multiply kernels are more competitive with `cuBLAS` for smaller matrices which might be useful when many small matrix multiplications are required as part of a bigger program.

This concludes our second introductory chapter. In the next chapter we discuss warp level programming which will provide important insights into how to get the best performance from your GPU. The examples in Chapter 3 include further improvements to the reduce kernels.

Endnotes Chapter 2

1 Flynn, Michael J. "Some computer organizations and their effectiveness." *IEEE transactions on computers* 100, no. 9 (1972): 948–960.

2 In practice the addition of floating-point numbers is not precisely commutative because the accumulation of rounding errors can depend on the order in which the terms are added together. Once the sum gets large, the contribution from subsequent small numbers is inaccurate or completely lost. This is a particular issue for F32 where only about 7 significant figures are useful. Interestingly parallel reduction techniques, where a number of partial sums are accumulated in parallel, are likely to be more robust than a single serial evaluation.

3 I first encountered this approach to parallel programming when learning MPI in the mid-1990s and it was a revelation. My previous encounters with trying to program multiple devices to run in parallel had involved writing different programs for each device and hand tuning at the assembly level to make the execution times identical on each device (MIMD) – a nightmare task compared to the common code SIMD model of CUDA and MPI.

4 Specifically, the member functions the host and device vector class do not have `__device__` definitions. Thrust was designed as a suite of host callable functions which ran on the GPU for speed. Users of thrust were not expected to write their own kernels.

5 Our recommendation that threads should be a multiple of 32 is for performance reasons. Any value in [1,1024] is allowed, it is just that values which are multiples of the warp size are more efficient. For example, if you specified 48 then every thread block would be run with one full warp of 32 threads and one ½ full warp of 16 threads leading to a 25% performance loss.

6 If you want your compiled code to run on different GPU models you can use the device query functions in CUDA to find the value of `Nsm` at run time.

7 As a technical aside, we mention that the GPU hardware manages branch divergence at the warp-engine level by maintaining a 32-bit active-thread bit mask for each active warp, the bits are turned on or off to determine with threads execute in the currently scheduled instruction.

8 The Cooperative Groups feature does allow grid-wide synchronisation of all threads in a grid during kernel execution but only if a number of restrictions are applied including having all thread blocks resident on the device at once.

9 The modern C++ practice of declaring and initialising objects in the same statement (RAII) cannot be applied to shared memory objects in CUDA kernels because the declaration is the same for all threads in the kernel, but the initialisation is usually thread dependent.

10 Actually, it should be second nature for you as a C++ programmer to always use longest first/shortest last ordering in variable declarations for all your classes and structs as well as special cases like CUDA dynamically declared arrays. This will achieve natural alignment for all your variables without the compiler having to insert "hidden" padding.

11 Giving up on containers for kernel arguments means that we have to pass array dimensions explicitly as separate arguments. This is a genuine loss and is a potential source of bugs. One advantage of containers is that objects know their sizes.

12 Of course, if you use `restrict` it is your responsibility to ensure that aliasing does not occur – the compiler still cannot actually check this.