CHAPTER

Grids, Blocks, and Threads

5

WHAT IT ALL MEANS

NVIDIA chose a rather interesting model for its scheduling, a variant of SIMD it calls SPMD (single program, multiple data). This is based on the underlying hardware implementation in many respects. At the heart of parallel programming is the idea of a thread, a single flow of execution through the program in the same way a piece of cotton flows through a garment. In the same way threads of cotton are woven into cloth, threads used together make up a parallel program. The CUDA programming model groups threads into special groups it calls warps, blocks, and grids, which we will look at in turn.

THREADS

A thread is the fundamental building block of a parallel program. Most C programmers are familiar with the concept if they have done any multicore programming. Even if you have never launched a thread in any code, you will be familiar with executing at least one thread, the single thread of execution through any serial piece of code.

With the advent of dual, quad, hex core processors, and beyond, more emphasis is explicitly placed on the programmer to make use of such hardware. Most programs written in the past few decades, with the exception of perhaps the past decade, were single-thread programs because the primary hardware on which they would execute was a single-core CPU. Sure, you had clusters and supercomputers that sought to exploit a high level of parallelism by duplicating the hardware and having thousands of commodity servers instead of a handful of massively powerful mac
ines. However, these were mostly restricted to universities and large institutions, not generally available to the masses.

Thinking in terms of lots of threads is hard. It's much easier to think in terms of one task at a time. Serial programming languages like C/C++ were born from a time when serial processing speed doubled every few years. There was little need to do the hard parallel programming. That stopped almost a decade ago, and now, like it or not, to improve program speed requires us to think in terms of parallel design.

Problem decomposition

Parallelism in the CPU domain tends to be driven by the desire to run more than one (single-threaded) program on a single CPU. This is the task-level parallelism that we covered earlier. Programs, which are data intensive, like video encoding, for example, use the data parallelism model and split the task in *N* parts where *N* is the number of CPU cores available. You might, for example, have each CPU core calculate one "frame" of data where there are no interdependencies between frames. You may also choose to split each frame into *N* segments and allocate each one of the segments to an individual core.

In the GPU domain, you see exactly these choices when attempting to speed up rendering of 3D worlds in computer games by using more than one GPU. You can send complete, alternate frames to each GPU (Figure 5.1). Alternatively, you can ask one GPU to render the different parts of the screen.

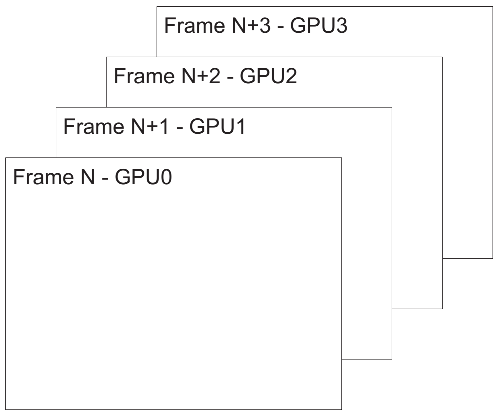


FIGURE 5.1

Alternate frame rendering (AFR) vs. Split Frame Rendering (SFR).

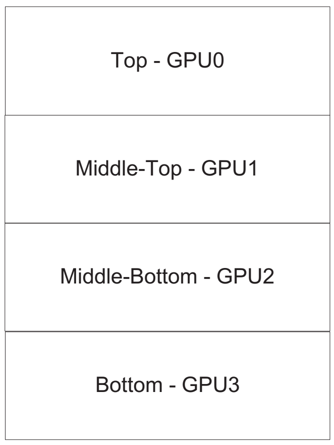


FIGURE 5.2

Coarse-grained parallelism.

However, there is a trade off here. If the dataset is self-contained, you can use less memory and transfer less data by only providing the GPU (or CPU) with the subset of the data you need to calculate. In the SFR GPU example used here, there may be no need for GPU3, which is rendering the floor to know the content of data from GPU0, which is probably rendering the sky. However, there may be shadows from a flying object, or the lighting level of the floor may need to vary based on the time of day. In such instances, it might be more beneficial to go with the alternate frame rendering approach because of this shared data.

We refer to SFR type splits as coarse-grained parallelism. Large chunks of data are split in some way between *N* powerful devices and then reconstructed later as the processed data. When designing applications for a parallel environment, choices at this level seriously impact the performance of your programs. The best choice here is very much linked to the actual hardware you will be using, as you will see with the various applications we develop throughout this book.

With a small number of powerful devices, such as in CPUs, the issue is often how to split the workload evenly. This is often easier to reason with because you are typically talking about only a small number of devices. With huge numbers of smaller devices, as with GPUs, they average out peaks in workload much better, but suffer from issues around synchronization and coordination.

In the same way as you have macro (large-scale) and micro (small-scale) economics, you have coarse and fine-grained parallelism. However, you only really find fine-grained parallelism at the programmer level on devices that support huge numbers of threads, such as GPUs. CPUs, by contrast, also support threads, but with a large overhead and thus are considered to be useful for more coarse-grained parallelism problems. CPUs, unlike GPUs, follow the MIMD (Multiple Instruction Multiple Data) model in that they support multiple independent instruction streams. This is a more flexible approach, but incurs additional overhead in terms of fetching multiple independent instruction streams as opposed to amortizing the single instruction stream over multiple processors.

To put this in context, let's consider a digital photo where you apply an image correction function to increase the brightness. On a GPU you might choose to assign one thread for every pixel in the image. On a quad-core CPU, you would likely assign one-quarter of the image to each CPU core.

How CPUs and GPUs are different

GPUs and CPUs are architecturally very different devices. CPUs are designed for running a small number of potentially quite complex tasks. GPUs are designed for running a large number of quite simple tasks. The CPU design is aimed at systems that execute a number of discrete and unconnected tasks. The GPU design is aimed at problems that can be broken down into thousands of tiny fragments and worked on individually. Thus, CPUs are very suitable for running operating systems and application software where there are a vast variety of tasks a computer may be performing at any given time.

CPUs and GPUs consequently support threads in very different ways. The CPU has a small number of registers per core that must be used to execute any given task. To achieve this, they rapidly context switch between tasks. Context switching on CPUs is expensive in terms of time, in that the entire register set must be saved to RAM and the next one restored from RAM. GPUs, by comparison, also use the same concept of context switching, but instead of having a single set of registers, they have multiple banks of registers. Consequently, a context switch simply involves setting a bank selector to switch in and out the current set of registers, which is several orders of magnitude faster than having to save to RAM.

Both CPUs and GPUs must deal with stall conditions. These are generally caused by I/O operations and memory fetches. The CPU does this by context switching. Providing there are enough tasks and the runtime of a thread is not too small, this works reasonably well. If there are not enough processes to keep the CPU busy, it will idle. If there are too many small tasks, each blocking after a short period, the CPU will spend most of its time context switching and very little time doing useful work. CPU scheduling policies are often based on time slicing, dividing the time equally among the threads. As the number of threads increases, the percentage of time spent context switching becomes increasingly large and the efficiency starts to rapidly drop off.

GPUs are designed to handle stall conditions and expect this to happen with high frequency. The GPU model is a data-parallel one and thus it needs thousands of threads to work efficiently. It uses this pool of available work to ensure it always has something useful to work on. Thus, when it hits a memory fetch operation or has to wait on the result of a calculation, the streaming processors simply switch to another instruction stream and return to the stalled instruction stream sometime later.

One of the major differences between CPUs and GPUs is the sheer number of processors on each device. CPUs are typically dual- or quad-core devices. That is to say they have a number of execution cores available to run programs on. The current Fermi GPUs have 16 SMs, which can be thought of a lot like CPU cores. CPUs often run single-thread programs, meaning they calculate just a single data point per core, per iteration. GPUs run in parallel by default. Thus, instead of calculating just a single data point per SM, GPUs calculate 32 per SM. This gives a 4 times advantage in terms of number of cores (SMs) over a typical quad core CPU, but also a 32 times advantage in terms of data throughput. Of course, CPU programs can also use all the available cores and extensions like MMX, SSE, and AVX. The question is how many CPU applications actually use these types of extensions.

GPUs also provide something quite unique—high-speed memory next to the SM, so-called shared memory. In many respects this implements the design philosophy of the Connection Machine and the Cell processor, in that it provides local workspace for the device outside of the standard register file. Thus, the programmer can leave data in this memory, safe in the knowledge the hardware will not evict it behind his or her back. It is also the primary mechanism communication between threads.

Task execution model

There are two major differences in the task execution model. The first is that groups of *N* SPs execute in a lock-step basis (Figure 5.3), running the *same* program but on different data. The second is that, because of this huge register file, switching threads has effectively *zero* overhead. Thus, the GPU can support a very large number of threads and is designed in this way.

Now what exactly do we mean by lock-step basis? Each instruction in the instruction queue is dispatched to every SP within an SM. Remember each SM can be thought of as single processor with *N* cores (SPs) embedded within it.

A conventional CPU will fetch a separate instruction stream for each CPU core. The GPU SPMD model used here allows an instruction fetch for *N* logical execution units, meaning you have 1/*N* the instructions memory bandwidth requirements of a conventional processor. This is a very similar approach to the vector or SIMD processors found in many high-end supercomputers.

However, this is not without its costs. As you will see later, if the program does not follow a nice neat execution flow where all *N* threads follow the same control path, for each branch, you will require additional execution cycles.

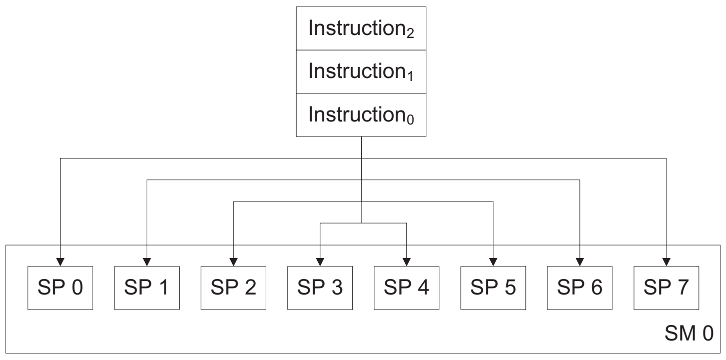


FIGURE 5.3

Lock-step instruction dispatch.

Threading on GPUs

So coming back to threads, let's look at a section of code and see what this means from a programming perspective.

```
void some_func(void)
{
  int i;

  for (i=0;i<128;i++)
  {
    a[i] = b[i] * c[i];
  }
}
```

This piece of code is very simple. It stores the result of a multiplication of `b` and `c` value for a given index in the result variable `a` for that same index. The `for` loop iterates 128 times (indexes 0 to 127). In CUDA you could translate this to 128 threads, each of which executes the line

```
a[i] = b[i] * c[i];
```

This is possible because there is no dependency between one iteration of the loop and the next. Thus, to transform this into a parallel program is actually quite easy. This is called loop parallelization and is very much the basis for one of the more popular parallel language extensions, OpenMP.

On a quad-core CPU you could also translate this to four blocks, where CPU core 1 handles indexes 0–31, core 2 indexes 32–63, core 3 indexes 64–95, and core 4 indexes 96–127. Some compilers will either automatically translate such blocks or translate them where the programmer marks that this loop can be parallelized. The Intel compiler is particularly good at this. Such compilers can be used to create embedded SSE instructions to vectorize a loop in this way, in addition to spawning multiple threads. This gives two levels of parallelism and is not too different from the GPU model.

In CUDA, you translate this loop by creating a kernel function, which is a function that executes on the GPU *only* and cannot be executed directly on the CPU. In the CUDA programming model the CPU handles the serial code execution, which is where it excels. When you come to a computationally intense section of code the CPU hands it over to the GPU to make use of the huge computational power it has. Some of you might remember the days when CPUs would use a floating-point coprocessor. Applications that used a large amount of floating-point math ran many times faster on machines fitted with such coprocessors. Exactly the same is true for GPUs. They are used to accelerate computationally intensive sections of a program.

The GPU kernel function, conceptually, looks identical to the loop body, but with the loop structure removed. Thus, you have the following:

```
__global__ void some_kernel_func(int * const a, const int * const b, const int * const c)
{
  a[i] = b[i] * c[i];
}
```

Notice you have lost the loop and the loop control variable, `i`. You also have a `__global__` prefix added to the C function that tells the compiler to generate GPU code and not CPU code when compiling this function, and to make that GPU code globally visible from within the CPU.

The CPU and GPU have separate memory spaces, meaning you cannot access CPU parameters in the GPU code and vice versa. There are some special ways of doing exactly this, which we'll cover later in the book, but for now we will deal with them as separate memory spaces. As a consequence, the global arrays `a`, `b`, and `c` at the CPU level are no longer visible on the GPU level. You have to declare memory space on the GPU, copy over the arrays from the CPU, and pass the kernel function pointers to the GPU memory space to both read and write from. When you are done, you copy that memory back into the CPU. We'll look at this a little later.

The next problem you have is that `i` is no longer defined; instead, the value of `i` is defined for you by the thread you are currently running. You will be launching 128 instances of this function, and initially this will be in the form of 128 threads. CUDA provides a special parameter, different for each thread, which defines the thread ID or number. You can use this to directly index into the array. This is very similar to MPI, where you get the process rank for each process.

The thread information is provided in a structure. As it's a structure element, we will store it in a variable, `thread_idx` for now to avoid having to reference the structure every time. Thus, the code becomes:

```
__global__ void some_kernel_func(int * const a, const int * const b, const int * const c)
{
  const unsigned int thread_idx = threadIdx.x;
  a[thread_idx] = b[thread_idx] * c[thread_idx];
}
```

Note, some people prefer `idx` or `tid` as the name for the thread index since these are somewhat shorter to type.

What is happening, now, is that for thread 0, the `thread_idx` calculation returns 0. For thread 1, it returns 1, and so on, up to thread 127, which uses index 127. Each thread does exactly two reads from memory, one multiply and one store operation, and then terminates. Notice how the code executed by each thread is identical, but the data changes. This is at the heart of the CUDA and SPMD model.

In OpenMP and MPI, you have similar blocks of code. They extract, for a given iteration of the loop, the thread ID or thread rank allocated to that thread. This is then used to index into the dataset.

A peek at hardware

Now remember you only actually have *N* cores on each SM, so how can you run 128 threads? Well, like the CPU, each thread group is placed into the SM and the *N* SPs start running the code. The first thing you do after extracting the thread index is fetch a parameter from the `b` and `c` array. Unfortunately, this doesn't happen immediately. In fact, some 400–600 GPU clocks can go by before the memory subsystem comes back with the requested data. During this time the set of *N* threads gets suspended.

Threads are, in practice, actually grouped into 32 thread groups, and when all 32 threads are waiting on something such as memory access, they are suspended. The technical term for these groups of threads is a warp (32 threads) and a half warp (16 threads), something we'll return to later.

Thus, the 128 threads translate into four groups of 32 threads. The first set all run together to extract the thread ID and then calculate the address in the arrays and issue a memory fetch request (see

Figure 5.4). The next instruction, a multiply, requires both operands to have been provided, so the thread is suspended. When all 32 threads in that block of 32 threads are suspended, the hardware switches to another warp.

In Figure 5.5, you can see that when warp 0 is suspended pending its memory access completing, warp 1 becomes the executing warp. The GPU continues in this manner until all warps have moved to the suspended state (see Figure 5.6).

Prior to issuing the memory fetch, fetches from consecutive threads are usually coalesced or grouped together. This reduces the overall latency (time to respond to the request), as there is an overhead associated in the hardware with managing each request. As a result of the coalescing, the memory fetch returns with the data for a whole group of threads, usually enough to enable an entire warp.

These threads are then placed in the ready state and become available for the GPU to switch in the next time it hits a blocking operation, such as another memory fetch from another set of threads.

Having executed all the warps (groups of 32 threads) the GPU becomes idle waiting for any one of the pending memory accesses to complete. At some point later, you'll get a sequence of memory blocks being returned from the memory subsystem. It is likely, but not guaranteed, that these will come back in the order in which they were requested.

Let's assume that addresses 0–31 were returned at the same time. Warp 0 moves to the ready queue, and since there is no warp currently executing, warp 0 automatically moves to the executing state (see Figure 5.7). Gradually all the pending memory requests will complete, resulting in all of the warp blocks moving back to the ready queue.

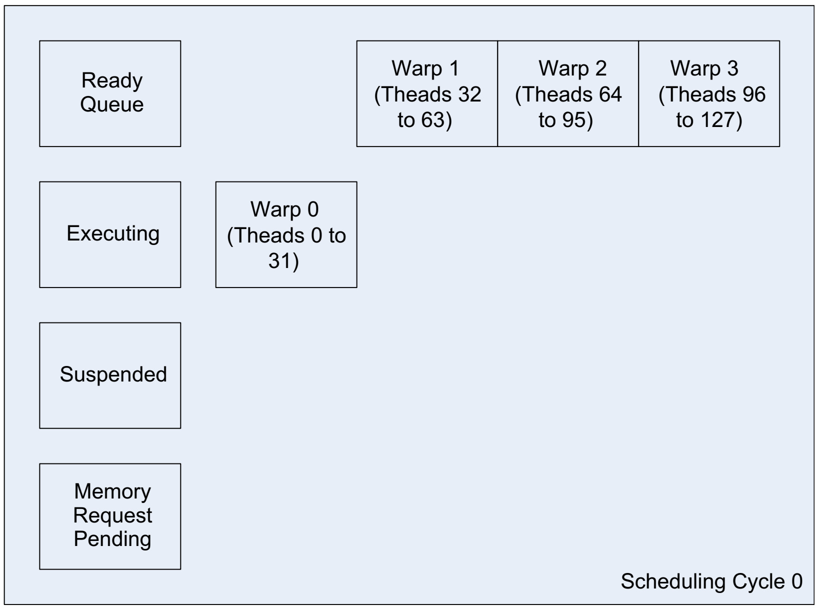


FIGURE 5.4

Cycle 0.

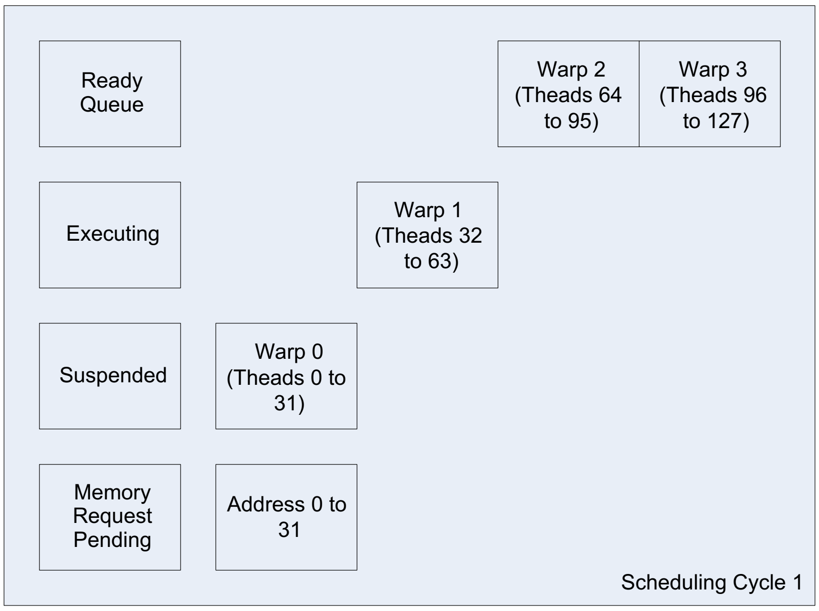


FIGURE 5.5

Cycle 1.

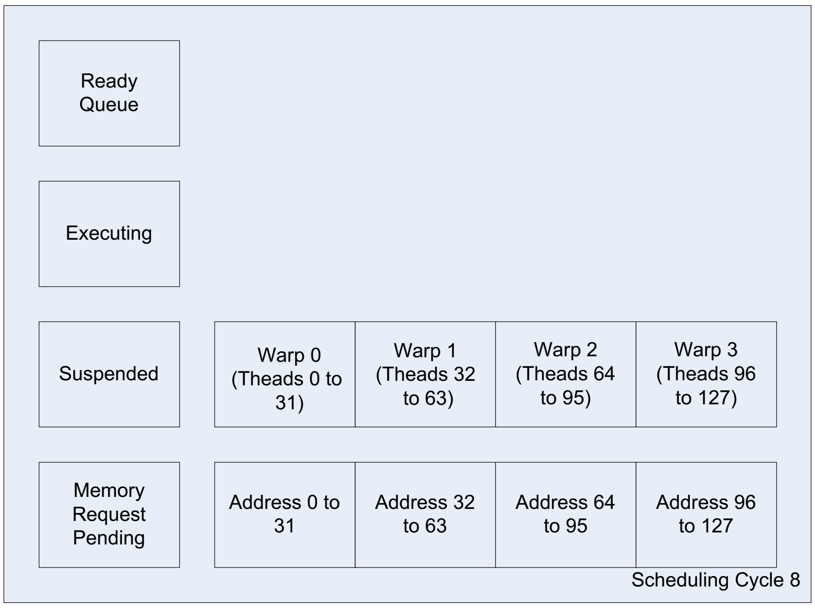


FIGURE 5.6

Cycle 8.

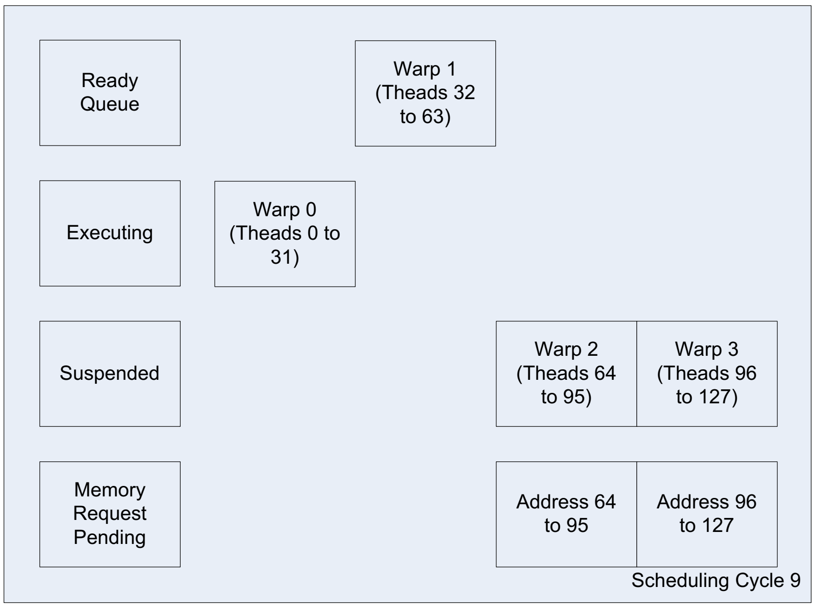


FIGURE 5.7

Cycle 9.

Once warp 0 has executed, its final instruction is a write to the destination array `a`. As there are no dependent instructions on this operation, warp 0 is then complete and is retired. The other warps move through this same cycle and eventually they have all issued a store request. Each warp is then retired, and the kernel completes, returning control to the CPU.

CUDA kernels

Now let's look a little more at how exactly you invoke a kernel. CUDA defines an extension to the C language used to invoke a kernel. Remember, a kernel is just a name for a function that executes on the GPU. To invoke a kernel you use the following syntax:

```
kernel_function<<<num_blocks, num_threads>>>(param1, param2, ...)
```

There are some other parameters you can pass, and we'll come back to this, but for now you have two important parameters to look at: `num_blocks` and `num_threads`. These can be either variables or literal values. I'd recommend the use of variables because you'll use them later when tuning performance.

The `num_blocks` parameter is something you have not yet covered and is covered in detail in the next section. For now all you need to do is ensure you have at least one block of threads.

The `num_threads` parameter is simply the number of threads you wish to launch into the kernel. For this simple example, this directly translates to the number of iterations of the loop. However, be aware that the hardware limits you to 512 threads per block on the early hardware and 1024 on the later hardware. In this example, it is not an issue, but for any real program it is almost certainly an issue. You'll see in the following section how to overcome this.

The next part of the kernel call is the parameters passed. Parameters can be passed via registers or constant memory, the choice of which is based on the compilers. If using registers, you will use one register for every thread per parameter passed. Thus, for 128 threads with three parameters, you use 3 × 128 = 384 registers. This may sound like a lot, but remember that you have at least 8192 registers in each SM and potentially more on later hardware revisions. So with 128 threads, you have a total of 64 registers (8192 registers ÷ 128 threads) available to you, *if* you run just one block of threads on an SM.

However, running one block of 128 threads per SM is a very bad idea, even if you can use 64 registers per thread. As soon as you access memory, the SM would effectively idle. Only in the very limited case of heavy arithmetic intensity utilizing the 64 registers should you even consider this sort of approach. In practice, multiple blocks are run on each SM to avoid any idle states.

BLOCKS

Now 512 threads are not really going to get you very far on a GPU. This may sound like a huge number to many programmers from the CPU domain, but on a GPU you usually need thousands or tens of thousands of concurrent threads to really achieve the throughput available on the device.

We touched on this previously in the last section on threads, with the `num_blocks` parameter for the kernel invocation. This is the first parameter within the <<< and >>> symbols:

```
kernel_function<<<num_blocks, num_threads>>>(param1, param2,.....)
```

If you change this from one to two, you double the number of threads you are asking the GPU to invoke on the hardware. Thus, the same call,

```
some_kernel_func<<< 2, 128 >>>(a, b, c);
```

will call the GPU function named `some_kernel_func` 2 × 128 times, each with a different thread. This, however, complicates the calculation of the `thread_idx` parameter, effectively the array index position. This previous, simple kernel needs a slight amendment to account for this.

```
__global__ void some_kernel_func(int * const a, const int * const b, const int * const c)
{
  const unsigned int thread_idx = (blockIdx.x * blockDim.x) + threadIdx.x;

  a[thread_idx] = b[thread_idx] * c[thread_idx];
}
```

To calculate the `thread_idx` parameter, you must now take into account the number of blocks. For the first block, `blockIdx.x` will contain zero, so effectively the `thread_idx` parameter is equal to the `threadIdx.x` parameter you used earlier. However, for block two, `blockIdx.x` will hold the value 1. The parameter `blockDim.x` holds the value 128, which is, in effect, the number of threads you requested per block in this example. Thus, you have a 1 × 128 thread base addresses, before adding in the thread offset from the `threadIdx.x` parameter.

Have you noticed the small error we have introduced in adding in another block? You will now launch 256 threads in total and index the array from 0 to 255. If you don't also change the size of the array, from 128 elements to 256 elements, you will access and write beyond the end of the array. This array out-of-bounds error will not be caught by the compiler and the code may actually run, depending on what is located after the destination array, `a`. Be careful when invoking a kernel that you do not access out of bounds elements.

For this example, we will stick with the 128-byte array size and change the kernel to invoke two blocks of 64 threads each:

```
some_kernel_func<<< 2, 64 >>>(a, b, c);
```

Thus, you get what is shown in Figure 5.8.

Notice how, despite now having two blocks, the `thread_idx` parameter still equates to the array index, exactly as before. So what is the point of using blocks? In this trivial example, absolutely nothing. However, in any real-world problem, you have far more than 512 elements to deal with. In fact, if you look at the limit on the number of blocks, you find you have 65,536 blocks you can use.

At 65,536 blocks, with 512 threads per block, you can schedule 33,554,432 (around 33.5 million) threads in total. At 512 threads, you can have up to three blocks per SM. Actually, this limit is based on the total number of threads per SM, which is 1536 in the latest Fermi hardware, and as little as 768 in the original G80 hardware.

If you schedule the maximum of 1024 threads per block on the Fermi hardware, 65,536 blocks would translate into around 64 million threads. Unfortunately, at 1024 threads, you only get one thread block per SM. Consequently, you'd need some 65,536 SMs in a single GPU before you could not allocate at least one block per SM. Currently, the maximum number of SMs found on any card is 30. Thus, there is some provision for the number of SMs to grow before you have more SMs than the number of blocks the hardware can support. This is one of the beauties of CUDA—the fact it can scale to thousands of execution units. The limit of the parallelism is only really the limit of the amount of parallelism that can be found in the application.

With 64 million threads, assuming one thread per array element, you can process up to 64 million elements. Assuming each element is a single-precision floating-point number, requiring 4 bytes of data, you'd need around 256 million bytes, or 256 MB, of data storage space. Almost all GPU cards support at least this amount of memory space, so working with threads and blocks alone you can achieve quite a large amount of parallelism and data coverage.

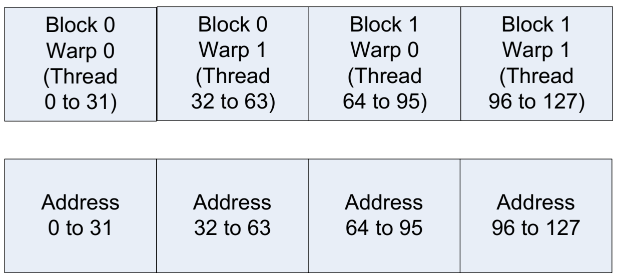

Block 0 Warp 0 (Thread 0 to 31)
Block 0 Warp 1 (Thread 32 to 63)
Block 1 Warp 0 (Thread 64 to 95)
Block 1 Warp 1 (Thread 96 to 127)
Address 0 to 31
Address 32 to 63
Address 64 to 95
Address 96 to 127

FIGURE 5.8

Block mapping to address.

For anyone worried about large datasets, where large problems can run into gigabytes, terabytes, or petabytes of data, there is a solution. For this, you generally either process more than one element per thread or use another dimension of blocks, which we'll cover in the next section.

Block arrangement

To ensure that we understand the block arrangement, we're going to write a short kernel program to print the block, thread, warp, and thread index to the screen. Now, unless you have at least version 3.2 of the SDK, the `printf` statement is not supported in kernels. So we'll ship the data back to the CPU and print it to the console window. The kernel program is thus as follows:

```
__global__ void what_is_my_id(unsigned int * const block,
                              unsigned int * const thread,
                              unsigned int * const warp,
                              unsigned int * const calc_thread)
{
  /* Thread id is block index * block size + thread offset into the block */
  const unsigned int thread_idx = (blockIdx.x * blockDim.x) + threadIdx.x;

  block[thread_idx] = blockIdx.x;
  thread[thread_idx] = threadIdx.x;

  /* Calculate warp using built in variable warpSize */
  warp[thread_idx] = threadIdx.x / warpSize;

  calc_thread[thread_idx] = thread_idx;
}
```

Now on the CPU you have to run a section of code, as follows, to allocate memory for the arrays on the GPU and then transfer the arrays back from the GPU and display them on the CPU.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

__global__ void what_is_my_id(unsigned int * const block,
            unsigned int * const thread,
            unsigned int * const warp,
            unsigned int * const calc_thread)
{
  /* Thread id is block index * block size + thread offset into the block */
  const unsigned int thread_idx = (blockIdx.x * blockDim.x) + threadIdx.x;

  block[thread_idx] = blockIdx.x;
  thread[thread_idx] = threadIdx.x;

  /* Calculate warp using built in variable warpSize */
  warp[thread_idx] = threadIdx.x / warpSize;

  calc_thread[thread_idx] = thread_idx;
}
```

```
#define ARRAY_SIZE 128
#define ARRAY_SIZE_IN_BYTES (sizeof(unsigned int) * (ARRAY_SIZE))

/* Declare statically four arrays of ARRAY_SIZE each */

unsigned int cpu_block[ARRAY_SIZE];
unsigned int cpu_thread[ARRAY_SIZE];
unsigned int cpu_warp[ARRAY_SIZE];
unsigned int cpu_calc_thread[ARRAY_SIZE];

int main(void)
{
  /* Total thread count = 2 * 64 = 128 */
  const unsigned int num_blocks = 2;
  const unsigned int num_threads = 64;
  char ch;

  /* Declare pointers for GPU based params */
  unsigned int * gpu_block;
  unsigned int * gpu_thread;
  unsigned int * gpu_warp;
  unsigned int * gpu_calc_thread;

  /* Declare loop counter for use later */
  unsigned int i;

  /* Allocate four arrays on the GPU */
  cudaMalloc((void **)&gpu_block, ARRAY_SIZE_IN_BYTES);
  cudaMalloc((void **)&gpu_thread, ARRAY_SIZE_IN_BYTES);
  cudaMalloc((void **)&gpu_warp, ARRAY_SIZE_IN_BYTES);
  cudaMalloc((void **)&gpu_calc_thread, ARRAY_SIZE_IN_BYTES);

  /* Execute our kernel */
  what_is_my_id<<<num_blocks, num_threads>>>(gpu_block, gpu_thread, gpu_warp,
                                            gpu_calc_thread);

 /* Copy back the gpu results to the CPU */
   cudaMemcpy(cpu_block, gpu_block, ARRAY_SIZE_IN_BYTES,
              cudaMemcpyDeviceToHost);
   cudaMemcpy(cpu_thread, gpu_thread, ARRAY_SIZE_IN_BYTES,
              cudaMemcpyDeviceToHost);
   cudaMemcpy(cpu_warp, gpu_warp, ARRAY_SIZE_IN_BYTES,
              cudaMemcpyDeviceToHost);
   cudaMemcpy(cpu_calc_thread, gpu_calc_thread, ARRAY_SIZE_IN_BYTES,
              cudaMemcpyDeviceToHost);

   /* Free the arrays on the GPU as now we're done with them */
  cudaFree(gpu_block);
  cudaFree(gpu_thread);
  cudaFree(gpu_warp);
  cudaFree(gpu_calc_thread);

  /* Iterate through the arrays and print */
  for (i=0; i < ARRAY_SIZE; i++)
  {
    printf("Calculated Thread: %3u - Block: %2u - Warp %2u - Thread %3u\n",
     cpu_calc_thread[i], cpu_block[i], cpu_warp[i], cpu_thread[i]);
  }
  ch = getch();
}
```

In this example, what you see is that each block is located immediately after the one before it. As you have only a single dimension to the array, laying out the thread blocks in a similar way is an easy way to conceptualize a problem. The output of the previous program is as follows:

```
Calculated Thread: 0 - Block: 0 - Warp 0 - Thread 0
Calculated Thread: 1 - Block: 0 - Warp 0 - Thread 1
Calculated Thread: 2 - Block: 0 - Warp 0 - Thread 2
Calculated Thread: 3 - Block: 0 - Warp 0 - Thread 3
Calculated Thread: 4 - Block: 0 - Warp 0 - Thread 4
...
Calculated Thread: 30 - Block: 0 - Warp 0 - Thread 30
Calculated Thread: 31 - Block: 0 - Warp 0 - Thread 31
Calculated Thread: 32 - Block: 0 - Warp 1 - Thread 32
Calculated Thread: 33 - Block: 0 - Warp 1 - Thread 33
Calculated Thread: 34 - Block: 0 - Warp 1 - Thread 34
...
Calculated Thread: 62 - Block: 0 - Warp 1 - Thread 62
Calculated Thread: 63 - Block: 0 - Warp 1 - Thread 63
Calculated Thread: 64 - Block: 1 - Warp 0 - Thread 0
Calculated Thread: 65 - Block: 1 - Warp 0 - Thread 1
Calculated Thread: 66 - Block: 1 - Warp 0 - Thread 2
Calculated Thread: 67 - Block: 1 - Warp 0 - Thread 3
...
Calculated Thread: 94 - Block: 1 - Warp 0 - Thread 30
Calculated Thread: 95 - Block: 1 - Warp 0 - Thread 31
Calculated Thread: 96 - Block: 1 - Warp 1 - Thread 32
Calculated Thread: 97 - Block: 1 - Warp 1 - Thread 33
Calculated Thread: 98 - Block: 1 - Warp 1 - Thread 34
Calculated Thread: 99 - Block: 1 - Warp 1 - Thread 35
Calculated Thread: 100 - Block: 1 - Warp 1 - Thread 36
...
Calculated Thread: 126 - Block: 1 - Warp 1 - Thread 62
Calculated Thread: 127 - Block: 1 - Warp 1 - Thread 63
```

As you can see, the calculated thread, or the thread ID, goes from 0 to 127. Within that you allocate two blocks of 64 threads each. The thread indexes within each of these blocks go from 0 to 63. You also see that each block generates two warps.

GRIDS

A grid is simply a set of blocks where you have an *X* and a *Y* axis, in effect a 2D mapping. The final *Y* mapping gives you $Y \times X \times T$ possibilities for a thread index. Let's look at this using an example, but limiting the *Y* axis to a single row to start off with.

If you were to look at a typical HD image, you have a 1920 × 1080 resolution. The number of threads in a block should *always* be a multiple of the warp size, which is currently defined as 32. As you can only schedule a full warp on the hardware, if you don't do this, then the remaining part of the warp goes unused and you have to introduce a condition to ensure you don't process elements off the end of the *X* axis. This, as you'll see later, slows everything down.

To avoid poor memory coalescing, you should always try to arrange the memory and thread usage so they map. This will be covered in more detail in the next chapter on memory. Failure to do so will result in something in the order of a five times drop in performance.

To avoid tiny blocks, as they don't make full use of the hardware, we'll pick 192 threads per block. In most cases, this is the *minimum* number of threads you should think about using. This gives you exactly 10 blocks across each row of the image, which is an easy number to work with (Figure 5.9). Using a thread size that is a multiple of the *X* axis and the warp size makes life a lot easier.

Along the top on the *X* axis, you have the thread index. The row index forms the *Y* axis. The height of the row is exactly one pixel. As you have 1080 rows of 10 blocks, you have in total 1080 × 10 = 10,800 blocks. As each block has 192 threads, you are scheduling just over two million threads, one for each pixel.

This particular layout is useful where you have one operation on a single pixel or data point, *or* where you have some operation on a number of data points in the *same* row. On the Fermi hardware, at eight blocks per SM, you'd need a total of 1350 SMs (10,800 total blocks ÷ 8 scheduled blocks) to run out of parallelism at the application level. On the Fermi hardware currently available, you have only 16 SMs (GTX580), so each SM would be given 675 blocks to process.

This is all very well, but what if your data is not row based? As with arrays, you are not limited to a single dimension. You can have a 2D thread block arrangement. A lot of image algorithms, for

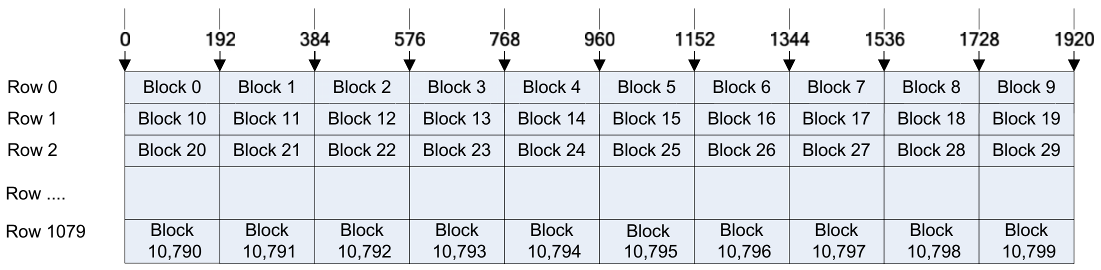


FIGURE 5.9

Block allocation to rows.

example, use 8 × 8 blocks of pixels. We're using pixels here to show this arrangement, as it's easy for most people to conceptualize. Your data need not be pixel based. You typically represent pixels as a red, green, and blue component. You could equally have *x*, *y*, and *z* spatial coordinates as a single data point, or a simple 2D or 3D matrix holding the data points.

Stride and offset

As with arrays in C, thread blocks can be thought of as 2D structures. However, for 2D thread blocks, we need to introduce some new concepts. Just like in array indexing, to index into a *Y* element of 2D array, you need to know the width of the array, the number of *X* elements. Consider the array in Figure 5.10.

The width of the array is referred to as the stride of the memory access. The offset is the column value being accessed, starting at the left, which is always element 0. Thus, you have array element 5 being accessed with the index [1][5] or via the address calculation (row × (sizeof(array_element) × width))) + ((sizeof(array_element) × offset)). This is the calculation the compiler effectively uses, in an optimized form, when you do multidimensional array indexing in C code.

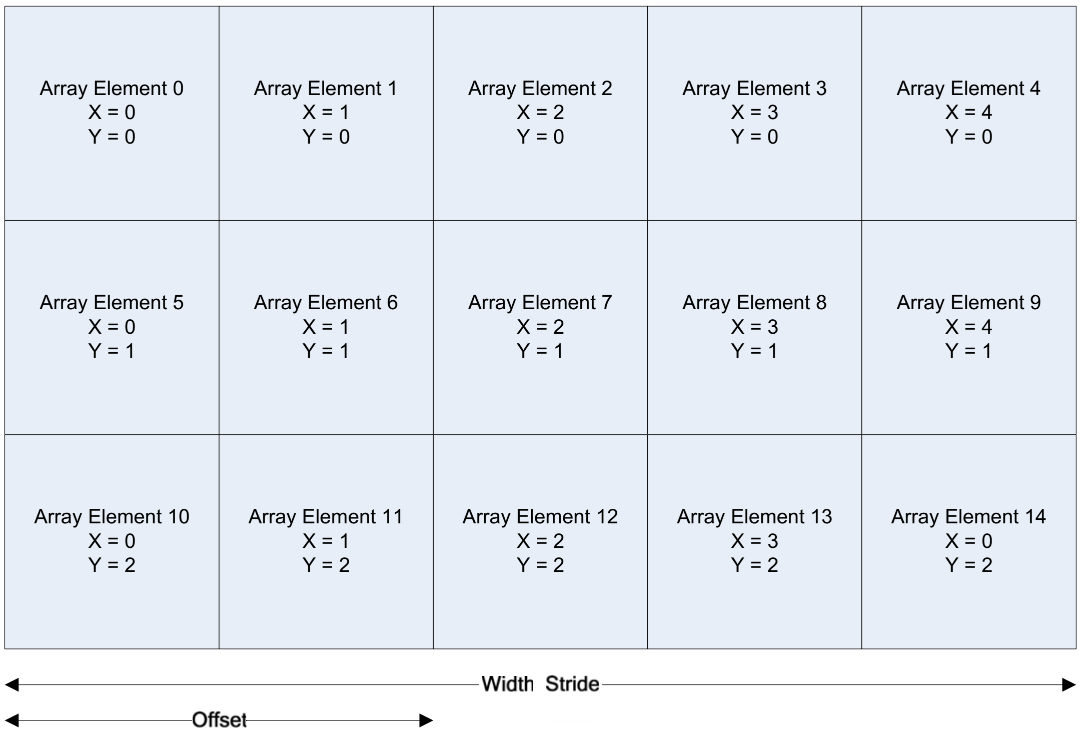

Array Element 0 X = 0 Y = 0
Array Element 1 X = 1 Y = 0
Array Element 2 X = 2 Y = 0
Array Element 3 X = 3 Y = 0
Array Element 4 X = 4 Y = 0
Array Element 5 X = 0 Y = 1
Array Element 6 X = 1 Y = 1
Array Element 7 X = 2 Y = 1
Array Element 8 X = 3 Y = 1
Array Element 9 X = 4 Y = 1
Array Element 10 X = 0 Y = 2
Array Element 11 X = 1 Y = 2
Array Element 12 X = 2 Y = 2
Array Element 13 X = 3 Y = 2
Array Element 14 X = 0 Y = 2
Width Stride
Offset

FIGURE 5.10

Array mapping to elements.

Now, how is this relevant to threads and blocks in CUDA? CUDA is designed to allow for data decomposition into parallel threads and blocks. It allows you to define 1D, 2D, or 3D indexes ($Y \times X \times T$) when referring to the parallel structure of the program. This maps directly onto the way a typical area of memory is set out, allowing the data you are processing to be allocated to individual SMs. The process of keeping data close to the processor hugely increases performance, both on the GPU and CPU.

However, there is one caveat you must be aware of when laying out such arrays. The width value of the array must always be a multiple of the warp size. If it is not, pad the array to the next largest multiple of the warp size. Padding to the next multiple of the warp size should introduce only a very modest increase in the size of the dataset. Be aware, however, you'll need to deal with the padded boundary, or halo cells, differently than the rest of the cells. You can do this using divergence in the execution flow (e.g., using an `if` statement) or you can simply calculate the padded cells and discard the result. We'll cover divergence and the problems it causes later in the book.

X and *Y* thread indexes

Having a 2D array in terms of blocks means you get two thread indexes, as you will be accessing the data in a 2D way:

```
const unsigned int idx = (blockIdx.x * blockDim.x) + threadIdx.x;
const unsigned int idy = (blockIdx.y * blockDim.y) + threadIdx.y;

some_array[idy][idx] += 1.0;
```

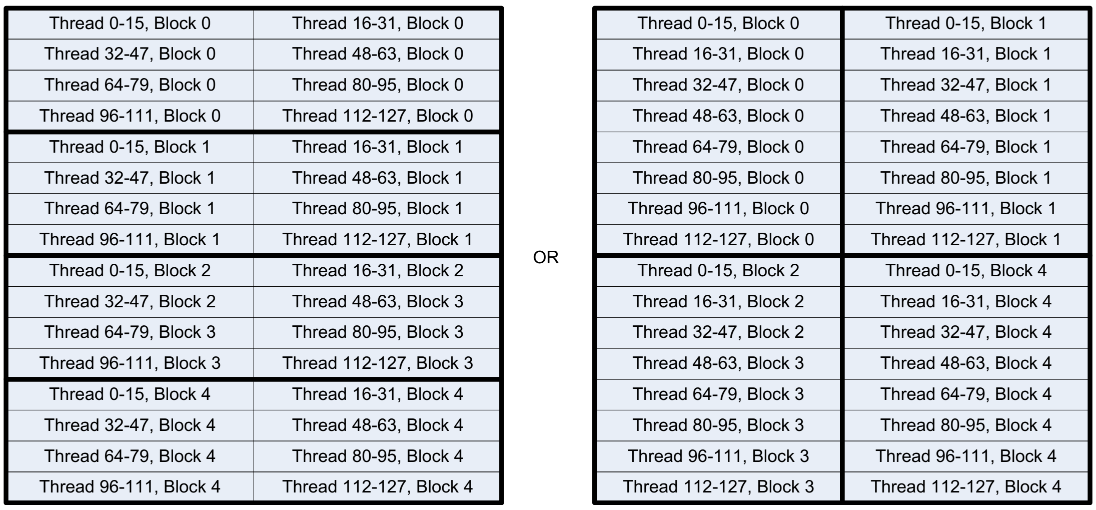
Notice the use of `blockDim.x` and `blockDim.y`, which the CUDA runtime completes for you, specifying the dimension on the *X* and *Y* axis. So let's modify the existing program to work on a 32×16 array. As you want to schedule four blocks, you can schedule them as stripes across the array, or as squares within the array, as shown in Figure 5.11.

You could also rotate the striped version 90 degrees and have a column per thread block. Never do this, as it will result in completely noncoalesced memory accesses that will drop the performance of your application by an order of magnitude or more. Be careful when parallelizing loops so that the access pattern always runs sequentially through memory in rows and never columns. This applies equally to CPU and GPU code.

Now why might you choose the square layout over the rectangle layout? Well, two reasons. The first is that threads within *the same block* can communicate using shared memory, a very quick way to cooperate with one another. The second consideration is you get marginally quicker memory access with single 128-byte transaction instead of two, 64-byte transactions, due to accessing within a warp being coalesced and 128 bytes being the size of a cache line in the Fermi hardware. In the square layout notice you have threads 0 to 15 mapped to one block and the next memory location belongs to another block. As a consequence you get two transactions instead of one, as with the rectangular layout. However, if the array was slightly larger, say 64×16, then you would not see this issue, as you'd have 32 threads accessing contiguous memory, and thus a single 128-byte fetch from memory issued.

Use the following to modify the program to use either of the two layouts:

```
dim3 threads_rect(32,4);
dim3 blocks_rect(1,4);
```



Thread 0-15, Block 0	Thread 16-31, Block 0
Thread 32-47, Block 0	Thread 48-63, Block 0
Thread 64-79, Block 0	Thread 80-95, Block 0
Thread 96-111, Block 0	Thread 112-127, Block 0
Thread 0-15, Block 1	Thread 16-31, Block 1
Thread 32-47, Block 1	Thread 48-63, Block 1
Thread 64-79, Block 1	Thread 80-95, Block 1
Thread 96-111, Block 1	Thread 112-127, Block 1
Thread 0-15, Block 2	Thread 16-31, Block 2
Thread 32-47, Block 2	Thread 48-63, Block 3
Thread 64-79, Block 3	Thread 80-95, Block 3
Thread 96-111, Block 3	Thread 112-127, Block 3
Thread 0-15, Block 4	Thread 16-31, Block 4
Thread 32-47, Block 4	Thread 48-63, Block 4
Thread 64-79, Block 4	Thread 80-95, Block 4
Thread 96-111, Block 4	Thread 112-127, Block 4

OR

Thread 0-15, Block 0	Thread 0-15, Block 1
Thread 16-31, Block 0	Thread 16-31, Block 1
Thread 32-47, Block 0	Thread 32-47, Block 1
Thread 48-63, Block 0	Thread 48-63, Block 1
Thread 64-79, Block 0	Thread 64-79, Block 1
Thread 80-95, Block 0	Thread 80-95, Block 1
Thread 96-111, Block 0	Thread 96-111, Block 1
Thread 112-127, Block 0	Thread 112-127, Block 1
Thread 0-15, Block 2	Thread 0-15, Block 4
Thread 16-31, Block 2	Thread 16-31, Block 4
Thread 32-47, Block 2	Thread 32-47, Block 4
Thread 48-63, Block 3	Thread 48-63, Block 4
Thread 64-79, Block 3	Thread 64-79, Block 4
Thread 80-95, Block 3	Thread 80-95, Block 4
Thread 96-111, Block 3	Thread 96-111, Block 4
Thread 112-127, Block 3	Thread 112-127, Block 4

FIGURE 5.11

Alternative thread block layouts.

```
or
dim3 threads_square(16,8);
dim3 blocks_square(2,2);
```

In either arrangement you have the same total number of threads ($32 \times 4 = 128$, $16 \times 8 = 128$). It's simply the layout of the threads that is different.

The `dim3` type is simply a special CUDA type that you have to use to create a 2D layout of threads. In the rectangle example, you're saying you want 32 threads along the *X* axis by 4 threads along the *Y* axis, within a single block. You're then saying you want the blocks to be laid out as one block wide by four blocks high.

You'll need to invoke the kernel with

```
some_kernel_func<<< blocks_rect, threads_rect >>>(a, b, c);
```

or

```
some_kernel_func<<< blocks_square, threads_square >>>(a, b, c);
```

As you no longer want just a single thread ID, but an *X* and *Y* position, you'll need to update the kernel to reflect this. However, you also need to linearize the thread ID because there are situations where you may want an absolute thread index. For this we need to introduce a couple of new concepts, shown in Figure 5.12.

You can see a number of new parameters, which are:

```
gridDim.x-The size in blocks of the X dimension of the grid.
gridDim.y-The size in blocks of the Y dimension of the grid.

blockDim.x-The size in threads of the X dimension of a single block.
blockDim.y-The size in threads of the Y dimension of a single block.
```

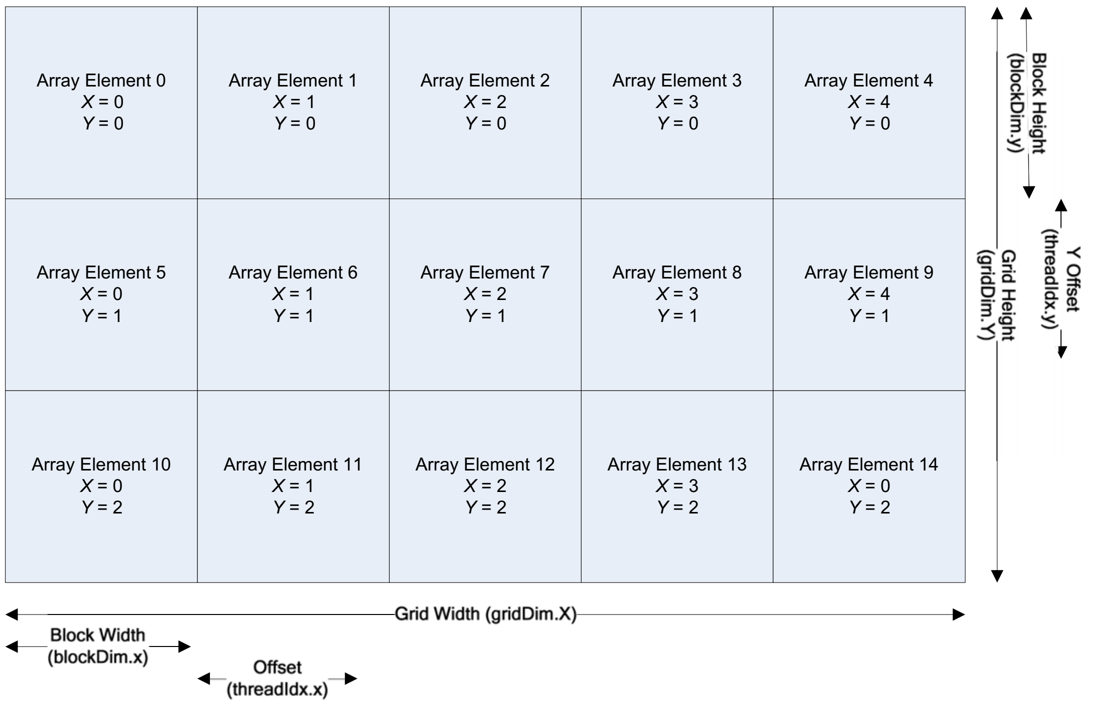


FIGURE 5.12

Grid, block, and thread dimensions.

```
theadIdx.x-The offset within a block of the X thread index.
theadIdx.y-The offset within a block of the Y thread index.
```

You can work out the absolute thread index by working out the *Y* position and multiplying this by number of threads in a row. You then simply add in the *X* offset from the start of the row. Thus, the thread index calculation is

```
thread_idx = ((gridDim.x * blockDim.x) * idy) + idx;
```

So you need to modify the kernel to additionally return the *X* and *Y* positions plus some other useful bits of information, as follows:

```
__global__ void what_is_my_id_2d_A(
unsigned int * const block_x,
unsigned int * const block_y,
unsigned int * const thread,
unsigned int * const calc_thread,
unsigned int * const x_thread,
unsigned int * const y_thread,
unsigned int * const grid_dimx,
unsigned int * const block_dimx,
unsigned int * const grid_dimy,
unsigned int * const block_dimy)
{
  const unsigned int idx        = (blockIdx.x * blockDim.x) + threadIdx.x;
  const unsigned int idy        = (blockIdx.y * blockDim.y) + threadIdx.y;
  const unsigned int thread_idx = ((gridDim.x * blockDim.x) * idy) + idx;

  block_x[thread_idx]     = blockIdx.x;
  block_y[thread_idx]     = blockIdx.y;
  thread[thread_idx]      = threadIdx.x;
  calc_thread[thread_idx] = thread_idx;
  x_thread[thread_idx]    = idx;
  y_thread[thread_idx]    = idy;
  grid_dimx[thread_idx]   = gridDim.x;
  block_dimx[thread_idx]  = blockDim.x;
  grid_dimy[thread_idx]   = gridDim.y;
  block_dimy[thread_idx]  = blockDim.y;
}
```

We'll call the kernel twice to demonstrate how you can arrange array blocks and threads.

As you're now passing an additional dataset to compute, you need an additional `cudaMalloc`, `cudaFree`, and `cudaMemcpy` to copy the data from the device. As you're using two dimensions, you'll also need to modify the array size to allocate and transfer the correct size of data.

```
#define ARRAY_SIZE_X 32
#define ARRAY_SIZE_Y 16

#define ARRAY_SIZE_IN_BYTES ((ARRAY_SIZE_X) * (ARRAY_SIZE_Y) * (sizeof(unsigned int)))

/* Declare statically six arrays of ARRAY_SIZE each */
unsigned int cpu_block_x[ARRAY_SIZE_Y][ARRAY_SIZE_X];
unsigned int cpu_block_y[ARRAY_SIZE_Y][ARRAY_SIZE_X];
unsigned int cpu_thread[ARRAY_SIZE_Y][ARRAY_SIZE_X];
unsigned int cpu_warp[ARRAY_SIZE_Y][ARRAY_SIZE_X];
unsigned int cpu_calc_thread[ARRAY_SIZE_Y][ARRAY_SIZE_X];
unsigned int cpu_xthread[ARRAY_SIZE_Y][ARRAY_SIZE_X];
unsigned int cpu_ythread[ARRAY_SIZE_Y][ARRAY_SIZE_X];
unsigned int cpu_grid_dimx[ARRAY_SIZE_Y][ARRAY_SIZE_X];
unsigned int cpu_block_dimx[ARRAY_SIZE_Y][ARRAY_SIZE_X];
unsigned int cpu_grid_dimy[ARRAY_SIZE_Y][ARRAY_SIZE_X];
unsigned int cpu_block_dimy[ARRAY_SIZE_Y][ARRAY_SIZE_X];

int main(void)
{
  /* Total thread count = 32 * 4 = 128 */
  const dim3 threads_rect(32, 4); /* 32 * 4 */
  const dim3 blocks_rect(1,4);
  /* Total thread count = 16 * 8 = 128 */
  const dim3 threads_square(16, 8); /* 16 * 8 */
  const dim3 blocks_square(2,2);

  /* Needed to wait for a character at exit */
  char ch;

  /* Declare pointers for GPU based params */
  unsigned int * gpu_block_x;
  unsigned int * gpu_block_y;
  unsigned int * gpu_thread;
  unsigned int * gpu_warp;
  unsigned int * gpu_calc_thread;
  unsigned int * gpu_xthread;
  unsigned int * gpu_ythread;
  unsigned int * gpu_grid_dimx;
  unsigned int * gpu_block_dimx;
  unsigned int * gpu_grid_dimy;
  unsigned int * gpu_block_dimy;

  /* Allocate four arrays on the GPU */
  cudaMalloc((void **)&gpu_block_x, ARRAY_SIZE_IN_BYTES);
  cudaMalloc((void **)&gpu_block_y, ARRAY_SIZE_IN_BYTES);
  cudaMalloc((void **)&gpu_thread, ARRAY_SIZE_IN_BYTES);
  cudaMalloc((void **)&gpu_calc_thread, ARRAY_SIZE_IN_BYTES);
  cudaMalloc((void **)&gpu_xthread, ARRAY_SIZE_IN_BYTES);
  cudaMalloc((void **)&gpu_ythread, ARRAY_SIZE_IN_BYTES);
  cudaMalloc((void **)&gpu_grid_dimx, ARRAY_SIZE_IN_BYTES);
  cudaMalloc((void **)&gpu_block_dimx, ARRAY_SIZE_IN_BYTES);
  cudaMalloc((void **)&gpu_grid_dimy, ARRAY_SIZE_IN_BYTES);
  cudaMalloc((void **)&gpu_block_dimy, ARRAY_SIZE_IN_BYTES);

  for (int kernel=0; kernel < 2; kernel++)
  {
    switch (kernel)
    {
      case 0:
      {
        /* Execute our kernel */
        what_is_my_id_2d_A<<<blocks_rect, threads_rect>>>(gpu_block_x, gpu_block_y,
gpu_thread, gpu_calc_thread, gpu_xthread, gpu_ythread, gpu_grid_dimx, gpu_block_dimx,
gpu_grid_dimy, gpu_block_dimy);
      } break;

      case 1:
      {
       /* Execute our kernel */
       what_is_my_id_2d_A<<<blocks_square, threads_square>>>(gpu_block_x, gpu_block_y,
gpu_thread, gpu_calc_thread, gpu_xthread, gpu_ythread, gpu_grid_dimx, gpu_block_dimx,
gpu_grid_dimy, gpu_block_dimy);
      } break;

      default: exit(1); break;
    }

    /* Copy back the gpu results to the CPU */
    cudaMemcpy(cpu_block_x, gpu_block_x, ARRAY_SIZE_IN_BYTES,
               cudaMemcpyDeviceToHost);
    cudaMemcpy(cpu_block_y, gpu_block_y, ARRAY_SIZE_IN_BYTES,
               cudaMemcpyDeviceToHost);
    cudaMemcpy(cpu_thread, gpu_thread, ARRAY_SIZE_IN_BYTES,
               cudaMemcpyDeviceToHost);
    cudaMemcpy(cpu_calc_thread, gpu_calc_thread, ARRAY_SIZE_IN_BYTES,
               cudaMemcpyDeviceToHost);
    cudaMemcpy(cpu_xthread, gpu_xthread, ARRAY_SIZE_IN_BYTES,
               cudaMemcpyDeviceToHost);
    cudaMemcpy(cpu_ythread, gpu_ythread, ARRAY_SIZE_IN_BYTES,
               cudaMemcpyDeviceToHost);
    cudaMemcpy(cpu_grid_dimx, gpu_grid_dimx, ARRAY_SIZE_IN_BYTES,
               cudaMemcpyDeviceToHost);
    cudaMemcpy(cpu_block_dimx,gpu_block_dimx, ARRAY_SIZE_IN_BYTES,
               cudaMemcpyDeviceToHost);
    cudaMemcpy(cpu_grid_dimy, gpu_grid_dimy, ARRAY_SIZE_IN_BYTES,
               cudaMemcpyDeviceToHost);
    cudaMemcpy(cpu_block_dimy, gpu_block_dimy, ARRAY_SIZE_IN_BYTES,
               cudaMemcpyDeviceToHost);

    printf("\nKernel %d\n", kernel);
    /* Iterate through the arrays and print */
    for (int y=0; y < ARRAY_SIZE_Y; y++)
    {
        for (int x=0; x < ARRAY_SIZE_X; x++)
        {
          printf("CT: %2u BKX: %1u BKY: %1u TID: %2u YTID: %2u XTID: %2u GDX: %1u BDX: %
1u GDY %1u BDY %1u\n", cpu_calc_thread[y][x], cpu_block_x[y][x], cpu_block_y[y][x],
cpu_thread[y][x], cpu_ythread[y][x], cpu_xthread[y][x], cpu_grid_dimx[y][x],
cpu_block_dimx[y][x], cpu_grid_dimy[y][x], cpu_block_dimy[y][x]);

          /* Wait for any key so we can see the console window */
          ch = getch();
          }
    }
    /* Wait for any key so we can see the console window */
    printf("Press any key to continue\n");
    ch = getch();
  }

  /* Free the arrays on the GPU as now we're done with them */
  cudaFree(gpu_block_x);
  cudaFree(gpu_block_y);
  cudaFree(gpu_thread);
  cudaFree(gpu_calc_thread);
  cudaFree(gpu_xthread);
  cudaFree(gpu_ythread);
  cudaFree(gpu_grid_dimx);
  cudaFree(gpu_block_dimx);
  cudaFree(gpu_grid_dimy);
  cudaFree(gpu_block_dimy);
}
```

The output is too large to list here. If you run the program in the downloadable source code section, you'll see you iterate through the threads and blocks as illustrated in Figure 5.12.

WARPS

We touched a little on warp scheduling when talking about threads. Warps are the basic unit of execution on the GPU. The GPU is effectively a collection of SIMD vector processors. Each group of threads, or warps, is executed together. This means, in the ideal case, only one fetch from memory for the current instruction and a broadcast of that instruction to the entire set of SPs in the warp. This is much more efficient than the CPU model, which fetches independent execution streams to support task-level parallelism. In the CPU model, for every core you have running an independent task, you can conceptually divide the memory bandwidth, and thus the effective instruction throughput, by the number of cores. In practice, on CPUs, the multilevel, on-chip caches hide a lot of this providing the program fits within the cache.

You find vector-type instructions on conventional CPUs, in the form of SSE, MMX, and AVX instructions. These execute the same single instruction on multiple data operands. Thus, you can say, for *N* values, increment all values by one. With SSE, you get 128-bit registers, so you can operate on four parameters at any given time. AVX extends this to 256 bits. This is quite powerful, but until recently, unless you were using the Intel compiler, there was little native support for this type of optimization. AVX is now supported by the current GNU gcc compiler. Microsoft Visual Studio 2010 supports it through the use of a "/arch:AVX" compiler switch. Given this lack of support until relatively recently, vector-type instructions are not as widely used as they could be, although this is likely to change significantly now that support is no longer restricted to the Intel compiler.

With GPU programming, you have no choice: It's vector architecture and expects you to write code that runs on thousands of threads. You can actually write a single-thread GPU program with a simple `if` statement checking if the thread ID is zero, but this will get you terrible performance compared with the CPU. It can, however, be useful just to get an initial serial CPU implementation working. This