NITPulse

NAVODAYA INSTITUTE OF TECHNOLOGY, RAICHUR

DEPARMENT OF COMPUTER SCIENCE & ENGINEERING

MODULE 5

GPU programming with CUDA

6.1 GPUs and GPGPU

In the late 1990s and early 2000s, the computer industry responded to the demand for highly realistic computer video games and video animations by developing extremely powerful **graphics processing units** or **GPUs**. These processors, as their name suggests, are designed to improve the performance of programs that need to render many detailed images.

The existence of this computational power was a temptation to programmers who didn't specialize in computer graphics, and by the early 2000s they were trying to apply the power of GPUs to solving general computational problems, problems such as searching and sorting, rather than graphics. This became known as **General Purpose computing on GPUs** or **GPGPU**.

One of the biggest difficulties faced by the early developers of GPGPU was that the GPUs of the time could only be programmed using computer graphics APIs, such as Direct3D and OpenGL. So programmers needed to reformulate algorithms for general computational problems so that they used graphics concepts, such as vertices, triangles, and pixels. This added considerable complexity to the development of early GPGPU programs, and it wasn't long before several groups started work on developing languages and compilers that allowed programmers to implement general algorithms for GPUs in APIs that more closely resembled conventional, high-level languages for CPUs.

These efforts led to the development of several APIs for general purpose programming on GPUs. Currently the most widely used APIs are CUDA and OpenCL. CUDA was developed for use on Nvidia GPUs. OpenCL, on the other hand, was designed to be highly portable. It was designed for use on arbitrary GPUs *and* other processors—processors such as field programmable gate arrays (FPGAs) and digital signal processors (DSPs). To

ensure this portability, an OpenCL program must include a good deal of code providing information about which systems it can be run on and information about how it should be run. Since CUDA was developed to run only on Nvidia GPUs, it requires relatively modest setup, and, as a consequence, we'll use it instead of OpenCL.

Copyright © 2022 Elsevier Inc. All rights reserved.**An Introduction to Parallel Programming.** https://doi.org/10.1016/B978-0-12-804605-0.00013-0

Table 6.1 Execution of branch on a SIMD system.

Time	Datapaths with `x[i] >= 0`	Datapaths with `x[i] < 0`
1	Test `x[i] >= 0`	Test `x[i] >= 0`
2	`x[i] += 1`	Idle
3	Idle	`x[i] -= 2`

6.2 GPU architectures

As we've seen (see Chapter 2), CPU architectures can be extremely complex. However, we often think of a conventional CPU as a SISD device in Flynn's Taxonomy (see Section 2.3): the processor fetches an instruction from memory and executes the instruction on a small number of data items. The instruction is an element of the *Single Instruction stream*—the "SI" in SISD—and the data items are elements of the *Single Data stream*—the "SD" in SISD. GPUs, however, are composed of SIMD or *Single Instruction stream, Multiple Data stream* processors. So, to understand how to program them, we need to first look at their architecture.

Recall (from Section 2.3) that we can think of a SIMD processor as being composed of a single control unit and multiple datapaths. The control unit fetches an instruction from memory and broadcasts it to the datapaths. Each datapath either executes the instruction on *its* data or is idle.

For example, suppose there are n datapaths that share an n-element array `x`. Also suppose that the ith datapath will work with `x[i]`. Now suppose we want to add 1 to the nonnegative elements of `x` and subtract 2 from the negative elements of `x`. We might implement this with the following code:

```
/* Datapath i executes the following code */ if ( x [ i ] >= 0)
   x [ i ]  +=  1; else
   x [ i ] -= 2;
```

In a typical SIMD system, each datapath carries out the test `x[i] >= 0`. Then the datapaths for which the test is true execute `x[i] += 1`, while those for which `x[i] < 0` are *idle*. Then the roles of the datapaths are reversed: those for which `x[i] >= 0` are idle while the other datapaths execute `x[i] -= 2`. See Table 6.1.

A typical GPU can be thought of as being composed of one or more SIMD processors. Nvidia GPUs are composed of **Streaming Multiprocessors** or **SMs**.[1] One SM can have several control units and many more datapaths. So an SM can be thought of as consisting of one or more SIMD

[1] The abbreviation that Nvidia uses for a streaming multiprocessor depends on the particular GPU microarchitecture. For example, Tesla and Fermi multiprocessors have SMs, Kepler multiprocessors have SMXs, and Maxwell multiprocessors have SMMs. More recent GPUs have SMs. We'll use SM, regardless of the microarchitecture.

processors. The SMs, however, operate asynchronously: there is no penalty if one branch of an **if–else** executes on one SM, and

Table 6.2 Execution of branch on a system with multiple SMs.

Time	Datapaths with `x[i] >= 0` (on SM A)	Datapaths with `x[i] < 0` (on SM B)
1	Test `x[i] >= 0`	Test `x[i] >= 0`
2	`x[i] += 1`	`x[i] -= 2`

the other executes on another SM. So in our preceding example, if all the threads with `x[i] >= 0` were executing on one SM, and all the threads with `x[i] < 0` were executing on another, the execution of our **if–else** example would require only two stages. (See Table 6.2.)

In Nvidia parlance, the datapaths are called cores, **Streaming Processors**, or **SPs**. Currently,[2] one of the most powerful Nvidia processor has 82 SMs, and each SM has 128 SPs for a total of 10,496 SPs. Since we use the term "core" to mean something else when we're discussing MIMD architectures, we'll use SP to denote an Nvidia datapath. Also note that Nvidia uses the term **SIMT** instead of SIMD. SIMT stands for Single Instruction Multiple Thread, and the term is used because threads on an SM that are executing the same instruction may not execute simultaneously: to hide memory access latency, some threads may block while memory is accessed and other threads, that have already accessed the data, may proceed with execution.

Each SM has a relatively small block of memory that is shared among its SPs. As we'll see, this memory can be accessed very quickly by the SPs. All of the SMs on a single chip also have access to a much larger block of memory that is shared among
all the SPs. Accessing this memory is relatively slow. (See Fig. 6.1.)

The GPU and its associated memory are usually physically separate from the CPU and its associated memory. In Nvidia documentation, the CPU together with its associated memory is often called the **host**, and the GPU together with its memory is called the **device**. In earlier systems the physical separation of host and device memories required that data was usually explicitly transferred between CPU memory and GPU memory. That is, a function was called that would transfer a block of data from host memory to device memory or vice versa. So, for example, data read from a file by the CPU or output data generated by the GPU would have to be transferred between the host and device with an explicit function call. However, in more recent Nvidia systems (those with compute capability ≥ 3.0), the explicit transfers in the source code aren't needed for correctness, although they may be able to improve overall performance. (See Fig. 6.2.)

[2] Spring 2021.

6.3 Heterogeneous computing

Up to now we've implicitly assumed that our parallel programs will be run on systems in which the individual processors have identical architectures. Writing a program

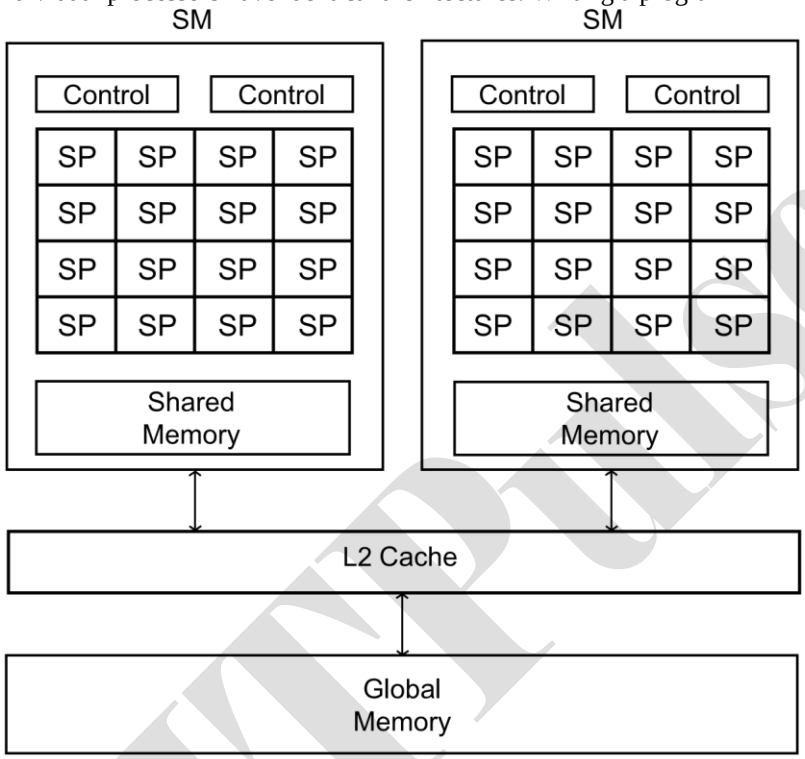

FIGURE 6.1

Simplified block diagram of a GPU.

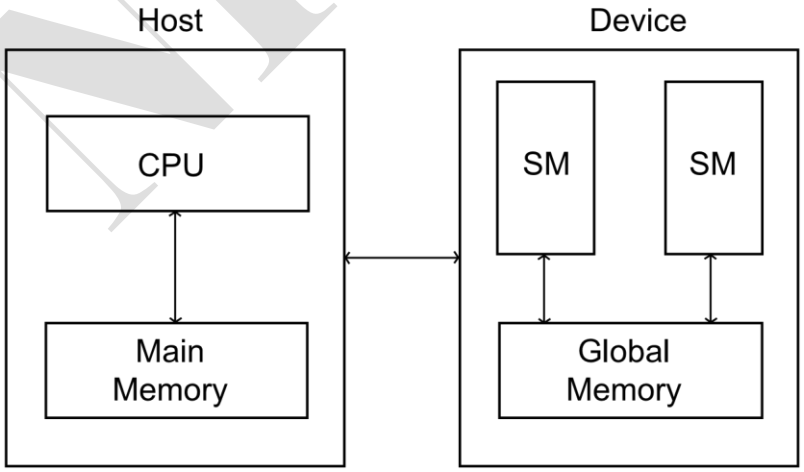

FIGURE 6.2

NIPulse

Simplified block diagram of a CPU and a GPU.

that runs on a GPU is an example of **heterogeneous** computing. The reason is that the programs make use of both a host processor—a conventional CPU—and a device processor—a GPU—and, as we've just seen, the two processors have different architectures.

We'll still write a single program (using the SPMD approach—see Section 2.4.1), but now there will be functions that we write for conventional CPUs and functions that we write for GPUs. So, effectively, we'll be writing two programs.

Heterogeneous computing has become much more important in recent years. Recall from Chapter 1 that from about 1986 to 2003, the single-thread performance of conventional CPUs was increasing, on average, more than 50% per year, but since 2003, the improvement in single-thread performance has decreased to the point that from 2015 to 2017, it has been growing at less than 4% per year [28]. So programmers are leaving no stone unturned in their search for ways to bolster performance, and one possibility is to make use of other types of processors, processors other than CPUs. Our focus is GPUs, but other possibilities include **Field Programmable Gate Arrays** or **FPGAs**, and **Digital Signal Processors** or **DSPs**. FPGAs contain programmable logic blocks and interconnects that can be configured prior to program execution. DSPs contain special circuitry for manipulating (e.g., compressing, filtering) signals, especially "real-world" analog signals.

6.4 CUDA hello

So let's start talking about the CUDA API, the API we'll be using to program heterogeneous CPU–GPU systems.

CUDA is a software platform that can be used to write GPGPU programs for heterogeneous systems equipped with an Nvidia GPU. CUDA was originally an acronym for "Compute Unified Device Architecture," which was meant to suggest that it provided a single interface for programming both CPU and GPU. More recently, however, Nvidia has decided that CUDA is not an acronym; it's simply the name of an API for GPGPU programming.

There is a language-specific CUDA API for several languages; for example, there are CUDA APIs for C, C++, Fortran, Python, and Java. We'll be using CUDA C, but we need to be aware that sometimes we'll need to use some C++ constructs. This is because the CUDA C compiler can compile both C and C++ programs, and it can do this because it is a modified C++ compiler. So where the specifications for C and C++ differ the CUDA compiler sometimes uses C++. For example, since the C library function `malloc` returns a **void*** pointer, a C program doesn't need a cast in the instruction

```
float *x = malloc ( n* sizeof ( float ) ) ;
```

However, in C++ a cast is required `float *x = (float *) malloc (n* sizeof (float ));`

As usual, we'll begin by implementing a version of the "hello, world" program. We'll write a CUDA C program in which each CUDA thread prints a greeting.[3] Since the program is heterogeneous, we will, effectively, be writing two programs: a host or CPU program and a device or GPU program.

Note that even though our programs are written in CUDA C, CUDA programs cannot be compiled with an ordinary C compiler. So unlike MPI and Pthreads, CUDA is not just a library that can be linked in to an ordinary C program: CUDA requires a special compiler. For example, an ordinary C compiler (such as `gcc`) generates a machine language executable for a single CPU (e.g., an x86 processor), but the CUDA compiler must generate machine language for two different processors: the host processor and the device processor.

6.4.1 The source code

The source code for a CUDA program that prints a greeting from each thread on the GPU is shown in Program 6.1.

As you might guess, there's a header file for CUDA programs, which we include in Line 2.

The `Hello` function follows the include directives and starts on Line 5. This function is run by each thread on the GPU. In CUDA parlance, it's called a **kernel**, a function that is started by the host but runs on the device. CUDA kernels are identified by the keyword `__global__`, and they always have return type **void**.

The `main` function follows the kernel on Line 12. Like ordinary C programs, CUDA C programs start execution in `main`, and the `main` function runs on the *host*. The function first gets the number of threads from the command line. It then starts the required number of copies of the kernel on Line 18. The call to `cudaDeviceSynchronize` will cause the main program to wait until all the threads have finished executing the kernel, and when this happens, the program terminates as usual with **return** 0.

6.4.2 Compiling and running the program

A CUDA program file that contains both host code and device code should be stored in a file with a ".cu" suffix. For example, our hello program is in a file called `cuda_hello.cu`. We can compile it using the CUDA compiler `nvcc`. The command should look something like this[4]:

```
$ nvcc -o cuda_hello cuda_hello.cu
```

If we want to run one thread on the GPU, we can type

[3] This program requires an Nvidia GPU with compute capability ≥ 2.0.

[4] Recall that the dollar sign (`$`) denotes the shell prompt, and it should not be typed.

```
$ ./cuda_hello 1
```

and the output will be

```
#include <stdio.h>
#include <cuda.h>                /* Header file for CUDA */

/* Device code: runs on GPU */
__global__ void Hello(void) {

    printf("Hello from thread %d!\n", threadIdx.x);
}   /* Hello */


/* Host code: Runs on CPU */ int main(int argc, char* argv[]) { int thread_count;
/* Number of threads to run on GPU */

    thread_count = strtol(argv[1], NULL, 10);
                             /* Get thread_count from command line */

    Hello<<<1, thread_count>>>();
                             /* Start thread_count threads on GPU, */

    cudaDeviceSynchronize();          /* Wait for GPU to finish */

    return 0;
}   /* main */
```

Program 6.1: CUDA program that prints greetings from the threads.

```
Hello from thread 0!
```

If we want to run ten threads on the GPU, we can type

```
$ ./cuda_hello 10
```

and the output of will be

```
Hello from thread 0!
Hello from thread 1!
Hello from thread 2!
```

NIPulse

```
Hello from thread 3!
Hello from thread 4!
Hello from thread 5!
Hello from thread 6!
Hello from thread 7!
Hello from thread 8!
Hello from thread 9!
```

6.5 A closer look

So what exactly happens when we run `cuda_hello`? Let's take a closer look.

As we noted earlier, execution begins on the host in the `main` function. It gets the number of threads from the command line by calling the C library `strtol` function.

Things get interesting in the call to the kernel in Line 18. Here we tell the system how many threads to start on the GPU by enclosing the pair

`1 , thread_count`

in triple angle brackets. If there were any arguments to the `Hello` function, we would enclose them in the following parentheses.

The kernel specifies the code that each thread will execute. So each of our threads will print a message

```
"Hello from thread %d\n"
```

The decimal int format specifier (`%d`) refers to the variable `threadIdx.x`. The struct `threadIdx` is one of several variables defined by CUDA when a kernel is started. In our example, the field `x` gives the relative index or rank of the thread that is executing.
So we use it to print a message containing the thread's rank.

After a thread has printed its message, it terminates execution.

Notice that our kernel code uses the Single-Program Multiple-Data or SPMD paradigm: each thread runs a copy of the same code on its own data. In this case, the only thread-specific data is the thread rank stored in `threadIdx.x`.

One very important difference between the execution of an ordinary C function and a CUDA kernel is that kernel execution is **asynchronous**. This means that the call to the kernel on the host *returns* as soon as the host has notified the system that it should start running the kernel, and even though the call in `main` has returned, the threads executing the kernel may not have finished executing. The call to `cudaDeviceSynchronize` in Line 21 forces the `main` function to wait until all the threads executing the kernel have completed. If we omitted the call to `cudaDeviceSynchronize`, our program could terminate before the threads produced any output, and it might appear that the kernel was never called.

When the host returns from the call to `cudaDeviceSynchronize`, the `main` function then terminates as usual with **return** 0. To summarize, then:

- Execution begins in `main`, which is running on the host.
- The number of threads is taken from the command line.
- The call to `Hello` starts the kernel.

NIIPulse

- The <<<1, `thread_count`>>> in the call specifies that `thread_count` copies of the kernel should be started on the device.
- When the kernel is started, the struct `threadIdx` is initialized by the system, and in our example the field `threadIdx.x` contains the thread's index or rank.
- Each thread prints its message and terminates.
- The call to `cudaDeviceSynchronize` in `main` forces the host to wait until all of the threads have completed kernel execution before continuing and terminating.

6.6 Threads, blocks, and grids

You're probably wondering why we put a "1" in the angle brackets in our call to `Hello`:

```
Hello <<<1, thread_count >>>();
```

Recall that an Nvidia GPU consists of a collection of streaming multiprocessors (SMs), and each streaming multiprocessor consists of a collection of streaming processors (SPs). When a CUDA kernel runs, each individual thread will execute its code on an SP. With "1" as the first value in angle brackets, all of the threads that are started by the kernel call will run on a single SM. If our GPU has two SMs, we can try to use both of them with the kernel call

```
Hello <<<2, thread_count /2 >>>();
```

If `thread_count` is even, this kernel call will start a total of `thread_count` threads, and the threads will be divided between the two SMs: `thread_count`/2 threads will run on each SM. (What happens if `thread_count` is odd?)

CUDA organizes threads into blocks and grids. A **thread block** (or just a **block** if the context makes it clear) is a collection of threads that run on a single SM. In a kernel call the first value in the angle brackets specifies the number of thread blocks. The second value is the number of threads in each thread block. So when we started the kernel with

```
Hello <<<1, thread_count >>>();
```

we were using one thread block, which consisted of `thread_count` threads, and, as a consequence, we only used one SM.

We can modify our greetings program so that it uses a user-specified number of blocks, each consisting of a user-specified number of threads. (See Program 6.2.) In this program we get both the number of thread blocks and the number of threads in each block from the command line. Now the kernel call starts `blk_ct` thread blocks, each of which contains `th_per_blk` threads.

When the kernel is started, each block is assigned to an SM, and the threads in the block are then run on that SM. The output is similar to the output from the original program, except that now we're using two system-defined variables: `threadIdx.x` and `blockIdx.x`. As you've probably guessed,

`threadIdx.x` gives a thread's rank or index in its block, and `blockIdx.x` gives a block's rank in the *grid*.

A **grid** is just the collection of thread blocks started by a kernel. So a thread block is composed of threads, and a grid is composed of thread blocks.

There are several built-in variables that a thread can use to get information on the grid started by the kernel. The following four variables are structs that are initialized in each thread's memory when a kernel begins execution:

- `threadIdx`: the rank or index of the thread in its thread block.

```
#include < s t d i o . h>
#include <cuda . h>              /* Header    f i l e    for CUDA */

/* Device      code :     runs    on GPU */
__global__ void Hello ( void )          {

    printf ( "Hello from thread %d in block %d\n" , threadIdx . x , blockIdx
          . x ) ;
}      /* Hello */

/* Host      code :      Runs on CPU */
int main ( int argc ,          char * argv [ ] )    {
      int blk_ct ;                        /* Number    of    thread      blocks */
      int th_per_blk ;              /* Number    of    threads    in    each      block */

       blk_ct = strtol ( argv [1] , NULL ,        10);
                        /* Get    number    of    blocks      from command      l i n e */
        th_per_blk = strtol ( argv [2] , NULL ,        10);
          /* Get    number    of    threads    per    block      from command      l i n e */

    Hello <<<blk_ct , th_per_blk > > >();
               /* S t a r t      blk_ct * th_per_blk      threads      on GPU, */

      cudaDeviceSynchronize ( ) ;                /* Wait      for GPU to      f i n i s h */

     return    0;
}      /* main */
```

- `blockDim`: the dimensions, shape, or size of the thread blocks.

1
2

NITPulse

Program 6.2: CUDA program that prints greetings from threads in multiple blocks.

- `blockIdx`: the rank or index of the block within the grid.
- `gridDim`: the dimensions, shape, or size of the grid.

All of these structs have three fields, `x`, `y`, and `z`,[5] and the fields all have unsigned integer types. The fields are often convenient for applications. For example, an application that uses graphics may find it convenient to assign a thread to a point in two- or three-dimensional space, and the fields in `threadIdx` can be used to indicate the point's position. An application that makes extensive use of matrices may find it convenient to assign a thread to an element of a matrix, and the fields in `threadIdx` can be used to indicate the column and row of the element.

6.7 Nvidia compute capabilities and device architectures

When we call a kernel with something like

```
int blk_ct , th_per_blk ;
```

[5] Nvidia devices that have compute capability < 2 (see Section 6.7) only allow x- and y-dimensions in a grid.

NTPulse

```
. . .
Hello <<<blk_ct , th_per_blk >>>();
```

the three-element structures `gridDim` and `blockDim` are initialized by assigning the values in angle brackets to the `x` fields. So, effectively, the following assignments are made:

```
gridDim . x = blk_ct ; blockDim . x = th_per_blk ;
```

The `y` and `z` fields are initialized to 1. If we want to use values other than 1 for the `y` and `z` fields, we should declare two variables of type `dim3`, and pass them into the call to the kernel. For example,

```
dim3 grid_dims , block_dims ; grid_dims . x = 2;
grid_dims . y = 3; grid_dims . z = 1; block_dims . x
= 4; block_dims . y = 4; block_dims . z = 4;
. . .
        Kernel <<<grid_dims , block_dims >>>      ( . . . ) ;
```

This should start a grid with $2 \times 3 \times 1 = 6$ blocks, each of which has $4^3 = 64$ threads. Note that all the blocks must have the same dimensions. More importantly, CUDA requires that thread blocks be independent. So one thread block must be able to complete its execution, regardless of the states of the other thread blocks: the thread blocks can be executed sequentially in any order, or they can be executed in parallel. This ensures that the GPU can schedule a block to execute solely on the basis of the state of that block: it doesn't need to check on the state of any other block.[6]

6.7 Nvidia compute capabilities and device architectures[7]

There are limits on the number of threads and the number of blocks. The limits depend on what Nvidia calls the **compute capability** of the GPU. The compute capability is a number having the form a.b. Currently the a-value or major revision number can be 1,2,3,5,6,7,8. (There is no major revision number 4.) The possible b-values or minor revision numbers depend on the major revision value, but currently

Table 6.3 GPU architectures and compute capabilities.

Name	Ampere	Tesla	Fermi	Kepler	Maxwell	Pascal	Volta	Turing
Compute capability	8.0	1.b	2.b	3.b	5.b	6.b	7.0	7.5

they fall in the range 0–7. CUDA no longer supports devices with compute capability < 3.

For devices with compute capability > 1, the maximum number of threads per block is 1024. For devices with compute capability 2.b, the maximum number of threads that can be assigned to a single SM is 1536, and for devices with compute capability > 2, the maximum is currently 2048.

[6] With the introduction of CUDA 9 and the Pascal processor, it became possible to synchronize threads in multiple blocks. See Subsection 7.1.13 and Exercise 7.6.

[7] The values in this section are current as of spring 2021, but some of them may change when Nvidia releases new GPUs and new versions of CUDA.

There are also limits on the sizes of the dimensions in both blocks and grids. For example, for compute capability > 1, the maximum `x`- or `y`-dimension is 1024, and the maximum `z`-dimension is 64. For further information, see the appendix on compute capabilities in the CUDA C++ Programming Guide [11].

Nvidia also has names for the microarchitectures of its GPUs. Table 6.3 shows the current list of architectures and some of their corresponding compute capabilities. Somewhat confusingly, Nvidia also uses Tesla as the name for their products targeting GPGPU.

We should note that Nvidia has a number of "product families" that can consist of anything from an Nvidia-based graphics card to a "system on a chip," which has the main hardware components of a system, such as a mobile phone in a single integrated circuit.

Finally, note that there are a number of versions of the CUDA API, and they do *not* correspond to the compute capabilities of the different GPUs.

6.8 Vector addition

GPUs and CUDA are designed to be especially effective when they run data-parallel programs. So let's write a very simple, data-parallel CUDA program that's embarrassingly parallel: a program that adds two vectors or arrays. We'll define three n-element arrays, `x`, `y`, and `z`. We'll initialize `x` and `y` on the host. Then a kernel can start at least n threads, and the ith thread will add

```
z[i] = x[i] + y[i];
```

Since GPUs tend to have more 32-bit than 64-bit floating point units, let's use arrays of `floats` rather than `doubles`:

```
float *x, *y, *z;
```

After allocating and initializing the arrays, we'll call the kernel, and after the kernel completes execution, the program checks the result, frees memory, and quits. See Program 6.3, which shows the kernel and the main function.

Let's take a closer look at the program.

NTFlse

```
__global__ void Vec_add ( const float x [ ] /* in */
    , const float y [ ] /* in */ , float z [ ] /* out
    */ , const int n /* in */ ) {
  int my_elt = blockDim . x * blockIdx . x + threadIdx . x ;

  /* t o t a l threads = blk_ct * th_per_blk may be > n */ if ( my_elt < n ) z
  [ my_elt ] = x [ my_elt ] + y [ my_elt ] ;
}  /* Vec_add */

int main ( int argc , char* argv [ ] ) {
  int n , th_per_blk , blk_ct ;
  char i_g ;  /* Are x and y user input or random? */
  float *x , *y , *z , *cz ;
  double diff_norm ;

  /* Get the command l i n e arguments , and set up vectors Get_args ( argc , argv
  , &n , &blk_ct , &th_per_blk , &i_g ) ;                                    */
  Allocate_vectors(&x , &y , &z , &cz , n ) ;                                */
  Init_vectors (x , y , n , i_g ) ;

  /* Invoke kernel and wait for i t to complete

  Vec_add <<<blk_ct , th_per_blk >>>(x , y , z , n ) ; cudaDeviceSynchronize
  ( ) ;

  /* Check for correctness */

  Serial_vec_add (x , y , cz , n ) ; diff_norm = Two_norm_diff (z , cz , n
  ) ; printf ( "Two-norm of difference between host and " ) ; printf (
  "device = %e\n" , diff_norm ) ;

  /* Free storage and quit */

  Free_vectors (x , y , z , cz ) ; return
      0;
}  /* main */
```

NVPulse

```
 5
 6
 7
 8
 9
10
11
12
13 14
15 16 17
18
19
20 21
22
23
24
25 26
27
28
29
30
31
32
33
34
35
36
37
```

Program 6.3: Kernel and `main` function of a CUDA program that adds two vectors.

6.8.1 The kernel

In the kernel (Lines 1–11), we first determine which element of `z` the thread should compute. We've chosen to make this index the same as the *global* rank or index of the thread. Since we're only using the `x` fields of the `blockDim` and `threadIdx` structs, there are a total of

Table 6.4 Global thread ranks or indexes in a grid with 4 blocks and 5 threads per block.

	threadIdx.x				
blockIdx.x	0	1	2	3	4
0	0	1	2	3	4

NTPulse

1	5	6	7	8	9
2	10	11	12	13	14
3	15	16	17	18	19

`gridDim.x * blockDim.x`

threads. So we can assign a unique "global" rank or index to each thread by using the formula

```
rank = blockDim.x * blockIdx.x + threadIdx.x
```

For example, if we have four blocks and five threads in each block, then the global ranks or indexes are shown in Table 6.4. In the kernel, we assign this global rank to `my_elt` and use this as the subscript for accessing each thread's elements of the arrays `x`, `y`, and `z`.

Note that we've allowed for the possibility that the total number of threads may not be exactly the same as the number of components of the vectors. So before carrying out the addition,

```
z[my_elt] = x[my_elt] + y[my_elt];
```

we first check that `my_elt < n`. For example, if we have n = 997, and we want at least two blocks with at least two threads per block, then, since 997 is prime, we can't possibly have exactly 997 threads. Since this kernel needs to be executed by at least n threads, we must start more than 997. For example, we might use four blocks of 256 threads, and the last 27 threads in the last block would skip the line

```
z[my_elt] = x[my_elt] + y[my_elt];
```

Note that if we needed to run our program on a system that didn't support CUDA, we could replace the kernel with a serial vector addition function. (See Program 6.4.) So we can view the CUDA kernel as taking the serial **for** loop and assigning each iteration to a different thread. This is often how we start the design process when we want to parallelize a serial code for CUDA: assign the iterations of a loop to individual threads.

Also note that if we apply Foster's method to parallelizing the serial vector sum, and we make the tasks the additions of the individual components, then we don't need to do anything for the communication and aggregation phases, and the mapping phase simply assigns each addition to a thread.

NVIDIA

```
void   Serial_vec_add (                    */,
       const   float x [ ]          /* in const */,
                  float y [ ]        /* in     */,
         float cz [ ] /* out const int n  /* in  */)


    for ( int i =       0; i < n ; i++) cz [ i
        ]     = x [ i ] + y [ i ];              {
}   /* Serial_vec_add */
```

Program 6.4: Serial vector addition function.

6.8.2 `Get_args`

After declaring the variables, the `main` function calls a `Get_args` function, which returns n, the number of elements in the arrays, `blk_ct`, the number of thread blocks, and `th_per_blk`, the number of threads in each block. It gets these from the command line. It also returns a **char** `i_g`. This tells the program whether the user will *i*nput `x` and `y` or whether it should *g*enerate them using a random number generator. If the user doesn't enter the correct number of command line arguments, the function prints a usage summary and terminates execution. Also if n is greater than the total number of threads, it prints a message and terminates. (See Program 6.5.) Note that `Get_args` is written in standard C, and it runs completely on the host.

6.8.3 `Allocate_vectors` and managed memory

After getting the command line arguments, the main function calls `Allocate_vectors`, which allocates storage for four n-element arrays of **float**:

```
x , y , z , cz
```

The first three arrays are used on both the host and the device. The fourth array, `cz`, is only used on the host: we use it to compute the vector sum with one core of the host. We do this so that we can check the result computed on the device. (See Program 6.6.)

NVIDIA

First note that since `cz` is only used on the host, we allocate its storage using the standard C library function `malloc`. For the other three arrays, we allocate storage in Lines 9–11 using the CUDA function

```
__host__ cudaError_t cudaMallocManaged (
    void** devPtr /* out */,
    size_t size /* in */,
    unsigned flags /* in */);
```

The `__host__` qualifier is a CUDA addition to C, and it indicates that the function

```
void Get_args (
    const int argc /* in */, char* argv[] /* in */, int* n_p /* out */,
    int* blk_ct_p /* out */, int** th_per_blk_p /* out */,
    char* i_g /* out */) {
  if (argc != 5) {
    /* Print an error message and exit */
    ...
  }

  *n_p = strtol(argv[1], NULL, 10); *blk_ct_p = strtol
  (argv[2], NULL, 10); *th_per_blk_p = strtol(argv[3]
  , NULL, 10); *i_g = argv[4][0];

  /* Is n > total thread count = blk_ct * th_per_blk ?
  if (*n_p > (*blk_ct_p)*(*th_per_blk_p)) { /* Print an error
    message and exit */
    ...
  }                                                    */
} /* Get_args */
```

should be called and run on the host. This is the default for functions in CUDA

1
2
3
4 5 6
7
8
9

NIPulse

Program 6.5: `Get_args` function from CUDA program that adds two vectors.

```
void Allocate_vectors  float (
        ** x_p  float ** y_p /* out */,
        float ** z_p  float **  /* out */,
        cz_p
         int          n      /* out */,
                             /* in */ )       {
    /* x , y , and z are used on host and device cudaMallocManaged  */
    ( x_p , n* sizeof ( float ) ) ; cudaMallocManaged ( y_p , n*
    sizeof ( float ) ) ; cudaMallocManaged ( z_p , n* sizeof ( float
    ) ) ;

     /* cz  i s  only  used  on  host */
    *cz_p  = ( float *) malloc  ( n* sizeof ( float ) ) ; } /*
Allocate_vectors */
```

```
 9 10
11
12
13
14
15
```

Program 6.6: Array allocation function of CUDA program that adds two vectors. programs, so it can be omitted when we're writing our own functions, and they'll only be run on the host.

The return value, which has type `cudaError_t`, allows the function to return an error. Most CUDA functions return a `cudaError_t` value, and if you're having problems with your code, it is a very good idea to check it. However, always checking it tends to clutter the code, and this can distract us from the main purpose of a program. So in the code we discuss we'll generally ignore `cudaError_t` return values.

The first argument is a pointer to a pointer: it refers to the pointer that's being allocated. The second argument specifies the number of bytes that should be allocated. The `flags` argument controls which kernels can access the allocated memory.
It defaults to `cudaMemAttachGlobal` and can be omitted.

The function `cudaMallocManaged` is one of several CUDA memory allocation functions. It allocates memory that will be automatically managed by the "unified memory system." This is a relatively recent addition to CUDA,[8] and it allows a programmer to write CUDA programs as if the host and device shared a single memory: pointers referring to memory allocated with `cudaMallocManaged` can be used on both the device and the host, even when the host and the device have separate physical memories. As you can imagine this greatly simplifies programming, but there are some cautions. Here are a few:

1. Unified memory requires a device with compute capability ≥ 3.0, and a 64-bit host operating system.
2. On devices with compute capability < 6.0 memory allocated with `cudaMallocManaged` cannot be simultaneously accessed by both the device and the host. When a kernel is executing, it has exclusive access to memory allocated with `cudaMallocManaged`.
3. Kernels that use unified memory can be slower than kernels that treat device memory as separate from host memory.

The last caution has to do with the transfer of data between the host and the device. When a program uses unified memory, it is up to the system to decide when to transfer from the host to the device or vice versa. In programs that explicitly transfer data, it is up to the programmer to include code that implements the transfers, and she may be able to exploit

[8] It first became available in CUDA 6.0.

NIPUSE

her knowledge of the code to do things that reduce the cost of transfers, things such as omitting some transfers or overlapping data transfer with computation.

At the end of this section we'll briefly discuss the modifications required if you want to explicitly handle the transfers between host and device.

6.8.4 Other functions called from `main`

Except for the `Free_vectors` function, the other host functions that we call from `main` are just standard C.

The function `Init_vectors` either reads `x` and `y` from `stdin` using `scanf` or generates them using the C library function `random`. It uses the last command line argument `i_g` to decide which it should do.

The `Serial_vec_add` function (Program 6.4) just adds `x` and `y` on the host using a **for** loop. It stores the result in the host array `cz`.

The `Two_norm_diff` function computes the "distance" between the vector `z` computed by the kernel and the vector `cz` computed by `Serial_vec_add`. So it takes the difference between corresponding components of `z` and `cz`, squares them, adds the squares, and takes the square root:

$$\sqrt{(z[0]-cz[0])^2+(z[1]-cz[1])^2+\cdots+(z[n-1]-cz[n-1])^2}.$$

See Program 6.7.

```
double      Two_norm_diff (
            const float z [ ] /* const float
                                  in */ , in
            cz [ ] /* const int n /*
                                  */ , in */ )
      double diff , sum =       0.0;
                                          {
      for ( int i = 0; i < n ; i++) { diff = z [ i ]
            - cz [ i ];

            sum += diff*diff ;

      }
      return sqrt ( sum ) ;
}     /* Two_norm_diff */
```

NTPulse

```
 8 9 10
11
12
```

Program 6.7: C function that finds the distance between two vectors.

The `Free_vectors` function just frees the arrays allocated by `Allocate_vectors`. The array `cz` is freed using the C library function `free`, but since the other arrays are allocated using `cudaMallocManaged`, they must be freed by calling `cudaFree`:

```
__host__ __device__ cudaError_t cudaFree ( void * ptr )
```

The qualifier `__device__` is a CUDA addition to C, and it indicates that the function can be called from the device. So `cudaFree` can be called from the host or the device. However, if a pointer is allocated on the device, it cannot be freed on the host, and vice versa.

It's important to note that unless memory allocated on the device is explicitly freed by the program, it won't be freed until the program terminates. So if a CUDA program calls two (or more) kernels, and the memory used by the first kernel isn't explicitly freed before the second is called, it will remain allocated, regardless of whether the second kernel actually uses it.

See Program 6.8.

```
void
    Free_vectors (
      float * x float *  /*
      * y float * z      /*  in / out */ , in /
      float * cz         /*  out */ , in / out */
                         /*  , in / out */ ) {
    /*      Allocatedwith   cudaMallocManaged      */
    cudaFree ( x ) ;
    cudaFree ( y ) ;
    cudaFree ( z ) ;

    /* Allocated free (
    cz ) ;              with   malloc */
}   /* Free_vectors */
```

```
1
2
3
4
5
```

NVPulse

Program 6.8: CUDA function that frees four arrays.

6.8.5 Explicit memory transfers[9]

Let's take a look at how to modify the vector addition program for a system that doesn't provide unified memory. Program 6.9 shows the kernel and main function for the modified program.

The first thing to notice is that the kernel is unchanged: the arguments are `x`, `y`, `z`, and `n`. It finds the thread's global index, `my_elt`, and if this is less than `n`, it adds the elements of `x` and `y` to get the corresponding element of `z`.

The basic structure of the `main` function is almost the same. However, since we're assuming unified memory is unavailable, pointers on the host aren't valid on the device, and vice versa: an address on the host may be illegal on the device, or, even worse, it might refer to memory that the device is using for some other purpose. Similar problems occur if we try to use a device address on the host. So instead of declaring and allocating storage for three arrays that are all valid on both the host and the device, we declare and allocate storage for three arrays that are valid on the host `hx`, `hy`, and `hz`, *and* we declare and allocate storage for three arrays that are valid on the device, `dx`, `dy`, and `dz`. The declarations are in Lines 15–16, and the allocations are in the `Allocate_vectors` function called in Line 20. The function itself is in Program 6.10. Since unified memory isn't available, instead of using `cudaMallocManaged`, we use the C library function `malloc` for the host arrays, and the CUDA function `cudaMalloc` for the device arrays:

```
__host__ __device__ cudaError_t cudaMalloc ( void** dev_p /*
                out */, size_t size /* in */) ;
```

[9] If your device has compute capability ≥**3.0**, you can skip this section.

```
__global__ void Vec_add (
        const float x [ ]          /* in */, const
        float y [ ]          /* in */, float      z [ ]
        /* out */, const     int       n          /* in */
        ){
    int my_elt = blockDim . x * blockIdx . x + threadIdx . x ;

    i f ( my_elt < n ) z [ my_elt ] = x [ my_elt ] + y [
        my_elt ] ;
}      /* Vec_add */

int main ( int argc , char* argv [ ] ) { int n ,
    th_per_blk , blk_ct ;
    char i_g ;    /* Are   x   and   y   user   input   or
        random? */ float *hx , *hy , *hz , *cz ;      /* Host   arrays   */
    float *dx , *dy , *dz ;  /* Device arrays   */ double diff_norm ;

    Get_args ( argc , argv , &n , &blk_ct , &th_per_blk , &i_g ) ;
    Allocate_vectors(&hx , &hy , &hz , &cz , &dx , &dy , &dz , n ) ;
    Init_vectors ( hx , hy , n , i_g ) ;

     /* Copy   vectors   x   and   y   from   host   to   device */
 cudaMemcpy ( dx , hx , n* sizeof ( float ) , cudaMemcpyHostToDevice ) ; cudaMemcpy ( dy ,
                hy , n* sizeof ( float ) , cudaMemcpyHostToDevice ) ;

    Vec_add <<<blk_ct , th_per_blk >>>(dx , dy , dz , n ) ;

    /* Wait for kernel to complete and copy r e s u l t to host */ cudaMemcpy ( hz , dz , n*
    sizeof ( float ) , cudaMemcpyDeviceToHost ) ;

    Serial_vec_add ( hx , hy , cz , n ) ; diff_norm = Two_norm_diff ( hz ,
    cz , n ) ; printf ( "Two-norm of difference between host and " ) ;
```

NVPulse

```
    printf ("device = %e\n",diff_norm);Free_vectors (hx,hy,hz,
    cz,dx,dy,dz);

     return 0;
}      /* main */
```

1
2
3
4
5 6

NITPulse

```
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
```

Program 6.9: Part of CUDA program that implements vector addition without unified memory.

NITPulse

```
void Allocate_vectors ( float ** hx_p /*
        out */ , float ** hy_p /* out */ ,
        float ** hz_p /* out */ , float **
        cz_p /* out */ , float ** dx_p /*
        out */ , float ** dy_p /* out */ ,
        float ** dz_p /* out */ ,
        int         n           /* in */ )      {

    /* dx , dy , and dz are used on device */ cudaMalloc ( dx_p
    , n* sizeof ( float ) ) ; cudaMalloc ( dy_p , n* sizeof ( float
    ) ) ; cudaMalloc ( dz_p , n* sizeof ( float ) ) ;

        /* hx , hy , hz , cz are used on host */ hx_p = ( float *)
     *
        malloc ( n* sizeof ( float ) ) ; *hy_p = ( float *) malloc
     *
        ( n* sizeof ( float ) ) ; hz_p = ( float *) malloc ( n*
    sizeof ( float ) ) ;
    *cz_p = ( float *) malloc ( n* sizeof ( float ) ) ; } /*
Allocate_vectors */
```

NTPulse

Program 6.10: `Allocate_vectors` function for CUDA vector addition program that doesn't use unified memory.

The first argument is a reference to a pointer that will be used *on the device*. The second argument specifies the number of bytes to allocate on the device.

After we've initialized `hx` and `hy` on the host, we copy their contents over to the device, storing the transferred contents in the memory allocated for `dx` and `dy`, respectively. The copying is done in Lines 24–26 using the CUDA function `cudaMemcpy`:

```
__host__   cudaError_t cudaMemcpy (
        void*          dest      /* out */,
        const void*    source    /* in  */,
        size_t         count     /* in  */,
        cudaMemcpyKind kind      /* in  */);
```

This copies `count` bytes from the memory referred to by `source` into the memory referred to by `dest`. The type of the `kind` argument, `cudaMemcpyKind`, is an enumerated type defined by CUDA that specifies where the `source` and `dest` pointers are located. For our purposes the two values of interest are `cudaMemcpyHostToDevice` and `cudaMemcpyDeviceToHost`. The first indicates that we're copying from the host to the device, and the second indicates that we're copying from the device to the host.

The call to the kernel in Line 28 uses the pointers `dx`, `dy`, and `dz`, because these are addresses that are valid on the *device*.

After the call to the kernel, we copy the result of the vector addition from the device to the host in Line 31 using `cudaMemcpy` again. A call to `cudaMemcpy` is *synchronous*, so it waits for the kernel to finish executing before carrying out the transfer. So in this version of vector addition we do *not* need to use `cudaDeviceSynchronize` to ensure that the kernel has completed before proceeding.

After copying the result from the device back to the host, the program checks the result, frees the memory allocated on the host and the device, and terminates. So for this part of the program, the only difference from the original program is that we're freeing seven pointers instead of four. As before, the `Free_vectors` function frees the storage allocated on the host with the C library function `free`. It uses `cudaFree` to free the storage allocated on the device.

6.9 Returning results from CUDA kernels

There are several things that you should be aware of regarding CUDA kernels. First, they always have return type **void**, so they can't be used to return a value. They also can't return

anything to the host through the standard C pass-by-reference. The reason for this is that addresses on the host are, in most systems, invalid on the device, and vice versa. For example, suppose we try something like this:

```
__global__ void Add(int x, int y, int* sum_p) {
    *sum_p = x + y;
} /* Add */

int main(void) { int sum =
    -5;
    Add <<<1, 1>>> (2, 3, &sum);
    cudaDeviceSynchronize(); printf("The sum
    is %d\n", sum);
return 0;
}
```

It's likely that either the host will print -5 or the device will hang. The reason is that the address `&sum` is probably invalid on the device. So the dereference

```
*sum_p = x + y;
```

is attempting to assign `x + y` to an invalid memory location.

There are several possible approaches to "returning" a result to the host from a kernel. One is to declare pointer variables and allocate a single memory location. On a system that supports unified memory, the computed value will be automatically copied back to host memory:

```
__global__ void Add(int x, int y, int* sum_p) {
    *sum_p = x + y;
} /* Add */
```

NTPulse

```
int main ( void ) { int * sum_p ; cudaMallocManaged(&sum_p ,
    sizeof ( int ) ) ; *sum_p = -5;
    Add <<<1, 1>>> (2 , 3 , sum_p ) ;
    cudaDeviceSynchronize ( ) ; printf ( "The sum is
    %d\n" , *sum_p ) ; cudaFree ( sum_p ) ;

        return 0;
}
```

If your system doesn't support unified memory, the same idea will work, but the result will have to be explicitly copied from the device to the host:

```
    __global__ void Add ( int x , int y , int *sum_p ) {
    *sum_p = x + y ;
    } /* Add */

int main ( void ) { int *hsum_p , *dsum_p ; hsum_p = ( int *)
    malloc ( sizeof ( int ) ) ; cudaMalloc(&dsum_p , sizeof (
    int ) ) ; *hsum_p = -5;
    Add <<<1, 1>>> (2 , 3 , dsum_p ) ; cudaMemcpy ( hsum_p ,
    dsum_p , sizeof ( int ) , cudaMemcpyDeviceToHost ) ;
    printf ( "The sum is %d\n" , *hsum_p ) ; free ( hsum_p
    ) ; cudaFree ( dsum_p ) ;

        return 0;
}
```

Note that in both the unified and non-unified memory settings, we're returning a *single* value from the device to the host.

If unified memory is available, another option is to use a global managed variable for the sum:

```
__managed__ int sum ;

__global__ void Add ( int x , int y ) { sum = x + y ;
    } /* Add */
```

NIPulse

```
int main ( void )  {
sum = -5; Add <<<1, 1>>> (2 , 3);

cudaDeviceSynchronize ( ) ;
printf ( "After kernel: The sum is %d\n" , sum ) ;

return 0;
}
```

The qualifier `__managed__` declares `sum` to be a managed **int** that is accessible to all the functions, regardless of whether they run on the host or the device. Since it's managed, the same restrictions apply to it that apply to managed variables allocated with `cudaMallocManaged`. So this option is unavailable on systems with compute capability < 3.0, and on systems with compute capability < 6.0, `sum` can't be accessed on the host while the kernel is running. So after the call to `Add` has started, the host can't access `sum` until after the call to `cudaDeviceSynchronize` has completed.

Since this last approach uses a global variable, it has the usual problem of reduced modularity associated with global variables.

6.10 CUDA trapezoidal rule I

6.10.1 The trapezoidal rule

Let's try to implement a CUDA version of the trapezoidal rule. Recall (see Section 3.2.1) that the trapezoidal rule estimates the area between an interval on the x-axis and the graph of a function by dividing the interval into subintervals and approximating the area between each subinterval and the graph by the area of a trapezoid. (See Fig. 3.3.) So if the interval is [a,b] and there are n trapezoids, we'll divide [a,b] into n equal subintervals, and the length of each subinterval will be

$$h = (b - a)/n.$$

Then if x_i is the left end point of the ith subinterval,

$$x_i = a + ih,$$

for i = 0,1,2,...,n − 1. To simplify the notation, we'll also denote b, the right end point of the interval, as

$$b = x_n = a + nh.$$

Recall that if a trapezoid has height h and base lengths c and d, then its area is

$$\frac{h}{2}(c + d).$$

So if we think of the length of the subinterval $[x_i,x_{i+1}]$ as the height of the ith trapezoid, and $f(x_i)$ and $f(x_{i+1})$ as the two base lengths (see Fig. 3.4), then the area of the ith trapezoid is

$$\frac{h}{2}[f(x_i) + f(x_{i+1})].$$

This gives us a total approximation of the area between the graph and the x-axis as

$$\frac{h}{2}[f(x_0) + f(x_1)] + \frac{h}{2}[f(x_1) + f(x_2)] + \cdots + \frac{h}{2}[f(x_{n-1}) + f(x_n)],$$

and we can rewrite this as

$$h\left[\frac{1}{2}(f(a) + f(b)) + (f(x_1) + f(x_2) + \cdots + f(x_{n-1}))\right].$$

We can implement this with the serial function shown in Program 6.11.

```
float Serial_trap (
      const float a /* in */, const float b /*
      in */, const int n /* in */) {
   float x, h = (b-a)/n; float trap = 0.5*(f
   (a) + f(b));

   for (int i = 1; i <= n-1; i++) { x = a + i*h;
      trap += f(x);
   }
   trap = trap*h;

   return trap;
}     /* Serial_trap */
```

NTPulse

Program 6.11: A serial function implementing the trapezoidal rule for a single CPU.

6.10.2 A CUDA implementation

If n is large, the vast majority of the work in the serial implementation is done by the **for** loop. So when we apply Foster's method to the trapezoidal rule, we're mainly interested in two types of tasks: the first is the evaluation of the function f at x_i, and the second is the addition of $f(x_i)$ into `trap`. Here i = 1,...,n − 1. The second type of task depends on the first. So we can aggregate these two tasks.

This suggests that each thread in our CUDA implementation might carry out one iteration of the serial **for** loop. We can assign a unique integer rank to each thread as we did with the vector addition program. Then we can compute an x-value, the function value, and add in the function value to the "running sum":

```
/* h and trap are formal arguments to the kernel */ int my_i = blockDim.x
* blockIdx.x + threadIdx.x ; float my_x = a + my_i*h ; float my_trap
= f ( my_x ) ; float trap += my_trap ;
```

However, it's immediately obvious that there are several problems here:

1. We haven't initialized `h` or `trap`.
2. The `my_i` value can be too large or too small: the serial loop ranges from 1 up to and including n − 1. The smallest value for `my_i` is 0 and the largest is the total number of threads minus 1.
3. The variable `trap` must be shared among the threads. So the addition of `my_trap` forms a race condition: when multiple threads try to update `trap` at roughly the same time, one thread can overwrite another thread's result, and the final value in trap may be wrong. (For a discussion of race conditions, see Section 2.4.3.)
4. The variable `trap` in the serial code is returned by the function, and, as we've seen, kernels must have **void** return type.
5. We see from the serial code that we need to multiply the total in `trap` by `h` after all of the threads have added their results.

Program 6.12 shows how we might deal with these problems. In the following sections, we'll look at the rationales for the various choices we've made.

6.10.3 Initialization, return value, and final update

To deal with the initialization and the final update (Items 1 and 5), we could try to select a single thread—say, thread 0 in block 0—to carry out the operations:

```
int my_i = blockDim.x * blockIdx.x + threadIdx.x ;
if ( my_i == 0) {
```

NVIDIA

```
    h = (b-a)/n; trap = 0.5*(f(a) + f(b));
  }...
    if (my_i == 0)
      trap = trap*h;
```

There are (at least) a couple of problems with these options: formal arguments to functions are private to the executing thread and **thread synchronization**.

Kernel and function arguments are private to the executing thread.

Like the threads started in Pthreads and OpenMP, each CUDA thread has its own stack and, and since formal arguments are allocated on the thread's stack, each thread has its own private variables `h` and `trap`. So any changes made to one of these variables by one thread won't be visible to the other threads. We could have each thread initialize `h`, but we could also just do the initialization once in the host. If we do this before the kernel is called, each thread will get a copy of the value of `h`.

Things are more complicated with `trap`. Since it's updated by multiple threads, it must be *shared* among the threads. We can achieve the *effect* of sharing `trap` by allocating storage for a memory location before the kernel is called. This allocated memory location will correspond to what we've been calling `trap`. Now we can pass

NITPulse

```
__global__ void Dev_trap (
        const       float a    /* in       */, const float b    /* in
         */, const float h    /* in       */, const int       n
         /* in       */,
         float *                  trap_p /* in / out */)       {
    int my_i = blockDim .x * blockIdx .x + threadIdx .x ;

    /* f ( x_0 ) and f ( x_n ) were computed on the host . So */ /* compute f ( x_1 ) , f
    ( x_2 ) , . . . , f ( x_ (n-1)) */ if (0 < my_i && my_i < n ) { float my_x = a +
    my_i*h ; float my_trap = f ( my_x ) ;
        atomicAdd ( trap_p , my_trap ) ;
    }
}       /* Dev_trap */

/* Host       code */
void Trap_wrapper (
        const float a /* in */, const float b /* in */, const int
        n /* in */, float * trap_p /* out */, const int blk_ct
        /* in */, const int th_per_blk /* in */) {

      /* trap_p      storage      allocated      in    main    with
      * cudaMallocManaged */
      *trap_p = 0.5*( f ( a ) + f ( b ) ) ; float h =
      (b-a ) / n ;

    Dev_trap <<<blk_ct , th_per_blk >>>(a , b , h , n , trap_p ) ;
    cudaDeviceSynchronize ( ) ;

    *trap_p = h*(*trap_p ) ;
}       /* Trap_wrapper */
```

NIPulse

```
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29 30
31
32
33
34
35
36
```

Program 6.12: CUDA kernel and wrapper implementing trapezoidal rule.

a *pointer* to the memory location to the kernel. That is, we can do something like this:

```
/* Host code */ float *
trap_p ;
   cudaMallocManaged(&trap_p ,      sizeof ( float ) ) ;
. . .
*trap_p =    0.5*( f ( a ) + f ( b ) ) ;

/* Call       kernel */
   . . .
/* After      return    from    kernel */
   *trap_p = h*(*trap_p ) ;
```

When we do this, each thread will get its own copy of `trap_p`, but all of the copies of `trap_p` will refer to the same memory location. So `*trap_p` will be shared.

NTPulse

Note that using a pointer instead of a simple float also solves the problem of returning the value of `trap` in Item 4.

A wrapper function

If you look at the code in Program 6.12, you'll see that we've placed most of the code we use before and after calling the kernel in a **wrapper function**, `Trap_wrapper`. A wrapper function is a function whose main purpose is to call another function. It can perform any preparation needed for the call. It can also perform any additional work needed after the call.

6.10.4 Using the correct threads

We assume that the number of threads, `blk_ct*th_per_blk`, is at least as large as the number of trapezoids. Since the serial **for** loop iterates from 1 up to `n`−1, thread 0 and any thread with `my_i` > n− 1, shouldn't execute the code in the body of the serial **for** loop. So we should include a test before the main part of the kernel code

```
if (0 < my_i && my_i < n ) {
   /* Compute x, f(x), and add in to *trap_p */
      . . .
}
```

See Line 11 in Program 6.12.

6.10.5 Updating the return value and `atomicAdd`

This leaves the problem of updating `*trap_p` (Item 3 in the list above). Since the memory location is shared, an update such as

```
*trap_p += my_trap ;
```

forms a race condition, and the actual value ultimately stored in `*trap_p` will be unpredictable. We're solving this problem by using a special CUDA library function, `atomicAdd`, to carry out the addition.

An operation carried out by a thread is **atomic** if it appears to all the other threads as if it were "indivisible." So if another thread tries to access the result of the operation or an operand used in the operation, the access will occur either before the operation started or after the operation completed. Effectively, then, the operation appears to consist of a single, indivisible, machine instruction.

As we saw earlier (see Section 2.4.3), addition is not ordinarily an atomic operation: it consists of several machine instructions. So if one thread is executing an addition, it's possible for another thread to access the operands and the result while the addition is in

NKPulse

progress. Because of this, the CUDA library defines several atomic addition functions. The one we're using has the following syntax:

```
__device__ float atomicAdd ( float * float_p         /* in /
        out */, float   val       /* in       */ ) ;
```

This atomically adds the contents of `val` to the contents of the memory referred to by `float_p` and stores the result in the memory referred to by `float_p`. It returns the value of the memory referred to by `float_p` at the beginning of the call. See Line 14 of Program 6.12.

6.10.6 Performance of the CUDA trapezoidal rule

We can find the run-time of our trapezoidal rule by finding the execution time of the `Trap_wrapper` function. The execution of this function includes all of the computations carried out by the serial trapezoidal rule, including the initialization of `*trap_p` (Line 29) and `h` (Line 30), and the final update to `*trap_p` (Line 35). It also includes all of the calculations in the body of the serial **for** loop in the `Dev_trap` kernel. So we can effectively determine the run-time of the CUDA trapezoidal rule by timing a host function, and we only need to insert calls to our timing functions before and after the call to `Trap_wrapper`. We use the `GET_TIME` macro defined in the `timer.h` header file on the book's website:

```
double start , finish ;
...
GET_TIME ( start );
Trap_wrapper (a , b , n , trap_p , blk_ct , th_per_blk ); GET_TIME ( finish );
printf ( "Elapsed time for cuda = %e seconds\n" , finish-start );
```

The same approach can be used to time the serial trapezoidal rule:

```
GET_TIME ( start ) trap = Serial_trap (a ,
b , n ); GET_TIME ( finish );
printf ( "Elapsed time for cpu = %e seconds\n" , finish-start );
```

Recall from the section on taking timings (Section 2.6.4) that we take a number of timings, and we ordinarily report the minimum elapsed time. However, if the vast majority of the times are much greater (e.g., 1% or 0.1% greater), then the minimum time may not be reproducible. So other users who run the program may get a time

Table 6.5 Mean run-times for serial and CUDA trapezoidal rule (times are in ms).

System	ARM Cortex-A15	Nvidia GK20A	Intel Core i7	Nvidia GeForce GTX Titan X
Clock	2.3 GHz	852 MHz	3.5 GHz	1.08 GHz
SMs, SPs		1, 192		24, 3072
Run-time	33.6	20.7	4.48	3.08

much larger than ours. When this happens, we report the mean or median of the elapsed times.

NTPulse

Now when we ran this program on our hardware, there *were* a number of times that were within 1% of the minimum time. However, we'll be comparing the runtimes of this program with programs that had very few run-times within 1% of the minimum. So for our discussion of implementing the trapezoidal rule using CUDA (Sections 6.10–6.13), we'll use the *mean* run-time, and the means are taken over at least 50 executions.

When we run the serial trapezoidal and the CUDA trapezoidal rule functions many times and take the means of the elapsed times, we get the results shown in Table 6.5. These were taken using $n = 2^{20} = 1{,}048{,}576$ trapezoids with $f(x) = x^2 + 1$, $a = -3$, and $b = 3$. The GPUs use 1024 blocks with 1024 threads per block for a total of 1,048,576 threads. The 192 SPs of the GK20A are clearly much faster than a fairly slow conventional processor, an ARM Cortex-A15, but a single core of an Intel Core i7 is much faster than the GK20A. The 3072 SPs on a Titan X *were* 45% faster than the single core of the Intel, but it would seem that with 3072 SPs, we should be able to do better.

6.11 CUDA trapezoidal rule II: improving performance

If you've read the Pthreads or OpenMP chapter, you can probably make a good guess at how to make the CUDA program run faster. For a thread's call to `atomicAdd` to actually be atomic, no other thread can update `*trap_p` while the call is in progress. In other words, the updates to `*trap_p` can't take place simultaneously, and our program may not be very parallel at this point.

One way to improve the performance is to carry out a tree-structured global sum that's similar to the tree-structured global sum we introduced in the MPI chapter (Section 3.4.1). However, because of the differences between the GPU architecture and the distributed-memory CPU architecture, the details are somewhat different.

6.11.1 Tree-structured communication

We can visualize the execution of the "global sum" we implemented in the CUDA trapezoidal rule as a more or less random, linear ordering of the threads. For ex-

NT Pulse

Table 6.6 Basic global sum with eight threads.

Time	Thread	my_trap	*trap_p
Start	—	—	9
t_0	5	11	20
t_1	2	5	25
t_2	3	7	32
t_3	7	15	47
t_4	4	9	56
t_5	6	13	69
t_6	0	1	70
t_7	1	3	73

ample, suppose we have only 8 threads and one thread block. Then our threads are 0,1,...,7, and one of the threads will be the first to succeed with the call to `atomicAdd`. Say it's thread 5. Then another thread will succeed. Say it's thread 2. Continuing in this fashion we get a sequence of `atomicAdds`, one per thread. Table 6.6 shows how this might proceed over time. Here, we're trying to keep the computations simple: we're assuming that $f(x) = 2x + 1$, $a = 0$, and $b = 8$. So $h = (8 - 0)/8 = 1$, and the value referenced by `trap_p` at the start of the global sum is

$$0.5 \times (f(a) + f(b)) = 0.5 \times (1 + 17) = 9.$$

What's important is that this approach may serialize the threads. So the computation may require a *sequence* of 8 calculations. Fig. 6.3 illustrates a possible computation.

So rather than have each thread wait for its turn to do an addition into `*trap_p`, we can pair up the threads so that half of the "active" threads add their partial sum to their partner's partial sum. This gives us a structure that resembles a tree (or, perhaps better, a shrub). See Fig. 6.4.

In our figures, we've gone from requiring a sequence of 8 consecutive additions to a sequence of 4. More generally, if we double the number of threads and values (e.g., increase from 8 to 16), we'll double the length of the sequence of additions using the basic approach, while we'll only add one using the second, tree-structured approach. For example, if we increase the number of threads and values from 8 to 16, the first approach requires a sequence of 16 additions, but the tree-structured approach only requires 5. In fact, if there are t threads and t values, the first approach requires a sequence of t additions, while the tree-structured approach requires $\log_2(t)+1$. For example, if we have 1000 threads and values, we'll go from 1000 communications and sums using the basic approach to 11 using the tree-structured approach, and if we have 1,000,000, we'll go from 1,000,000 to 21!

There are two standard implementations of a tree-structured sum in CUDA. One implementation uses shared memory, and in devices with compute capability < 3 this

NIPulse

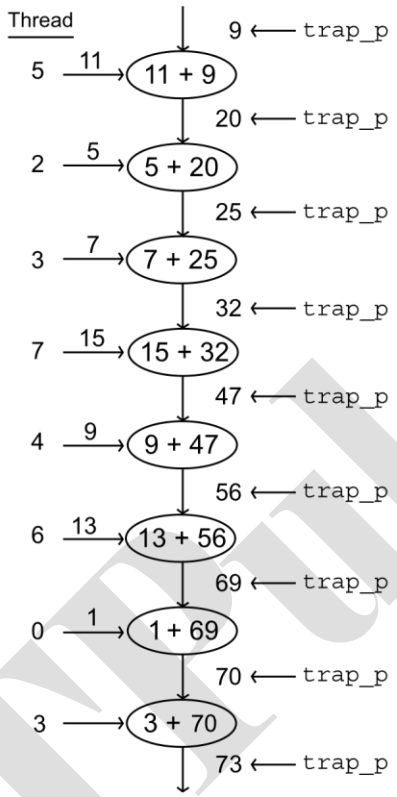

FIGURE 6.3

Basic sum.

is the best implementation. However, in devices with compute capability ≥ 3 there are several functions called **warp shuffles**, that allow a collection of threads within a *warp* to read variables stored by other threads in the warp.

6.11.2 Local variables, registers, shared and global memory

Before we explain the details of how warp shuffles work, let's digress for a moment and talk about memory in CUDA. In Section 6.2 we mentioned that SMs in an Nvidia processor have access to two collections of memory locations: each SM has access to its own "shared" memory, which is accessible only to the SPs belonging to the SM. More precisely, the shared memory allocated for a thread block is only accessible to the threads in that block. On the other hand, all of the SPs and all of the threads have access to "global" memory. The number of shared memory locations is

relatively small, but they are quite fast, while the number of global memory locations is relatively large, but they are relatively slow. So we can think of the GPU memory

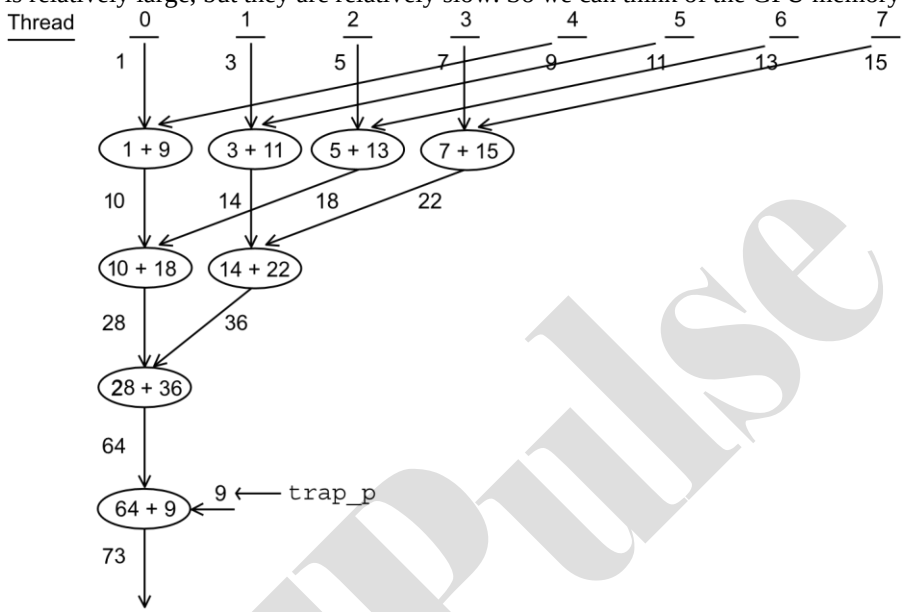

FIGURE 6.4

Tree-structured sum.

Table 6.7 Memory statistics for some Nvidia GPUs.

GPU	Compute Capability	Registers: Bytes per Thread	Shared Mem: Bytes per Block	Global Mem: Bytes per GPU
Quadro 600	2.1	504	48K	1G
GK20A (Jetson TK1)	3.2	504	48K	2G
GeForce GTX Titan X	5.2	504	48K	12G

as a hierarchy with three "levels." At the bottom, is the slowest, largest level: global memory. In the middle is a faster, smaller level: shared memory. At the top is the fastest, smallest level: the registers. For example, Table 6.7 gives some information on relative sizes. Access times also increase dramatically. It takes on the order of 1 cycle to copy a 4-byte int from one register to another. Depending on the system it can take up to an order of magnitude more time to copy from one shared memory location to another, and it can take from two to three orders of magnitude more time to copy from one global memory location to another.

An obvious question here: what about local variables? How much storage is available for them? And how fast is it? This depends on total available memory and program memory usage. If there is enough storage, local variables are stored in

NVPulse

registers. However, if there isn't enough register storage, local variables are "spilled" to a region of global memory that's thread private, i.e., only the thread that owns the local variables can access them.

So as long as we have sufficient register storage, we expect the performance of a kernel to improve if we increase our use of registers and reduce our use of shared and/or global memory. The catch, of course, is that the storage available in registers is tiny compared to the storage available in shared and global memory.

6.11.3 Warps and warp shuffles

In particular, if we can implement a global sum in registers, we expect its performance to be superior to an implementation that uses shared or global memory, and the **warp shuffle** functions introduced in CUDA 3.0 allow us to do this.

In CUDA a **warp** is a set of threads with consecutive ranks belonging to a thread block. The number of threads in a warp is currently 32, although Nvidia has stated that this could change. There is a variable initialized by the system that stores the size of a warp:

```
int warpSize
```

The threads in a warp operate in SIMD fashion. So threads in different warps can execute different statements with no penalty, while threads within the same warp must execute the same statement. When the threads within a warp attempt to execute different statements—e.g., they take different branches in an **if–else** statement—the threads are said to have **diverged**. When divergent threads finish executing different statements, and start executing the same statement, they are said to have **converged**.

The rank of a thread within a warp is called the thread's **lane**, and it can be computed using the formula

```
lane = threadIdx.x % warpSize;
```

The warp shuffle functions allow the threads in a warp to read from registers used by another thread in the same warp. Let's take a look at the one we'll use to implement a tree-structured sum of the values stored by the threads in a warp[10]:

```
__device__ float __shfl_down_sync(
    unsigned mask,   /* in */,
    float var,       /* in */,
    unsigned diff,   /* in */,
    int width = warpSize /* in */);
```

[10] Note that the syntax of the warp shuffles was changed in CUDA 9.0. So you may run across CUDA programs that use the older syntax.

NIPulse

The `mask` argument indicates which threads are participating in the call. A bit, representing the thread's lane, must be set for each participating thread to ensure that all of the threads in the call have converged—i.e., arrived at the call—before any thread begins executing the call to `__shfl_down_sync`. We'll ordinarily use all the threads in the warp. So we'll usually define

```
mask = 0xffffffff ;
```

Recall that `0x` denotes a hexadecimal (base 16) value and `0xf` is 15_10, which is 1111_2.[11] So this value of `mask` is 32 1's in binary, and it indicates that every thread in the warp participates in the call to `__shfl_down_sync`. If the thread with lane l calls `__shfl_down_sync`, then the value stored in `var` on the thread with

$$\text{lane} = l + \texttt{diff}$$

is returned on thread l. Since `diff` has type **unsigned**, it is ≥ 0. So the value that's returned is from a *higher*-ranked thread. Hence the name "shuffle *down*."

We'll only use `width` = `warpSize`, and since its default value is `warpSize`, we'll omit it from our calls.

There are several possible issues:

- What happens if thread l calls `__shfl_down_sync` but thread l + `diff` doesn't? In this case, the value returned by the call on thread l is *undefined*.
- What happens if thread l calls `__shfl_down_sync` but l + `diff` ≥ `warpSize`? In this case the call will return the value in `var` already stored on thread l.
- What happens if thread l calls `__shfl_down_sync`, and l + `diff` < `warpSize`, but l + `diff` > largest lane in the warp. In other words, because the thread block size is not a multiple of `warpSize`, the last warp in the block has fewer than `warpSize` threads. Say there are w threads in the last warp, where 0 < w < `warpSize`. Then if

  $$l + \texttt{diff} \geq w,$$

  the value returned by the call is also undefined.

  So to avoid undefined results, it's best if

[11] The subscripts indicate the base.

NVPulse

- All the threads in the warp call `__shfl_down_sync`, and
- All the warps have `warpSize` threads, or, equivalently, the thread block size (`blockDim.x`) is a multiple of `warpSize`.

6.11.4 Implementing tree-structured global sum with a warp shuffle

So we can implement a tree-structured global sum using the following code:

```
__device__ float Warp_sum ( float var ) { unsigned mask =
    0xffffffff ;

    for     ( int diff = warpSize / 2 ; diff >        0; diff = diff /2)
            var += __shfl_down_sync_sync ( mask , var , diff ) ;
    return var ;
}   /* Warp_sum */
```

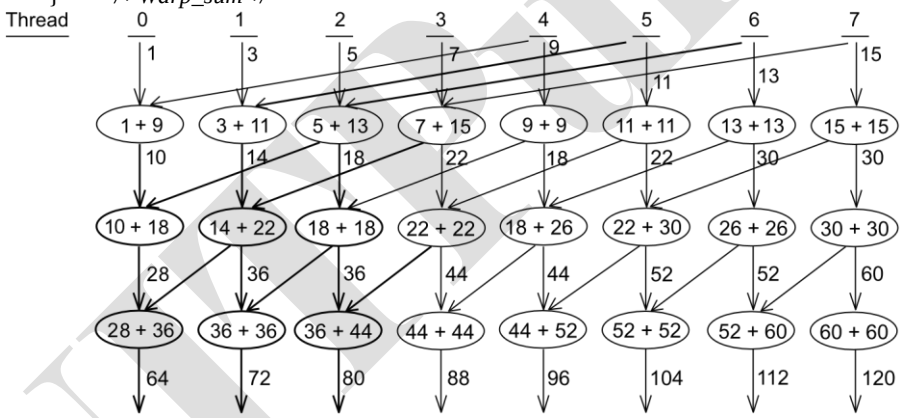

FIGURE 6.5

Tree-structured sum using warp shuffle.

Fig. 6.5 shows how the function would operate if `warpSize` were 8. (The diagram would be illegible if we used a `warpSize` of 32.) Perhaps the most confusing point in the behavior of `__shfl_down_sync` is that when the lane ID

$$l + \texttt{diff} \geq \texttt{warpSize},$$

the call returns the value in the *caller's* `var`. In the diagram this is shown by having only one arrow entering the oval with the sum, and it's labeled with the value just calculated by the thread carrying out the sum. In the row corresponding to `diff` = 4 (the first row of sums), the threads with lane IDs l = 4,5,6, and 7 all have l + 4 ≥ 8.

NVPulse

So the call to `__shfl_down_sync` returns their current `var` values, 9,11,13, and 15, respectively, and these values are doubled, because the return value of the call is added into the calling thread's variable `var`. Similar behavior occurs in the row corresponding to the sums for `diff` = 2 and lane IDs l = 6 and 7, and in the last row when `diff` = 1 for the thread with lane ID l = 7.

From a practical standpoint, it's important to remember that this implementation will only return the correct sum on the thread with lane ID 0. If all of the threads need the result, we can use an alternative warp shuffle function, `__shfl_xor`. See Exercise 6.6.

6.11.5 Shared memory and an alternative to the warp shuffle

If your GPU has compute capability < 3.0, you won't be able to use the warp shuffle functions in your code, and a thread won't be able to directly access the registers of other threads. However, your code *can* use shared memory, and threads in the same thread block can all access the same shared memory locations. In fact, although shared memory access is slower than register access, we'll see that the shared memory implementation can be just as fast as the warp shuffle implementation.

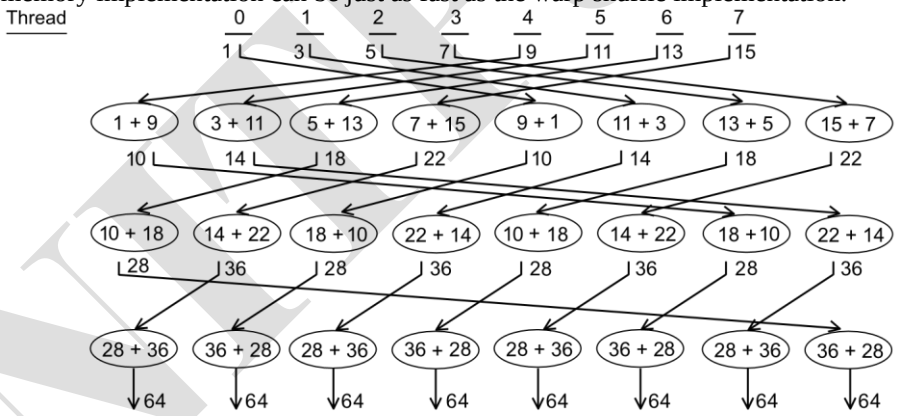

FIGURE 6.6

Dissemination sum using shared memory.

Since the threads belonging to a single warp operate synchronously, we can implement something very similar to a warp shuffle using shared memory instead of registers.

```
__device__ float Shared_mem_sum ( float shared_vals [ ] ) { int my_lane =
    threadIdx .x % warpSize ;
```

```
    for ( int diff = warpSize / 2 ; diff > 0; diff = diff /2) { /* Make sure 0 <=
        source < warpSize */ int source = ( my_lane + diff ) % warpSize ;
          shared_vals [ my_lane ]     += shared_vals [ source ];
    }
    return shared_vals [ my_lane ];
}
```

This should be called by all the threads in a warp, and the array `shared_vals` should be stored in the shared memory of the SM that's running the warp. Since the threads in the warp are operating in SIMD fashion, they effectively execute the code of the function in lockstep. So there's no race condition in the updates to `shared_vals`: all the threads read the values in `shared_vals[source]` before any thread updates `shared_vals[my_lane]`.

Technically speaking, this isn't a tree-structured sum. It's sometimes called a **dissemination sum** or **dissemination reduction**. Fig. 6.6 illustrates the copying and additions that take place. Unlike the earlier figures, this figure doesn't show the direct contributions that a thread makes to its sums: including these lines would have made the figure too difficult to read. Also note that every thread reads a value from another thread in each pass through the **for** statement. After all these values have been added in, every thread has the correct sum—not just thread 0. Although we won't need this for the trapezoidal rule, this can be useful in other applications. Furthermore, in any cycle in which the threads in a warp are working, each thread either executes the current instruction or it is idle. So the cost of having every thread execute the same instruction shouldn't be any greater than having some of the threads execute one instruction and the others idle.

An obvious question here is: how does `Shared_mem_sum` make use of Nvidia's shared memory? The answer is that it's not required to use shared memory. The function's argument, the array `shared_vals`, could reside in either global memory or shared memory. In either case, the function would return the sum of the elements of `shared_vals`.

However, to get the best performance, the argument `shared_vals` should be defined to be `__shared__` in a kernel. For example, if we know that `shared_vals` will need to store at most 32 floats in each thread block, we can add this definition to our kernel:

```
__shared__ float shared_vals [ 3 2 ];
```

For each thread block this sets aside storage for a collection of 32 floats in the shared memory of the SM to which the block is assigned.

Alternatively, if it isn't known at compile time how much shared memory is needed, it can be declared as **extern** `__shared__` **float** `shared_vals [ ];`

Nvidia

and when the kernel is called, a third argument can be included in the triple angle brackets specifying the size *in bytes* of the block of shared memory. For example, if we were using `Shared_mem_sum` in a trapezoidal rule program, we might call the kernel `Dev_trap` with

```
Dev_trap <<<blk_ct , th_per_blk , th_per_blk* sizeof ( float )>>>
   (... args to Dev_trap ...);
```

This would allocate storage for `th_per_blk` floats in the `shared_vals` array in each thread block.

6.12 Implementation of trapezoidal rule with `warpSize` thread blocks

Let's put together what we've learned about more efficient sums, warps, warp shuffles, and shared memory to create a couple of new implementations of the trapezoidal rule.

For both versions we'll assume that the thread blocks consist of `warpSize` threads, and we'll use one of our "tree-structured" sums to add the results of the threads in the warp. After computing the function values and adding the results within a warp, the thread with lane ID 0 in the warp will add the warp sum into the total using `Atomic_add`.

NIPulse

6.12.1 Host code

For both the warp shuffle and the shared memory versions, the host code is virtually identical to the code for our first CUDA version. The only substantive difference is that there is no `th_per_blk` variable in the new versions, since we're assuming that each thread block has `warpSize` threads.

6.12.2 Kernel with warp shuffle

Our kernel is shown in Program 6.13. Initialization of `my_trap` is the same as it was in our original implementation (Program 6.12). However, instead of adding each

```
__global__ void Dev_trap (
      const    float a   /*    in    */, const float b
       /*      in      */, const float h   /*      in
       */, const int      n      /*      in      */,
       float *            trap_p      /* in / out */)    {
   int my_i = blockDim.x * blockIdx.x + threadIdx.x;

   float my_trap =        0.0f; if (0 <
   my_i && my_i < n )     { float my_x
   = a + my_i*h;
      my_trap = f( my_x );
   }

   float result = Warp_sum( my_trap );

   /* result is correct only on thread 0 */ if ( threadIdx.x == 0)
atomicAdd ( trap_p , result ); } /* Dev_trap */
```

NTPulse

Program 6.13: CUDA kernel implementing trapezoidal rule and using `Warp_sum`.

thread's calculation directly into `*trap_p`, each warp (or, in this case, thread block) calls the `Warp_sum` function (Fig. 6.5) to add the values computed by the threads in the warp. Then, when the warp returns, thread (or lane) 0 adds the warp sum for its thread block (`result`) into the global total. Since, in general, this version will use multiple thread blocks, there will be multiple warp sums that need to be added to `*trap_p`. So if we didn't use `atomicAdd`, the addition of `result` to `*trap_p` would form a race condition.

6.12.3 Kernel with shared memory

The kernel that uses shared memory is shown in Program 6.14. It is almost identical

```
__global__ void Dev_trap (
      const float a  /* in */, const float b  /* in
      */, const float h  /* in */, const int n
      /* in */,
      float * trap_p  /* out */) {
   __shared__ float shared_vals [ WARPSZ ]; int my_i = blockDim . x *
   blockIdx . x + threadIdx . x ; int my_lane = threadIdx . x % warpSize
   ;

   shared_vals [ my_lane ] = 0.0 f ; if (0
   < my_i && my_i < n ) { float my_x = a
   + my_i*h ;
      shared_vals [ my_lane ] = f ( my_x );
   }

   float result = Shared_mem_sum ( shared_vals );

   /* r e s u l t i s the same on a l l threads in a block . */ if ( threadIdx . x ==
0) atomicAdd ( trap_p , result ); } /* Dev_trap */
```

to the version that uses the warp shuffle. The main differences are that it declares an

Program 6.14: CUDA kernel implementing trapezoidal rule and using shared memory.

array of shared memory in Line 7; it initializes this array in Lines 11 and 14; and, of course, the call to `Shared_mem_sum` is passed this array rather than a scalar register.

Since we know at compile time how much storage we'll need in `shared_vals`, we can define the array by simply preceding the ordinary C definition with the CUDA qualifier `__shared__`:

```
__shared__ float shared_vals [ WARPSZ ];
```

Note that the CUDA defined variable `warpSize` is *not* defined at compile-time. So our program defines a preprocessor macro

```
#define WARPSZ 32
```

6.12.4 Performance

Of course, we want to see how the various implementations perform. (See Table 6.8.) The problem is the same as the problem we ran earlier (see Table 6.5): we're integrating $f(x) = x^2 + 1$ on the interval $[-3,3]$, and there are $2^{20} = 1{,}048{,}576$ trapezoids. However, since the thread block size is 32, we're using 32,768 thread blocks ($32 \times 32{,}768 = 1{,}048{,}576$).

Table 6.8 Mean run-times for trapezoidal rule using block size of 32 threads (times in ms).

System	ARM Cortex-A15	Nvidia GK20A	Intel Core i7	Nvidia GeForce GTX Titan X
Clock	2.3 GHz	852 MHz	3.5 GHz	1.08 GHz
SMs, SPs		1, 192		24, 3072
Original	33.6	20.7	4.48	3.08
Warp Shuffle		14.4		0.210
Shared Memory		15.0		0.206

We see that on both systems and with both sum implementations, the new programs do significantly better than the original. For the GK20A, the warp shuffle version runs in about 70% of the time of the original, and the shared memory version runs in about 72% of the time of the original. For the Titan X, the improvements are much more impressive: both versions run in less than 7% of the time of the original. Perhaps most striking is the fact that on the Titan X, the warp shuffle is, on average, slightly slower than the shared memory version.

6.13 CUDA trapezoidal rule III: blocks with more than one warp

Limiting ourselves to thread blocks with only 32 threads reduces the power and flexibility of our CUDA programs. For example, devices with compute capability ≥ 2.0 can have blocks with as many as 1024 threads or 32 warps, and CUDA provides a fast barrier that can be used to synchronize *all* the threads in a block. So if we limited ourselves to only 32 threads in a block, we wouldn't be using one of the most useful features of CUDA: the ability to efficiently synchronize large numbers of threads.

So what would a "block" sum look like if we allowed ourselves to use blocks with up to 1024 threads? We could use one of our existing warp sums to add the values computed by the threads in each warp. Then we would have as many as 1024/32 = 32 warp sums, and we could use one warp in the thread block to add the warp sums.

Since two threads belong to the same warp if their ranks in the block have the same quotient when divided by `warpSize`, to add the warp sums, we can use warp 0, the threads with ranks 0, 1, ..., 31 in the block.

6.13.1 `__syncthreads`

We might try to use the following pseudocode for finding the sum of the values computed by all the threads in a block:

```
Each thread computes its contribution ;
Each warp adds its threads' contributions;
Warp 0 in block adds warp sums;
```

However, there's a race condition. Do you see it? When warp 0 tries to compute the total of the warp sums in the block, it doesn't know whether all the warps in the block have completed their sums. For example, suppose we have two warps, warp 0 and warp 1, each

of which has 32 threads. Recall that the threads in a warp operate in SIMD fashion: no thread in the warp proceeds to a new instruction until all the threads in the warp have completed (or skipped) the current instruction. But the threads in warp 0 can operate independently of the threads in warp 1. So if warp 0 finishes computing its sum before warp 1 computes its sum, warp 0 could try to add warp 1's sum to its sum before warp 1 has finished, and, in this case, the block sum could be incorrect.

So we must make sure that warp 0 doesn't start adding up the warp sums until all of the warps in the block are done. We can do this by using CUDA's fast barrier:

```
__device__ void __syncthreads ( void ) ;
```

This will cause the threads in the thread block to wait in the call until all of the threads have started the call. Using `__syncthreads`, we can modify our pseudocode so that the race condition is avoided:

```
Each thread computes its contribution ;
Each warp adds its threads' contributions;
__syncthreads();
Warp 0 in block adds warp sums;
```

Now warp 0 won't be able to add the warp sums until every warp in the block has completed its sum.

There are a couple of important caveats when we use `__syncthreads`. First, it's critical that *all* of the threads in the block execute the call. For example, if the block contains at least two threads, and our code includes something like this:

```
int my_x = threadIdx . x ; if ( my_x <
blockDim . x /2)
    __syncthreads ( ) ; my_x ++;
```

then only half the threads in the block will call `__syncthreads`, and these threads can't proceed until *all* the threads in the block have called `__syncthreads`. So they will wait forever for the other threads to call `__syncthreads`.

The second caveat is that `__syncthreads` only synchronizes the threads in a block. If a grid contains at least two blocks, and if all the threads in the grid call `__syncthreads` then the threads in different blocks will continue to operate independently of each other. So we can't synchronize the threads in a general grid with `__syncthreads`.[12]

6.13.2 More shared memory

If we try to implement the pseudocode in CUDA, we'll see that there's an important detail that the pseudocode doesn't show: after the call to `__syncthreads`, how does warp 0 obtain access to the sums computed by the other warps? It can't use a warp shuffle and registers: the warp shuffles only allow a thread to read a register belonging to another thread when

[12] CUDA 9 includes an API that allows programs to define barriers across more general collections of threads than thread blocks, but defining a barrier across multiple thread blocks requires hardware support that's not available in processors with compute capability < 6.

NVPulse

that thread belongs to the same warp, and, for the final warp sum we would like the threads in warp 0 to read registers belonging to threads in *other* warps.

You may have guessed that the solution is to use shared memory. If we use warp shuffles to compute the warp sums, we can just declare a shared array that can store up to 32 floats, and the thread with lane 0 in warp w can store its warp sum in element w of the array:

```
__shared__ float warp_sum_arr [ WARPSZ ]; int my_warp =
threadIdx.x / warpSize; int my_lane = threadIdx.x %
warpSize; // Threads calculate their contributions;
...
float my_result = Warp_sum(my_trap);
    if (my_lane == 0) warp_sum_arr[my_warp] = my_result;
__syncthreads();
    // Warp 0 adds the sums in warp_sum_arr
...
```

6.13.3 Shared memory warp sums

If we're using shared memory instead of warp shuffles to compute the warp sums, we'll need enough shared memory for each warp in a thread block. Since shared variables are shared by *all* the threads in a thread block, we need an array large enough to hold the contributions of all of the threads to the sum. So we can declare an array with 1024 elements—the largest possible block size—and partition it among the warps:

```
    // Make max thread block size available at compile time
#define MAX_BLKSZ 1024
...
__shared__ float thread_calcs[MAX_BLKSZ];
```

Now each warp will store its threads' calculations in a subarray of `thread_calcs`:

```
float* shared_vals = thread_calcs + my_warp*warpSize;
```

In this setting a thread stores its contribution in the subarray referred to by `shared_vals`:

```
    shared_vals[my_lane] = f(my_x);
```

Now each warp can compute the sum of its threads' contributions by using our shared memory implementation that uses blocks with 32 threads:

```
float my_result = Shared_mem_sum(shared_vals);
```

To continue we need to store the warp sums in locations that can be accessed by the threads in warp 0 in the block, and it might be tempting to try to make a subarray of `thread_calcs` do "double duty." For example, we might try to use the first 32 elements for both the contributions of the threads in warp 0, and the warp sums computed by the warps

NIPulse

in the block. So if we have a block with 32 warps of 32 threads, warp w might store its sum in `thread_calcs[w]` for w = 0,1,2,...,31.

The problem with this approach is that we'll get another race condition. When can the other warps safely overwrite the elements in warp 0's block? After a warp has completed its call to `Shared_mem_sum`, it would need to wait until warp 0 has finished its call to `Shared_mem_sum` before writing to `thread_calcs`:

```
float my_result = Shared_mem_sum ( shared_vals ) ;
__syncthreads ( ) ; if ( my_lane == 0) thread_calcs [ my_warp ] = my_result .
```

This is all well and good, but warp 0 still can't proceed with the final call to `Shared_mem_sum`: it must wait until all the warps have written to `thread_calcs`. So we would need a *second* call to `__syncthreads` before warp 0 could proceed:

```
if ( my_lane == 0) thread_calcs [ my_warp ] = my_result .
   __syncthreads ( ) ;
// I t ' s safe for warp 0 to proceed . . .
if ( my_warp == 0)
      my_result = Shared_mem_sum ( thread_calcs ) ;
```

Calls to `__syncthreads` are fast, but they're not free: every thread in the thread block will have to wait until all the threads in the block have called `__syncthreads`. So this can be costly. For example, if there are more threads in the block than there are SPs in an SM, the threads in the block won't all be able to execute simultaneously. So some threads will be delayed reaching the second call to `__syncthreads`, and all of the threads in the block will be delayed until the last thread is able to call `__syncthreads`.
So we should only call `__syncthreads`() when we have to.

Alternatively, each warp could store its warp sum in the "first" element of its subarray:

```
float my_result = Shared_mem_sum ( shared_vals ) ; if ( my_lane == 0)
shared_vals [0] = my_result ;
__syncthreads ( ) ;
. . .
```

It might at first appear that this would result in a race condition when the thread with lane 0 attempts to update `shared_vals`, but the update is OK. Can you explain why?

6.13.4 Shared memory banks

However, this implementation may not be as fast as possible. The reason has to do with details of the design of shared memory: Nvidia divides the shared memory on

Table 6.9 Shared memory banks: Columns are memory banks. The entries in the body of the table show subscripts of elements of `thread_calcs`.

	Bank					
Subscripts	0	1	2	...	30	31

NVIDIA

	0	1	2	. . .	30	31
	32	33	34	. . .	62	63
	64	65	66	. . .	94	95
	96	97	98	. . .	126	127
	.	.	.		.	.
	.	.	.	...	.	.
	.	.	.		.	.
	992	993	994	. . .	1022	1023

an SM into 32 "banks" (16 for GPUs with compute capability < 2.0). This is done so that the 32 threads in a warp can simultaneously access shared memory: the threads in a warp can simultaneously access shared memory when each thread accesses a different bank.

Table 6.9 illustrates the organization of `thread_calcs`. In the table, the columns are banks, and the rows show the subscripts of consecutive elements of `thread_calcs`. So the 32 threads in a warp can simultaneously access the 32 elements in any one of the rows, or, more generally, if each thread access is to a different column.

When two or more threads access different elements in a single bank (or column in the table), then those accesses must be serialized. So the problem with our approach to saving the warp sums in elements 0, 32, 64, ..., 992 is that these are all in the same bank. So when we try to execute them, the GPU will serialize access, e.g., element 0 will be written, then element 32, then element 64, etc. So the writes will take something like 32 times as long as it would if the 32 elements were stored in different banks, e.g., a row of the table.

The details of bank access are a little complicated and some of the details depend on the compute capability, but the main points are

- If each thread in a warp accesses a different bank, the accesses can happen simultaneously.
- If multiple threads access different memory locations in a single bank, the accesses must be serialized.
- If multiple threads read the same memory location in a bank, the value read is broadcast to the reading threads, and the reads are simultaneous.

The CUDA programming Guide [11] provides full details.

Thus we could exploit the use of the shared memory banks if we stored the results in a contiguous subarray of shared memory. Since each thread block can use at least 16 Kbytes of shared memory, and our "current" definition of `shared_vals` only uses at most 1024 floats or 4 Kbytes of shared memory, there is plenty of shared memory available for storing 32 more floats.

So if we're using shared memory warp sums, a simple solution is to declare *two* arrays of shared memory: one for storing the computations made by each thread, and another for storing the warp sums.

```
__shared__ float thread_calcs [ MAX_BLKSZ ]; __shared__ float warp_sum_arr [
WARPSZ ]; float * shared_vals = thread_calcs + my_warp*warpSize ;
```

NIH Use

```
...
float my_result = Shared_mem_sum ( shared_vals ) ; if ( my_lane == 0)
warp_sum_arr [ my_warp ] = my_result ;
__syncthreads ( ) ;
...
```

6.13.5 Finishing up

The remaining codes for the warp sum kernel and the shared memory sum kernel are very similar. First warp 0 computes the sum of the elements in `warp_sum_arr`. Then thread 0 in the block adds the block sum into the total across all the threads in the grid using `atomicAdd`. Here's the code for the shared memory sum:

```
if ( my_warp == 0) { if ( threadIdx . x >= blockDim . x / warpSize
    )
      warp_sum_arr [ threadIdx . x ]    =   0.0;
    blk_result = Shared_mem_sum ( warp_sum_arr ) ;
}

if       ( threadIdx . x == 0) atomicAdd ( trap_p , blk_result ) ;
```

In the test `threadIdx.x > blockDim.x/warpSize` we're checking to see if there are fewer than 32 warps in the block. If there are, then the final elements in `warp_sum_arr` won't have been initialized. For example, if there are 256 warps in the block, then

```
blockDim . x / warpSize = 256/32 = 8
```

So there are only 8 warps in a block and we'll have only initialized elements 0,1,...,7 of `warp_sum_arr`. But the warp sum function expects 32 values. So for the threads with `threadIdx.x >= 8`, we assign `warp_sum_arr [ threadIdx . x ] = 0.0;`

For the sake of completeness, Program 6.15 shows the kernel that uses shared memory. The main differences between this kernel and the kernel that uses warp shuffles are that the declaration of the first shared array isn't needed in the warp shuffle version, and, of course, the warp shuffle version calls `Warp_sum` instead of `Shared_mem_sum`.

6.13.6 Performance

Before moving on, let's take a final look at the run-times for our various versions of the trapezoidal rule. (See Table 6.10.) The problem is the same: find the area under

```
__global__ void Dev_trap (
        const float a  /* in */, const float b  /*
        in */, const float h  /* in */, const int
         n        /* in */,
         float *          trap_p        /* out */)    {
    __shared__ float thread_calcs [ MAX_BLKSZ ]; __shared__ float
    warp_sum_arr [ WARPSZ ]; int my_i = blockDim . x * blockIdx . x +
    threadIdx . x ; int my_warp = threadIdx . x / warpSize ; int my_lane
    = threadIdx . x % warpSize ;
    float * shared_vals = thread_calcs + my_warp*warpSize ; float blk_result
    = 0.0;

    shared_vals [ my_lane ] = 0.0 f ; if (0
    < my_i && my_i < n ) { float my_x = a
    + my_i*h ;
         shared_vals [ my_lane ]   = f ( my_x );
    }

    float my_result = Shared_mem_sum ( shared_vals ); if ( my_lane == 0)
    warp_sum_arr [ my_warp ] = my_result ;
    __syncthreads ( );

    if ( my_warp == 0) { if ( threadIdx . x >= blockDim . x /
        warpSize ) warp_sum_arr [ threadIdx . x ] = 0.0;
        blk_result = Shared_mem_sum ( warp_sum_arr );
    }

    if ( threadIdx . x == 0) atomicAdd ( trap_p , blk_result ); } /* Dev_trap
*/
```

NTPulse

Program 6.15: CUDA kernel implementing trapezoidal rule and using shared memory. This version can use large thread blocks.

the graph of $y = x^2+1$ between $x = -3$ and $x = 3$ using $2^{20} = 1{,}048{,}576$ trapezoids.

However, instead of using a block size of 32 threads, this version uses a block size of 1024 threads. The larger blocks provide a significant advantage on the GK20A: the warp shuffle version is more than 10% faster than the version that uses 32 threads per block, and the shared memory version is about 5% faster. On the Titan X, the performance improvement is huge: the warp shuffle version is more than 30% faster, and the shared memory version is more than 25% faster. So on the faster GPU, reducing the number of threads calling `atomicAdd` was well worth the additional programming effort.

NVPulse

Table 6.10 Mean run-times for trapezoidal rule using arbitrary block size (times in ms).

System	ARM Cortex-A15	Nvidia GK20A	Intel Core i7	Nvidia GeForce GTX Titan X
Clock	2.3 GHz	852 MHz	3.5 GHz	1.08 GHz
SMs, SPs		1, 192		24, 3072
Original	33.6	20.7	4.48	3.08
Warp Shuffle, 32 ths/blk		14.4		0.210
Shared Memory, 32 ths/blk		15.0		0.206
Warp Shuffle		12.8		0.141
Shared Memory		14.3		0.150

NITPulse