MODULE 5

GPU programming with CUDA: GPUs and GPGPU, GPU architectures, Heterogeneous computing, Threads, blocks, and grids Nvidia compute capabilities and device architectures, Vector addition, returning results from CUDA kernels, CUDA trapezoidal rule I, CUDA trapezoidal rule II: improving performance, CUDA trapezoidal rule III: blocks with more than one warp

5.1 GPUs and GPGPU

In the late 1990s and early 2000s, the demand for highly realistic video games and animations drove the development of **powerful graphics processing units (GPUs)**. These processors were specifically designed to accelerate programs that rendered complex and detailed images. As the computational capabilities of GPUs became evident, programmers beyond the field of computer graphics began exploring their use for **general computational problems**, such as searching, sorting, and numerical simulations. This shift gave rise to **General-Purpose computing on GPUs (GPGPU)**.

Early attempts at GPGPU faced significant challenges because GPUs could only be programmed using **graphics APIs** like Direct3D and OpenGL. Programmers had to reformulate conventional algorithms in terms of graphics concepts such as **vertices, triangles, and pixels**, making development cumbersome and complex. To overcome this, researchers developed **languages and compilers** that allowed general algorithms to be implemented on GPUs using constructs similar to high-level CPU programming.

These developments eventually led to widely used APIs for general-purpose GPU programming. **CUDA**, developed by Nvidia, is optimized for Nvidia GPUs and requires relatively modest setup. **OpenCL**, in contrast, was designed for **portability**, allowing programs to run on a variety of devices, including GPUs, **FPGAs**, and **digital signal processors (DSPs)**, but it demands more detailed setup to handle different systems. Due to its simplicity for Nvidia GPUs, CUDA is often preferred in practice.

This evolution illustrates how GPUs, originally tailored for graphics, have become powerful tools for accelerating a broad range of **parallel computations**, forming a crucial part of modern high-performance computing.

5.1 GPUs architectures

A **CPU** is generally a **SISD (Single Instruction, Single Data)** device in **Flynn's taxonomy**, where each instruction operates on a small set of data. In contrast, **GPUs** are **SIMD (Single Instruction, Multiple Data)** processors that apply one instruction to many data elements simultaneously, making them highly efficient for parallel workloads.

A **SIMD processor** consists of a single **control unit** and multiple **datapaths**. The control unit broadcasts an instruction, and each datapath executes it on its own data or remains idle. For instance, in an array *x*, if we wish to add 1 to nonnegative elements and subtract 2 from negative ones, each datapath tests x[i] >= 0 and executes accordingly. During this process, some datapaths remain idle while others execute, as shown in **Table 6.1**.

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`

Modern **GPUs** consist of several **Streaming Multiprocessors (SMs)**, each containing many **Streaming Processors (SPs)—**the actual datapaths. SMs operate **asynchronously**, allowing different branches of code to execute on different SMs without penalty, reducing execution stages as shown in **Table 6.2**. Nvidia GPUs, for instance, can have up to **82 SMs** with **128 SPs** each, totaling over **10,000 SPs**. Nvidia refers to this model as **SIMT (Single Instruction, Multiple Thread)**, since threads may execute at different times to hide **memory latency**.

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`

Each SM includes a small **shared memory** accessible quickly by its SPs, while all SMs share a larger, slower **global memory** (see **Figure 6.1**).

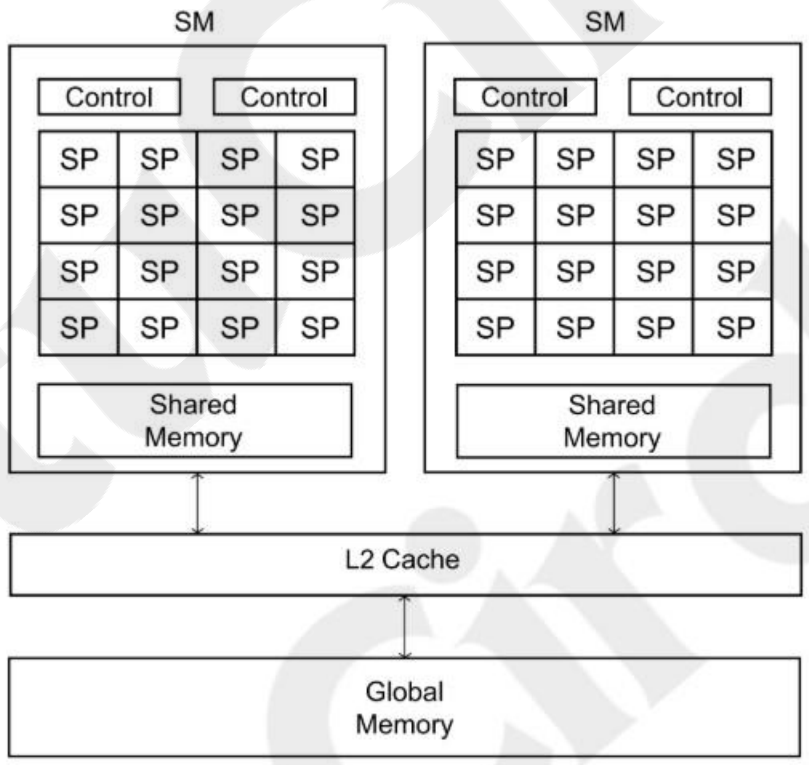


FIGURE 6.1

Simplified block diagram of a GPU.

The **CPU and GPU** have separate memories—the **host** and **device**, respectively. Earlier systems required explicit data transfers between them, but newer Nvidia architectures (with **compute capability ≥ 3.0**) support **unified memory**, eliminating the need for explicit transfers while retaining performance benefits (see **Figure 6.2**).

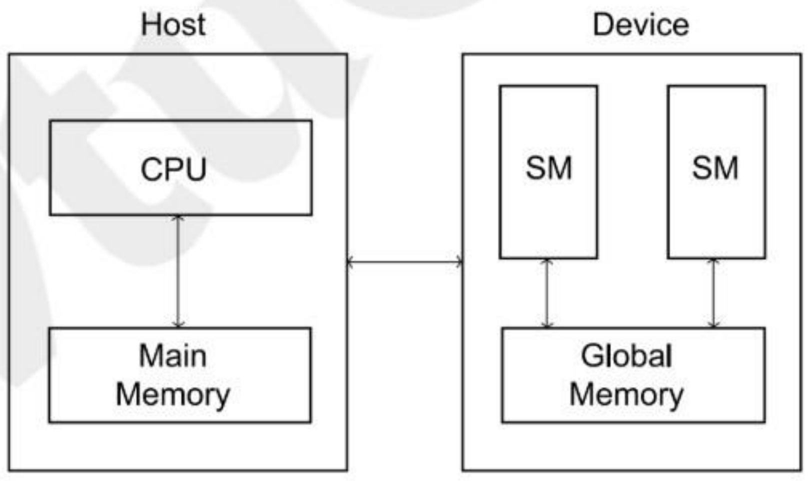


FIGURE 6.2

Simplified block diagram of a CPU and a GPU.

Thus, GPU architecture achieves massive **parallelism** and **high throughput** by combining SIMT execution, fast shared memory, and asynchronous multiprocessors, making it central to modern parallel computing.

5.3 Heterogeneous computing

Heterogeneous computing refers to systems that use processors with **different architectures** working together to execute a program. When we write programs that utilize both a **CPU (host processor)** and a **GPU (device processor)**, we are engaging in heterogeneous computing. Though we write a single program following the **SPMD (Single Program, Multiple Data)** model, parts of the program are designed for the CPU and other parts for the GPU, effectively creating two cooperating programs within one.

This approach has gained significant importance in recent years. Between 1986 and 2003, **CPU single-thread performance** grew by over 50% annually. However, after 2003, this growth slowed dramatically—to less than **4% per year** between 2015 and 2017. As a result, programmers began exploring other hardware to achieve better performance. **GPUs**, with their massive parallelism, are one major solution, but other specialized processors also play a role.

Field Programmable Gate Arrays (FPGAs) offer **programmable logic blocks and interconnects**, allowing reconfiguration before program execution to suit specific tasks. **Digital Signal Processors (DSPs)**, on the other hand, are designed with **special circuitry** for processing and transforming signals, such as compression or filtering of **real-world analog data**.

Thus, heterogeneous computing represents a powerful paradigm in modern systems, combining different processor types—each with its strengths—to achieve **higher performance and efficiency** than homogeneous CPU-only architectures.

5.4 CUDA hello

CUDA (Compute Unified Device Architecture) is a **software platform developed by NVIDIA** for programming heterogeneous systems that include both **CPU (host)** and **GPU (device)**. It enables developers to write **GPGPU (General-Purpose computing on Graphics Processing Units)** applications, allowing the GPU to be used for computations beyond traditional graphics processing.

Although CUDA originally stood for *Compute Unified Device Architecture*, NVIDIA now treats it simply as the name of their **API for GPU programming**.

CUDA provides **language-specific APIs** for several languages such as **C, C++, Fortran, Python, and Java.** In most engineering applications, we use **CUDA C**, which is an extension of the C programming language. However, since the CUDA compiler is a **modified C++ compiler**, certain C++ constructs may appear even in CUDA C programs. For instance, C requires no cast when using malloc(), while C++ does:

c

```c
// C version
float *x = malloc(n * sizeof(float));

// C++ version
float *x = (float *) malloc(n * sizeof(float));
```

Host and Device Programs

A CUDA program is **heterogeneous**, meaning it contains:

- A **Host (CPU)** part – executed on the main processor.
- A **Device (GPU)** part – executed on the GPU cores.

When we write a CUDA program, we are effectively writing **two interacting programs**: one running on the CPU and one on the GPU. The CPU launches and manages tasks on the GPU using CUDA API calls.

Compilation

CUDA programs **cannot be compiled with a regular C compiler (like gcc)**. This is because the code must be compiled into **machine instructions for two architectures—**one for the CPU and another for the GPU.

To do this, we use **nvcc**, the NVIDIA CUDA compiler, which separates host and device code and compiles them appropriately.

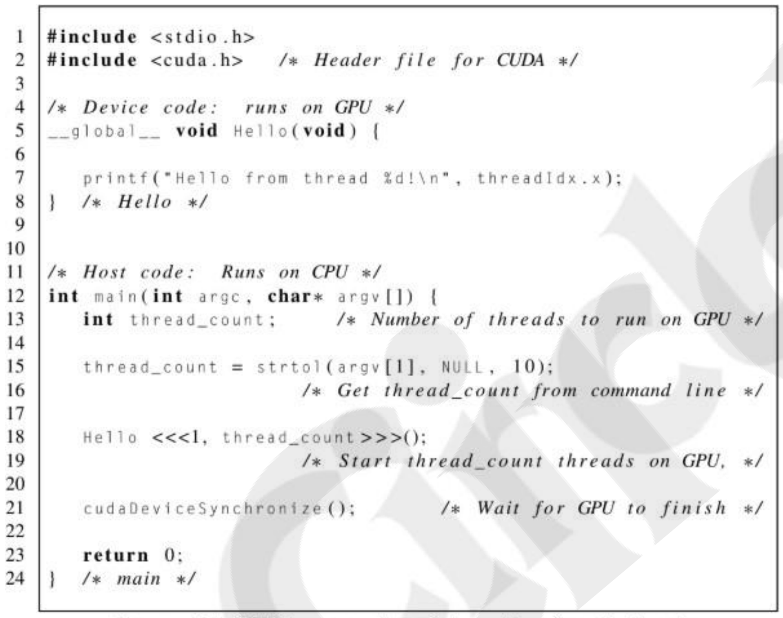
Example: CUDA "Hello, World" Program

The simplest example in CUDA is similar in spirit to the classic "hello, world" program, except that **each CUDA thread prints a greeting** from the GPU. This demonstrates that GPU threads can execute code independently.

5.4.1 The source code

The CUDA program begins with including the **CUDA header file.** The **Hello** function, defined with the keyword **__global__**, is a **kernel—**a function started by the host (CPU) but executed by each thread on the **device (GPU).** All CUDA kernels have a **void return type**.

The **main** function, like in ordinary C programs, runs on the host. It reads the number of threads from the command line and launches that many copies of the kernel using the syntax Hello<<<1, n>>>();. The call to **cudaDeviceSynchronize()** ensures that the CPU waits until all GPU threads have completed execution. Once all threads finish, the program ends normally with **return 0**.



```
#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.

5.4.2 Compiling and running the program

A CUDA source file containing both host and device code must use the **".cu"** extension. For example, the hello program is saved as **cuda_hello.cu**. It is compiled using the **NVIDIA CUDA compiler (nvcc)**, which generates code for both CPU and GPU.

To compile:

ruby

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

To run the program with **one GPU thread**:

shell

```
$ ./cuda_hello 1
```

Output:

cpp

```
Hello from thread 0!
```

To run the program with **ten GPU threads**:

```shell
$ ./cuda_hello 10
```

Output:

```
Hello from thread 0!
Hello from thread 1!
Hello from thread 2!
...
Hello from thread 9!
```

Each line is printed by a **different GPU thread**, showing that multiple threads can execute the same kernel in parallel.

5.6 Threads, Blocks, and Grids in CUDA

In CUDA, **threads** are the smallest units of execution, and they run on the **Streaming Processors (SPs)** within a GPU's **Streaming Multiprocessors (SMs)**.

When we launch a kernel like

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

the numbers inside the angle brackets specify how the work is divided. The **first value (1)** represents the **number of thread blocks**, and the **second (thread_count)** represents the **number of threads per block**.

An NVIDIA GPU consists of several **Streaming Multiprocessors (SMs)**, and each SM contains many **Streaming Processors (SPs)**. Every CUDA **thread** executes on one SP (

If the kernel is launched as

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

and thread_count is even, CUDA creates two thread blocks, each with thread_count/2 threads. These blocks may be assigned to two different SMs.

CUDA organizes computation in a **hierarchical structure**

- **Threads** are grouped into **Blocks**, which execute on a single SM.
- **Blocks** are grouped into a **Grid**, which represents all threads launched by one kernel call.

Each thread can determine its position using predefined variables:

- threadIdx – thread's index within its block.
- blockIdx – block's index within the grid.
- blockDim – dimensions (number of threads) in a block.

- gridDim – dimensions (number of blocks) in the grid.

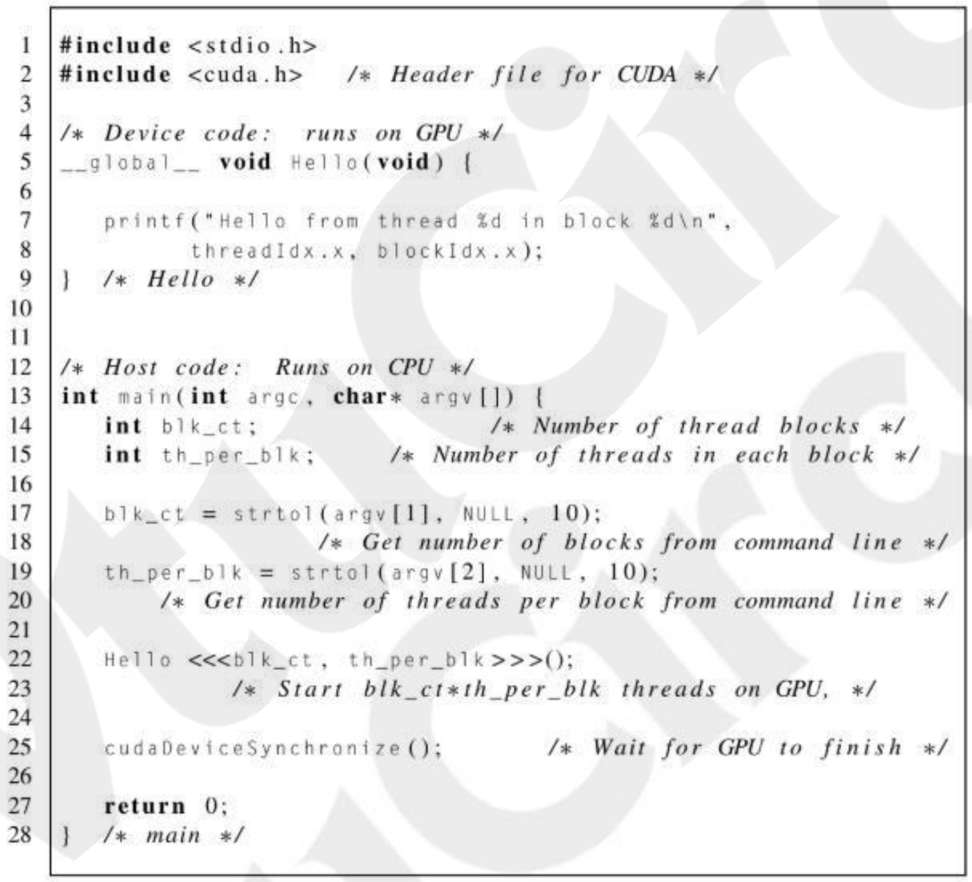
These are 3D structures (x, y, and z) that allow mapping of threads to multi-dimensional data. For instance, in image or matrix applications, threadIdx.x and threadIdx.y can represent a pixel's row and column.

Example of a 3D grid and block configuration:

```
dim3 grid_dims(2, 3, 1);
dim3 block_dims(4, 4, 4);
Kernel<<<grid_dims, block_dims>>>();
```

This creates a grid of **2 × 3 × 1 = 6 blocks**, each with **4 × 4 × 4 = 64 threads**.

All thread blocks must have **identical dimensions** and be **independent**, meaning any block can finish execution regardless of others. This independence allows the GPU to schedule blocks in **any order or in parallel**, ensuring maximum flexibility and performance.



```
#include <stdio.h>
#include <cuda.h>   /* Header file 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 line */
   th_per_blk = strtol(argv[2], NULL, 10);
       /* Get number of threads per block from command line */

   Hello <<<blk_ct, th_per_blk>>>();
            /* Start blk_ct*th_per_blk threads on GPU, */

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

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

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

5.7 NVIDIA Compute Capabilities and Device Architectures

Each NVIDIA GPU has specific hardware limits that determine how many **threads**, **blocks**, and **resources** it can handle. These limits are defined by what NVIDIA calls the **Compute Capability** of the GPU.

The compute capability is written as a **version number a.b**, where

- **a** = major revision number (possible values: 1, 2, 3, 5, 6, 7, 8)
- **b** = minor revision number (ranges from 0–7 depending on the major version)

Devices with compute capability **below 3.0 are no longer supported** by CUDA.

For GPUs with compute capability **greater than 1.x**:

- Maximum threads per block = **1024**
- For compute capability **2.b**, a single **Streaming Multiprocessor (SM)** can handle up to **1536 threads**
- For compute capability **greater than 2.x**, a single SM can manage up to **2048 threads**

There are also **dimension limits** for thread blocks and grids:

- For compute capability > 1:
  - Maximum block dimension in **x or y** = **1024**
  - Maximum block dimension in **z** = **64**

(Refer to *Table 6.3* for a list of NVIDIA GPU architectures and their corresponding compute capabilities.)

NVIDIA assigns **architecture names** (such as *Fermi, Kepler, Maxwell, Pascal, Volta, Turing, Ampere,* and *Ada Lovelace*) to its GPU microarchitectures. Note that *Tesla* is both a **GPU architecture name** and the **brand name** for NVIDIA's GPGPU product line, which can include standalone GPUs or **system-on-chip** devices.

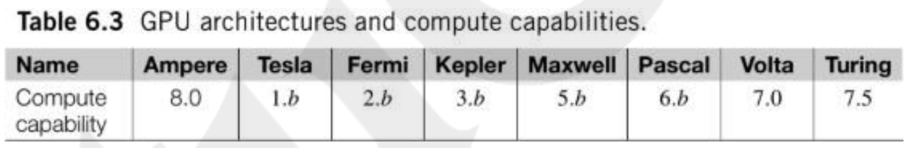


Table 6.3 GPU architectures and compute capabilities.

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

Lastly, **CUDA API versions** evolve independently of **compute capabilities**, meaning a newer CUDA Toolkit can still support older GPUs, provided they meet the minimum compute capability requirement.

5.8 Vector addition

GPUs and CUDA are especially efficient when executing **data-parallel programs**, where the same operation is applied to many data elements simultaneously. A classic example is **vector addition**, which is also **embarrassingly parallel**—each computation is independent of the others.

We define three arrays of equal size n:

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

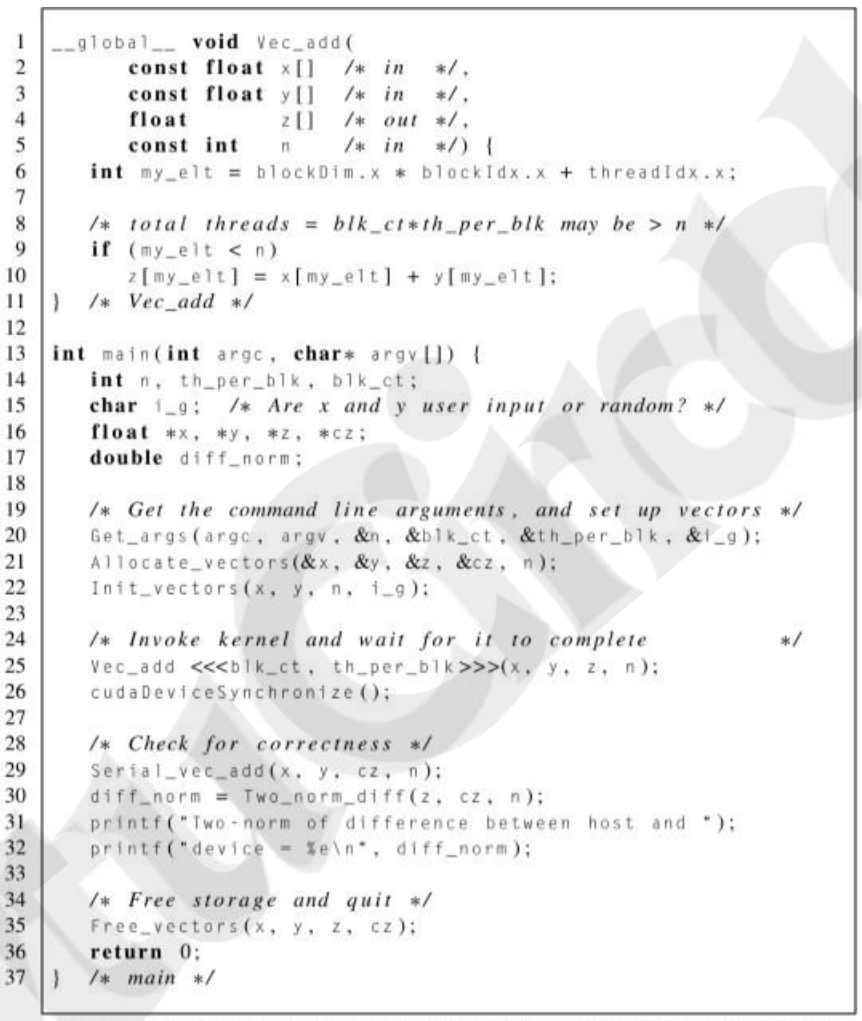
The arrays x and y are initialized on the **host (CPU)**, and the **device (GPU)** performs the addition:

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

Each GPU thread handles one element of the arrays, so a kernel will launch **at least n threads**, with thread i computing the i-th element of the result.

Since most GPUs are optimized for **32-bit (float)** operations rather than **64-bit (double)**, single-precision floats are preferred for performance.

After the kernel finishes execution, the host program checks the results, frees allocated memory, and exits.



```
 1  __global__ void Vec_add(
 2        const float x[]   /* in  */,
 3        const float y[]   /* in  */,
 4        float       z[]   /* out */,
 5        const int   n     /* in  */) {
 6     int my_elt = blockDim.x * blockIdx.x + threadIdx.x;
 7
 8     /* total threads = blk_ct*th_per_blk may be > n */
 9     if (my_elt < n)
10        z[my_elt] = x[my_elt] + y[my_elt];
11  }  /* Vec_add */
12
13  int main(int argc, char* argv[]) {
14     int n, th_per_blk, blk_ct;
15     char i_g;   /* Are x and y user input or random? */
16     float *x, *y, *z, *cz;
17     double diff_norm;
18
19     /* Get the command line arguments, and set up vectors */
20     Get_args(argc, argv, &n, &blk_ct, &th_per_blk, &i_g);
21     Allocate_vectors(&x, &y, &z, &cz, n);
22     Init_vectors(x, y, n, i_g);
23
24     /* Invoke kernel and wait for it to complete           */
25     Vec_add <<<blk_ct, th_per_blk>>>(x, y, z, n);
26     cudaDeviceSynchronize();
27
28     /* Check for correctness */
29     Serial_vec_add(x, y, cz, n);
30     diff_norm = Two_norm_diff(z, cz, n);
31     printf("Two-norm of difference between host and ");
32     printf("device = %e\n", diff_norm);
33
34     /* Free storage and quit */
35     Free_vectors(x, y, z, cz);
36     return 0;
37  }  /* main */
```

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

5.8.1 The Kernel – Vector Addition in CUDA

In the **kernel function** (*Lines 1–11* of *Program 6.3*), each thread is responsible for computing one element of the output vector z. To do this, we must determine which element each thread should process.

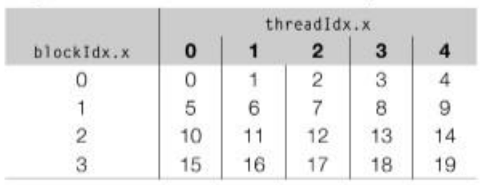
CUDA threads are organized hierarchically into **blocks** and **grids**, so each thread's **global index (rank)** can be computed as:

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

This gives every thread a unique index across the entire grid.

For example, if there are **four blocks** with **five threads per block**, the total number of threads is 4 × 5 = 20, and the **global ranks** range from 0 to 19 (see *Table 6.4*).

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

In the kernel, this index is stored in a variable such as my_elt, which is then used to access the elements of the arrays:

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

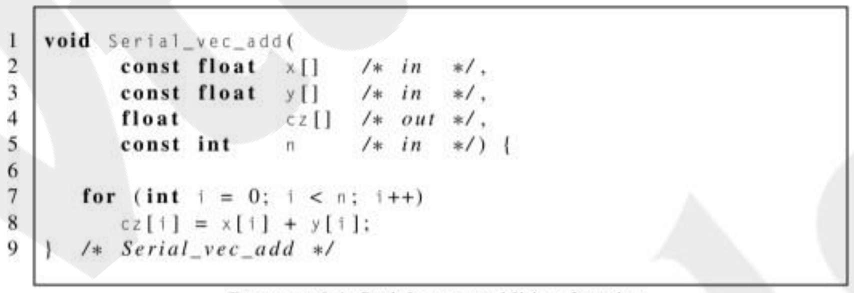
Before performing the addition, the program checks:

```
if (my_elt < n)
    z[my_elt] = x[my_elt] + y[my_elt];
```

This ensures that no thread tries to access elements beyond the array bounds when the total number of threads exceeds n.

For instance, if n = 997, we cannot evenly divide threads across blocks (since 997 is prime). We might choose **4 blocks of 256 threads** (total = 1024 threads), meaning the extra 27 threads simply skip the addition.

If CUDA support isn't available, the same logic can be implemented using a **serial vector addition** (as shown in *Program 6.4*).



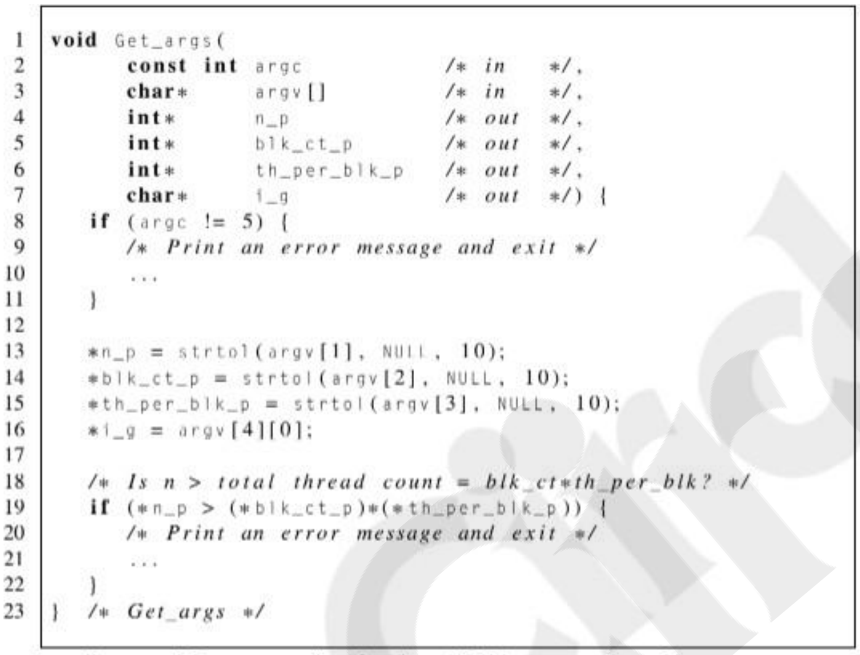
```
void Serial_vec_add(
      const float  x[]   /* in  */,
      const float  y[]   /* in  */,
      float        cz[]  /* out */,
      const int    n     /* in  */) {

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

Program 6.4: Serial vector addition function.

Thus, a **CUDA kernel** can be viewed as a **parallel version of a serial loop**, where each loop iteration is assigned to a different thread. Following **Foster's design methodology**, this program's tasks are independent additions—no communication or aggregation is needed, only a direct mapping of elements to threads.

5.8.2 Get_args

In the main function, after declaring variables, the **Get_args** function is invoked to retrieve four parameters: **n** (the array size), **blk_ct** (number of thread blocks), **th_per_blk** (threads per block), and **i_g** (a flag indicating input mode). These values are obtained from the command line. If the arguments are missing or invalid, the function displays a usage message and stops execution. It also verifies that **n** does not exceed the total available threads; otherwise, it terminates with an error. When **i_g** specifies user input, arrays **x** and **y** are entered manually; otherwise, they are populated with random values. The **Get_args** function is written in standard C and executes entirely on the **host**. *(See Program 6.5.)*



```
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 */
```

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

5.8.3 Allocate_vectors and managed memory

After obtaining the command-line parameters, the **main** function invokes **Allocate_vectors** to allocate four float arrays of size *n*: **x**, **y**, **z**, and **cz**. Among these, **x**, **y**, and **z** are used both on the host and device, while **cz** is used only on the host to compute the vector sum sequentially for verification of the device's result. *(See Program 6.6.)*

Since **cz** is host-only, its memory is allocated using the standard C library function **malloc()**. For **x**, **y**, and **z**, allocation is performed using the CUDA function **cudaMallocManaged()**, which provides *unified memory* accessible by both host and device:

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

The _host__qualifier indicates that the function executes on the host. The function returns a value of type **cudaError_t**, useful for error checking, though often omitted for brevity. The first argument is a pointer to the pointer being allocated, the second specifies the size in bytes, and the optional flags parameter controls kernel access (default: cudaMemAttachGlobal).

cudaMallocManaged() simplifies programming by providing a unified memory model where host and device share access to the same allocation, though it has certain requirements and trade-offs:

1. Requires devices with **compute capability ≥ 3.0** and a **64-bit host OS**.
2. For devices with **compute capability < 6.0**, unified memory cannot be accessed simultaneously by host and device.
3. Unified memory may be slower than explicit data transfers due to system-managed synchronization.

When using unified memory, the system automatically decides when to move data between host and device. While this eases programming, explicit memory management (via **cudaMalloc**, **cudaMemcpy**, etc.) can offer better performance through optimized or overlapped transfers.

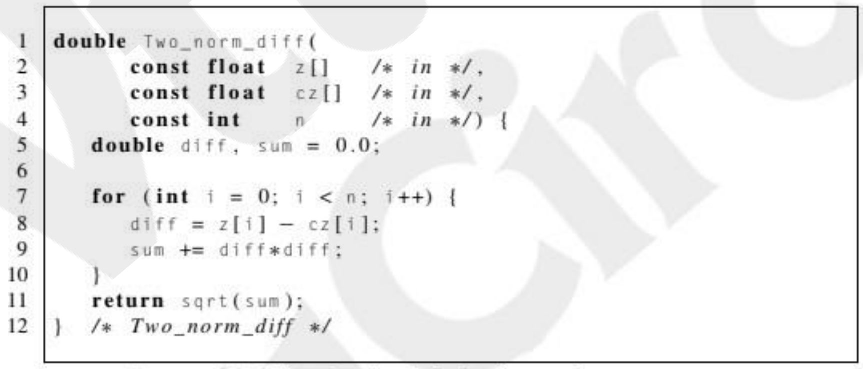
5.8.4 Other functions called from main

Except for **Free_vectors**, all other host functions invoked from **main** are written in standard C. The **Init_vectors** function initializes arrays **x** and **y**—either by reading values from standard input using **scanf** or by generating them randomly using the **random()** library function, based on the command-line argument **i_g**.

The **Serial_vec_add** function (refer to *Program 6.4*) performs vector addition on the host using a simple for loop and stores the result in the array **cz.**

The **Two_norm_diff** function calculates the Euclidean distance between the vectors **z** (computed on the device) and **cz** (computed serially on the host). It computes the square of the difference for each corresponding pair of elements, sums them, and then takes the square root:

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



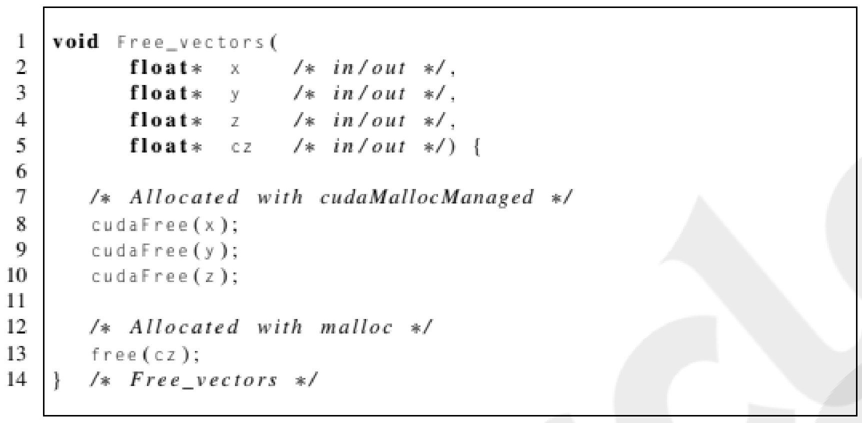
```
double Two_norm_diff(
      const float  z[]    /* in */,
      const float  cz[]   /* in */,
      const int    n      /* in */) {
   double diff, sum = 0.0;

   for (int i = 0; i < n; i++) {
      diff = z[i] - cz[i];
      sum += diff*diff;
   }
   return sqrt(sum);
}  /* Two_norm_diff */
```

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

The **Free_vectors** function releases all dynamically allocated memory. The host-only array **cz** is freed using the standard **free()** function, while the arrays **x**, **y**, and **z**, allocated via **cudaMallocManaged**, are freed using **cudaFree()**:

The _device_qualifier means this function can be invoked from either host or device code, though a pointer allocated on the device cannot be freed on the host and vice versa.

It is crucial to explicitly free device memory within the program; otherwise, it remains allocated until program termination. This becomes important in applications invoking multiple kernels—if memory from earlier kernels isn't released, it stays occupied even if later kernels don't use it.
(Refer to Program 6.8.)



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

   /*  Allocated  with  cudaMallocManaged  */
   cudaFree(x);
   cudaFree(y);
   cudaFree(z);

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

Program 6.8: CUDA function that frees four arrays.

5.8.5 Explicit memory transfers

In systems **without unified memory**, the vector addition program must explicitly handle memory transfers between the host and the device. The **kernel** itself remains unchanged—it receives arrays **x**, **y**, and **z**, along with **n**, computes the global thread index **my_elt**, and performs the addition if **my_elt < n**.

However, in the **main function**, since host and device pointers are distinct and not interchangeable, separate memory allocations are required. Thus, six arrays are declared—**hx**, **hy**, **hz** for host memory, and **dx**, **dy**, **dz** for device memory (Lines 15–16). Allocation is handled in the **Allocate_vectors** function (see *Program 6.10*). Instead of using **cudaMallocManaged**, the host arrays use the C **malloc()** function, while the device arrays use the CUDA **cudaMalloc()** function:

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

Here, **dev_p** points to device memory, and **size** specifies the number of bytes to allocate.

Once the host arrays **hx** and **hy** are initialized, their data is transferred to device memory **dx** and **dy** using **cudaMemcpy()** (Lines 24–26):

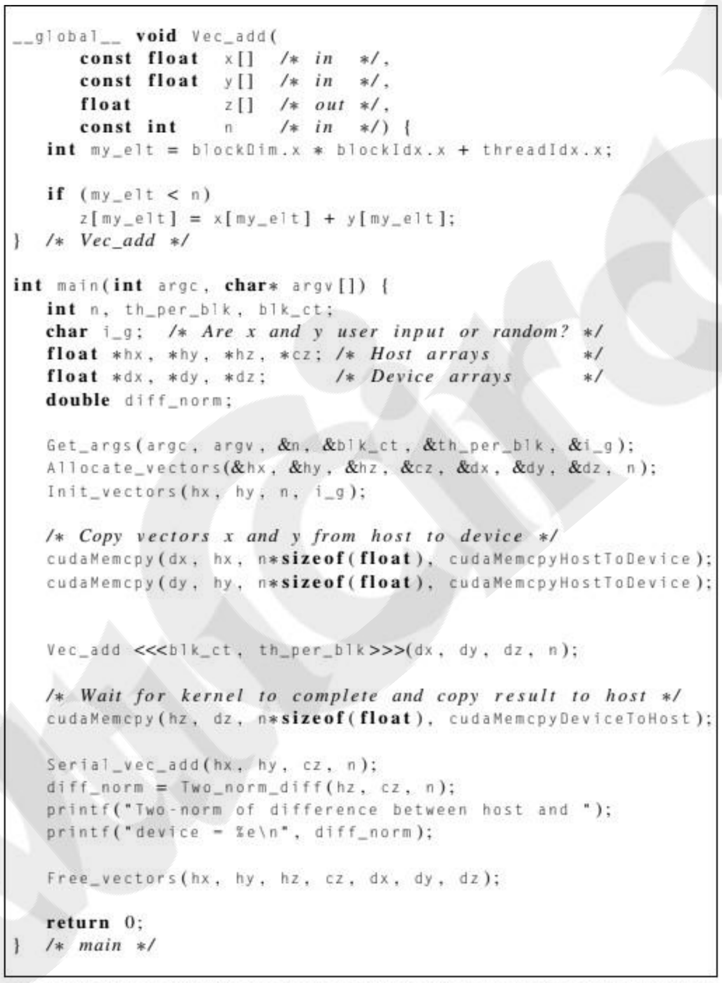
```
__host___ cudaError_t  cudaMemcpy(void *dest, const void *source, size_t  count,
cudaMemcpyKind kind);
```

The **kind** argument specifies the transfer direction:

- **cudaMemcpyHostToDevice** → copy from host to device
- **cudaMemcpyDeviceToHost** → copy from device to host

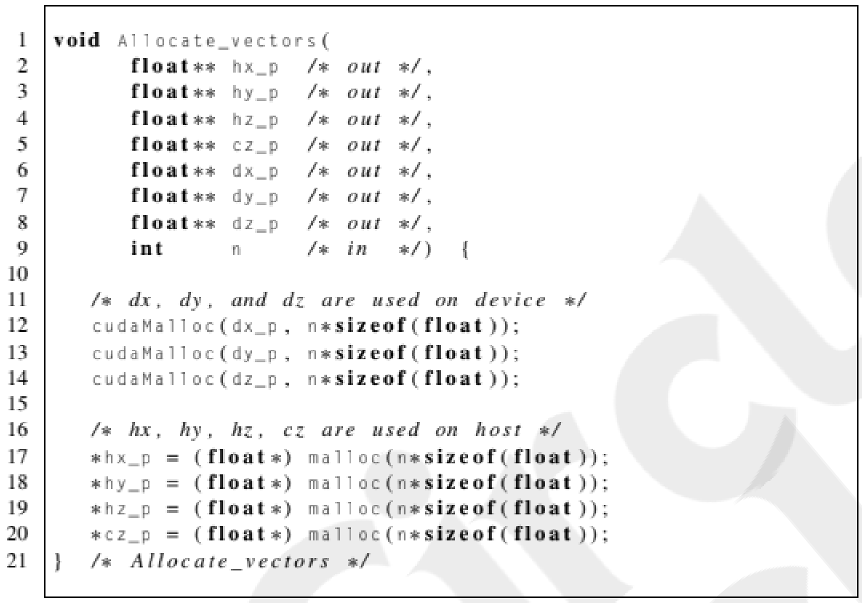
The kernel launch (Line 28) uses **dx**, **dy**, and **dz**, which are valid device pointers. After execution, the results in **dz** are copied back to **hz** on the host (Line 31) using **cudaMemcpy()** again. Since **cudaMemcpy** is **synchronous**, it ensures that the kernel completes before data transfer begins, making an explicit **cudaDeviceSynchronize()** unnecessary.

Finally, the program verifies the results, releases memory, and exits. The **Free_vectors** function now frees both host and device allocations: **free()** for host memory and **cudaFree()** for device memory. Thus, the modified program manages **seven pointers** in total—three on the host, three on the device, and one for result verification.



```
1  __global__ void Vec_add(
2        const float  x[]  /* in   */,
3        const float  y[]  /* in   */,
4        float        z[]  /* out  */,
5        const int    n    /* in   */) {
6     int my_elt = blockDim.x * blockIdx.x + threadIdx.x;
7
8     if (my_elt < n)
9        z[my_elt] = x[my_elt] + y[my_elt];
10 }  /* Vec_add */
11
12 int main(int argc, char* argv[]) {
13    int n, th_per_blk, blk_ct;
14    char i_g;  /* Are x and y user input or random? */
15    float *hx, *hy, *hz, *cz; /* Host arrays          */
16    float *dx, *dy, *dz;      /* Device arrays        */
17    double diff_norm;
18
19    Get_args(argc, argv, &n, &blk_ct, &th_per_blk, &i_g);
20    Allocate_vectors(&hx, &hy, &hz, &cz, &dx, &dy, &dz, n);
21    Init_vectors(hx, hy, n, i_g);
22
23    /* Copy vectors x and y from host to device */
24    cudaMemcpy(dx, hx, n*sizeof(float), cudaMemcpyHostToDevice);
25    cudaMemcpy(dy, hy, n*sizeof(float), cudaMemcpyHostToDevice);
26
27
28    Vec_add <<<blk_ct, th_per_blk>>>(dx, dy, dz, n);
29
30    /* Wait for kernel to complete and copy result to host */
31    cudaMemcpy(hz, dz, n*sizeof(float), cudaMemcpyDeviceToHost);
32
33    Serial_vec_add(hx, hy, cz, n);
34    diff_norm = Two_norm_diff(hz, cz, n);
35    printf("Two-norm of difference between host and ");
36    printf("device = %e\n", diff_norm);
37
38    Free_vectors(hx, hy, hz, cz, dx, dy, dz);
39
40    return 0;
41 }  /* main */
```

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



```
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 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 */
```

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

5.9 Returning results from CUDA kernels

CUDA kernels have several important restrictions and behaviors that programmers must keep in mind. **All kernels must have a return type of void**, meaning they cannot directly return values to the host. Similarly, they cannot use **standard C pass-by-reference** to send results back, since **host memory addresses are generally invalid on the device**, and vice versa.

For example, if we attempt to pass a host variable's address to a kernel (as in the code using &sum), the device will likely hang or the host will print the unchanged value because the device tried to write to an invalid address.

To "return" values from a kernel, several methods can be used:

1. **Using Unified Memory –** The host and device share memory allocated with cudaMallocManaged.

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

   Here, the device writes to a managed pointer that is automatically synchronized with the host.

2. **Without Unified Memory** – Separate allocations are made for host and device, and data is explicitly copied back using `cudaMemcpy`.

```
int *hsum_p, *dsum_p;
hsum_p = (int*)malloc(sizeof(int));
cudaMalloc(&dsum_p, sizeof(int));
Add<<<1,1>>>(2, 3, dsum_p);
cudaMemcpy(hsum_p, dsum_p, sizeof(int), cudaMemcpyDeviceToHost);
printf("The sum is %d\n", *hsum_p);
```

This version ensures compatibility with older systems but requires manual data transfer.

3. **Using a Managed Global Variable** – A global variable can be declared with the qualifier __managed__, making it accessible from both the host and the device.

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

The host can then read sum after synchronization. However, this approach is only supported on devices with **compute capability ≥ 3.0**, and **simultaneous host-device access** is restricted on devices below **6.0**.

Although the managed global variable method is convenient, it reduces **modularity** and is generally discouraged in complex programs. Thus, while CUDA provides flexible options for returning results from kernels, explicit memory management often remains the more robust and portable approach.

5.10 CUDA trapezoidal rule I

5.10.1 The trapezoidal rule

The text says **"Recall (see Sec. 3.2.1)"**, which refers to the earlier explanation of the trapezoidal rule in Chapter 3.

Figure 3.3 illustrates the **trapezoidal approximation of the area under a curve** — it visually shows dividing the interval $[a, b]$into $n$subintervals and representing each part as a trapezoid.

Figure 3.4 (mentioned later) specifically shows **the ith trapezoid**, where $h$is the width and $f(x_i)$, $f(x_{i+1})$are the two base lengths.
This figure visually corresponds to the expression

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

So when citing in your lecture notes or slides, you can keep the references like this:

The trapezoidal rule divides the interval $[a, b]$into $n$equal subintervals (see Fig. 3.3), each approximated by a trapezoid of height $h = (b - a)/n$. For the i^{th}trapezoid (see Fig. 3.4), the area is $\frac{h}{2}[f(x_i) + f(x_{i+1})]$.

Then follow it with:

Hence, the total area is approximated as

$$\int_a^b f(x)\,dx \approx \frac{h}{2}\left[f(a) + f(b) + 2\sum_{i=1}^{n-1} f(x_i)\right]$$

5.10.2 A CUDA implementation

In the serial version of the trapezoidal rule, most of the computation occurs within the **for loop** that evaluates the function $f(x_i)$and accumulates the partial sums into the total area. According to **Foster's design methodology**, these two major tasks can be parallelized:

1. **Function evaluation** — computing $f(x_i)$at each subinterval.
2. **Summation** — adding $f(x_i)$to the partial sum, trap.

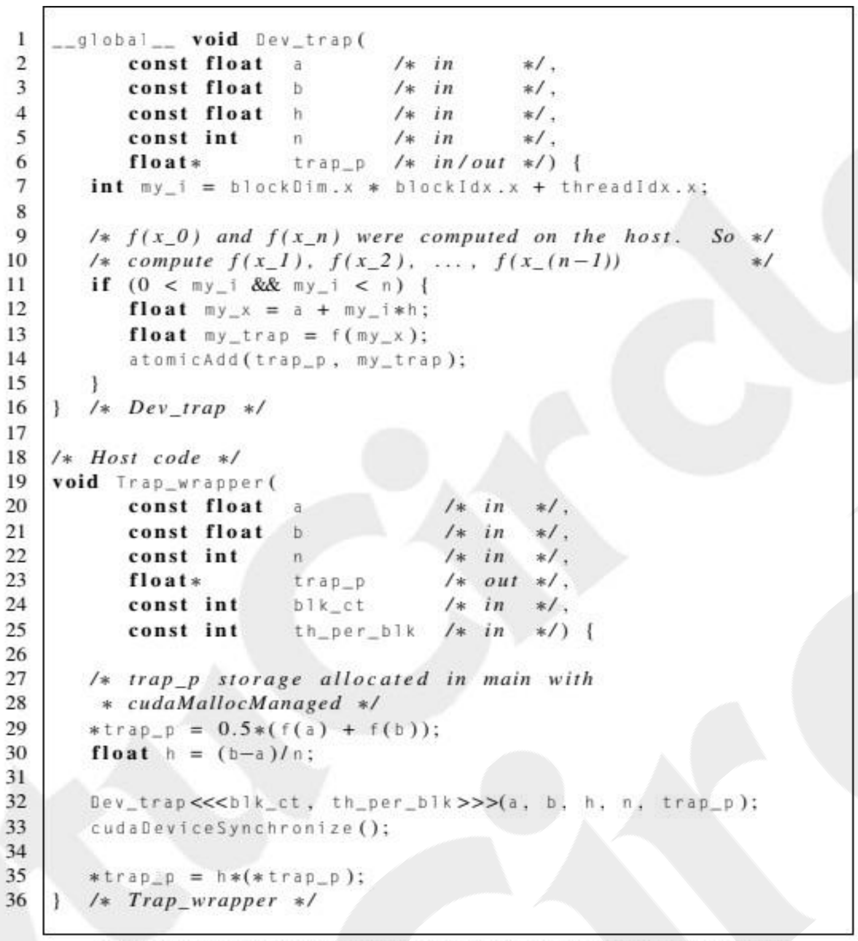
Since the second depends on the first, both can be combined into a single unit of work. Thus, in CUDA, each thread can be assigned one iteration of the serial loop — computing its own $x_i$and $f(x_i)$and contributing to the total sum.

Each thread is identified by a unique **rank (thread index)**, similar to the vector addition program discussed earlier.
The basic mapping looks like this:

```
/* 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);
trap += my_trap;
```

However, this naive implementation introduces several **issues**:

1. **Uninitialized variables** — The parameters h and trap must be properly initialized before use.
2. **Out-of-range thread indices** — The loop in the serial code runs from i = 1 to i = n − 1, but thread indices start from 0. Threads with indices outside this range must not execute computations.
3. **Race condition** — Multiple threads updating the shared variable trap simultaneously can lead to incorrect results, since one thread may overwrite another's update. (Refer to **Section 2.4.3**, "Race Conditions.")
4. **Return type restriction** — CUDA kernels have a void return type, so results must be communicated through parameters or device memory.
5. **Final scaling** — After all threads complete their partial computations, the total sum must be multiplied by h to obtain the final area.



```
__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 */
```

Program 6.12: CUDA kernel and wrapper implementing trapezoidal rule.

Program 6.12 in the textbook presents a corrected CUDA implementation that resolves these issues — using shared memory, synchronization, and reduction techniques to safely combine partial sums from multiple threads.

5.10.3 Initialization, return value, and final update

To handle initialization and final computation (Items 1 and 5 from Section 6.10.2), one might consider assigning a **single thread**—for example, **thread 0 of block 0**—to perform these operations:

```
int my_i = blockDim.x * blockIdx.x + threadIdx.x;
if (my_i == 0) {
    h = (b - a) / n;
    trap = 0.5 * (f(a) + f(b));
}
...
if (my_i == 0)
    trap = trap * h;
```

However, this approach introduces important **problems** related to **variable scope** and **thread synchronization**.

Private versus Shared Variables

In CUDA, each thread has its own **stack**, similar to threads in Pthreads and OpenMP.
All **formal arguments** of a kernel are allocated on this per-thread stack, meaning that each thread gets its **own private copies** of variables such as h and trap.
Hence, any modification to h or trap by one thread will not be visible to others.

The variable h can be safely initialized **once in the host** before launching the kernel, as its value remains constant for all threads.
But trap, which accumulates partial results, must be **shared among threads** to correctly compute the total area.

Using Managed Memory for Shared Access

To achieve shared access, we allocate trap in **unified memory** (accessible by both host and device) and pass a **pointer** to it as a kernel argument:

Here, each thread receives its **own copy of the pointer** trap_p, but all these pointers reference the **same memory location**.
Thus, the dereferenced variable *trap_p becomes **shared** among all threads.

This pointer-based approach also naturally resolves the earlier limitation that **CUDA kernels cannot return values** (Item 4 from Section 6.10.2), since results can now be written directly to the shared memory location.

Wrapper Function

As seen in **Program 6.12**, the CUDA trapezoidal rule employs a **wrapper function** named Trap_wrapper.
A **wrapper function** primarily exists to call another function—in this case, the kernel—while performing necessary setup and cleanup tasks.
It handles all **preparation before launching the kernel** (such as memory allocation, initialization, and parameter setup) and any **post-processing** (like scaling or memory deallocation) once the kernel completes execution.

5.10.4 Using the Correct Threads

In the CUDA implementation of the trapezoidal rule, we assume that the **total number of threads**, given by
blk_ct * th_per_blk, is **at least as large as the number of trapezoids** n.

Recall that in the serial version, the for loop iterates over indices
i = 1, 2, ..., n - 1.
This means that **thread 0** (corresponding to i = 0) and any **thread with index greater than or equal to n** should **not execute** the main computation portion of the kernel.

To enforce this, a simple conditional check is added before performing any calculations inside the kernel:

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

This ensures that only threads corresponding to valid trapezoid indices participate in evaluating the function and updating the shared variable.

Such a check prevents **out-of-range accesses** and avoids unnecessary or invalid computations by extra threads that may have been launched to fill a complete block.
This conditional test appears as **Line 11 in Program 6.12**, where it forms an essential safeguard for correct parallel execution.

5.10.5 Updating the Return Value and atomicAdd

The next issue concerns updating the shared memory location *trap_p (Item 3 from the earlier list). Since this variable is **shared among threads**, a simple update like

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

creates a **race condition**. Multiple threads may attempt to modify *trap_p simultaneously, and because the operation is not atomic, the final value stored could be **unpredictable or incorrect**.

To eliminate this problem, CUDA provides **atomic operations**, which ensure that certain updates appear **indivisible** to all other threads. An operation is said to be *atomic* if, from the viewpoint of every thread, it happens all at once — no other thread can observe the intermediate state of the operation.

Thus, if one thread performs an atomic addition, any other thread's attempt to read or write to the same memory location will occur **entirely before or after** the operation — never during it. This guarantees consistent and correct results even in concurrent execution.

Addition, by default, is **not atomic**; it involves multiple machine instructions. Hence, CUDA provides special built-in atomic functions to perform such updates safely. The one relevant to our case is atomicAdd, defined as:

```
__device__ float atomicAdd(
    float* float_p,  // in/out: address of variable in device memory
    float  val       // in: value to add
);
```

This function **atomically adds** val to the value stored at the location pointed to by float_p, and then stores the result back in the same location.
It also returns the value that was originally present in *float_p at the start of the call.

In the CUDA trapezoidal rule implementation (Program 6.12), atomicAdd is used (see **Line 14**) to safely accumulate each thread's contribution (my_trap) to the shared total area variable *trap_p, ensuring correctness even when thousands of threads execute concurrently.

5.10.6 Performance of the CUDA trapezoidal rule

To determine the **run-time performance** of the CUDA trapezoidal rule, we measure the execution time of the **Trap_wrapper** function. This wrapper function encapsulates the entire trapezoidal rule computation — including initialization, kernel execution, and final updates — so its elapsed time effectively represents the **total GPU execution time**.

The execution steps included within Trap_wrapper are:

- Initialization of *trap_p with the starting sum (Line 29).
- Calculation of the step size h = (b - a)/n (Line 30).
- Launching of the CUDA kernel Dev_trap, where each thread computes its partial contribution.
- The final update of *trap_p after all threads finish (Line 35).

Hence, by placing timing calls immediately **before and after** the call to Trap_wrapper, we capture the **total elapsed time** of the GPU-based trapezoidal computation.

The timing is measured using the GET_TIME macro (defined in timer.h), which records wall-clock time:

As noted in earlier performance discussions (Section 2.6.4), multiple timing trials should be conducted, and the **minimum** time is typically reported. However, if only a few results fall within 1% of the minimum, that value may not be reproducible. In such cases, it's better to report the **mean or median** elapsed time to provide a more representative and stable measure of performance.

In the experiments described, both the serial and CUDA trapezoidal rule implementations were executed repeatedly, and **mean times** (averaged over at least 50 runs) were reported.

For example, using:

- $n = 2^{20} = 1{,}048{,}576$trapezoids
- $f(x) = x^2 + 1$
- Interval $[a, b] = [-3{,}3]$
- 1024 blocks × 1024 threads per block = 1,048,576 total threads

The results in **Table 6.5** show:

- The **GK20A GPU** (192 SPs) was significantly faster than an **ARM Cortex-A15** CPU.
- However, a single **Intel Core i7** core was still faster than the GK20A GPU.
- The **Titan X GPU** (3072 SPs) outperformed the Intel core by **about 45%**, though the gain was less than expected given the large number of streaming processors (SPs).

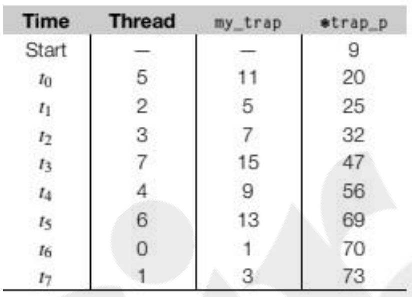
These results highlight that **raw hardware parallelism** alone does not guarantee proportional speedup — **memory bandwidth, kernel launch overhead**, and **arithmetic intensity** all influence the effective performance of CUDA programs like the trapezoidal rule.

5.11 CUDA trapezoidal rule II: improving performance

5.11.1 Tree-structured communication

We can visualize the execution of the *global sum* in the CUDA trapezoidal rule as a more or less random, linear ordering of the threads. For example, consider 8 threads within a single block numbered **0–7**. One of the threads will be the first to succeed with the call to atomicAdd; suppose it's thread 5. Then another thread—say thread 2—succeeds next, and so on. Continuing in this way, we get a sequence of atomic additions, one per thread. **Table 6.6** illustrates how this sequence might progress over time.

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

In this example, to keep the computation simple, assume $f(x) = 2x + 1$, $a = 0$, and $b = 8$. Hence,

$$h = \frac{8-0}{8} = 1,$$

and the initial value of trap_p is:

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

What's important to note is that this approach may effectively **serialize** the threads since each thread must wait for its turn to perform its atomicAdd. The computation then requires a sequence of 8 consecutive additions. **Figure 6.3** illustrates one possible order of execution.

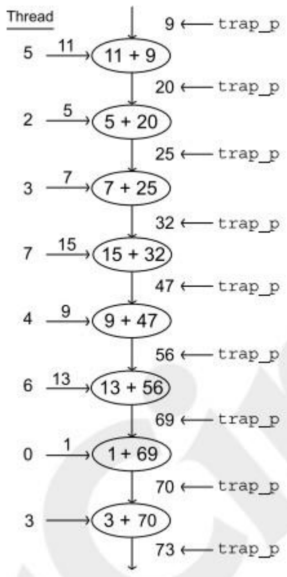


FIGURE 6.3

Basic sum.

To improve efficiency, instead of allowing each thread to wait for its turn to add its result into *trap_p, threads can be **paired** so that half of the active threads add their partial sum to their partner's sum. This yields a **tree-structured pattern** (or "shrub-like" structure) as shown in **Figure 6.4**.

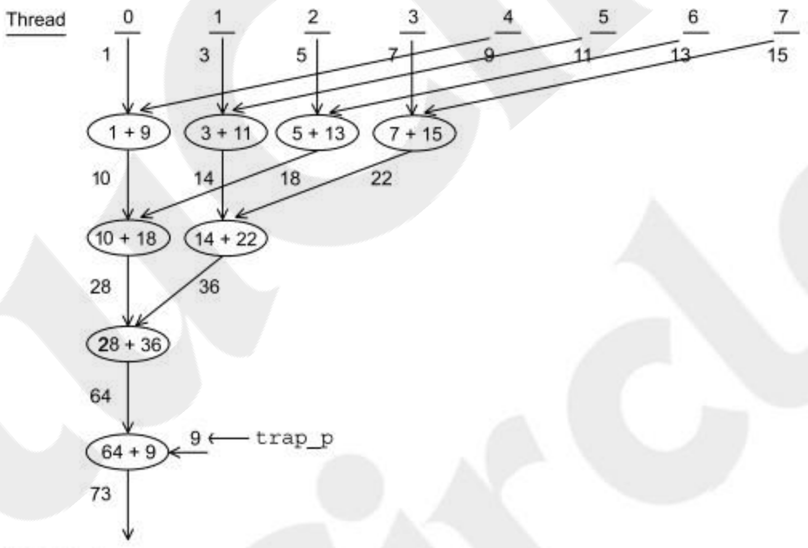


FIGURE 6.4

Tree-structured sum.

In the figures, we've gone from needing 8 consecutive additions (linear sequence) to only 4 (tree sequence). More generally:

- With the basic approach, if we double the number of threads (e.g., 8 → 16), we double the number of additions required.
- With the tree-structured approach, we add only one more step to the sequence.

Thus, with $t$threads and $t$values:

- The **basic approach** requires $t$additions.
- The **tree-structured approach** requires $\log_2(t) + 1$additions.

For example:

- $t = 1000$: from 1000 to 11 additions.
- $t = 1{,}000{,}000$: from 1,000,000 to 21 additions.

There are two standard CUDA implementations of the tree-structured sum:

1. **Shared Memory Implementation:**
   Used on devices with compute capability < 3, where threads within a block store and reduce partial sums in shared memory.
2. **Warp Shuffle Implementation:**
   For devices with compute capability ≥ 3, CUDA provides **warp shuffle** functions that allow threads within a warp to directly access other threads' data. This approach eliminates the need for shared memory and synchronization barriers, providing faster reductions.

5.11.2 Local variables, registers, shared and global memory

NVIDIA GPU has access to two main types of memory:

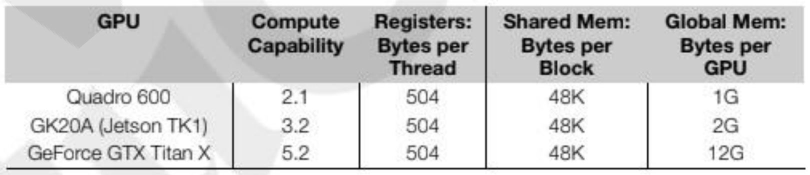
- **Shared memory**, which is private to the SM and accessible only by the threads within that SM. More precisely, the shared memory allocated for a *thread block* can be accessed only by threads belonging to that block.
- **Global memory**, which is accessible to all SMs and all threads in the grid.

The key distinction lies in **speed** and **capacity**. Shared memory is relatively small but very fast, while global memory is large but considerably slower. Thus, GPU memory can be viewed as a **three-level hierarchy**:

1. **Registers** – the fastest and smallest level.
2. **Shared memory** – intermediate in both speed and size.
3. **Global memory** – the largest but slowest level.

Table 6.7 provides comparative data illustrating these relative sizes. Typically, accessing data stored in registers takes about **1 clock cycle**. Moving data between shared memory locations can take roughly an **order of magnitude** longer, while global memory accesses may take **two to three orders of magnitude** more time.

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

This dramatic difference explains why optimizing memory usage is crucial for CUDA performance.

A natural question arises: *what about local variables?* Where are they stored, and how quickly can they be accessed?

The answer depends on the availability of registers and total memory usage. When sufficient registers are available, local variables are stored directly in **registers**, giving maximum speed. However, if register space is exhausted, the compiler automatically **spills** local variables to a special portion of **global memory** reserved for each thread — often called **local memory**. Although thread-private, this spilled memory inherits the **latency and access cost of global memory**.

Therefore, as long as there is enough register storage, a kernel's performance generally improves when it maximizes **register usage** and minimizes reliance on **shared** and especially **global** memory. The trade-off, however, is that register capacity is very limited compared to the much larger pools of shared and global memory.

5.11.3 Warps and warp shuffles

If we can carry out the **global sum entirely within registers**, the performance will naturally be superior to using shared or global memory. This is made possible through the **warp shuffle functions** introduced in CUDA devices with **compute capability 3.0** and higher.

In CUDA, a **warp** is a group of **threads with consecutive ranks** within a thread block. The number of threads in a warp is currently **32**, although NVIDIA has indicated this value may change in future architectures. The system provides a predefined variable:

```
int warpSize;
```

Threads in a warp execute in **SIMD (Single Instruction, Multiple Data)** fashion. This means that threads across different warps can execute different instructions simultaneously without penalty, but all threads within a single warp must execute the same instruction. When threads within a warp take **different control paths**, such as opposite branches of an if–else statement, they **diverge**. Once they complete their respective paths and resume executing the same instruction, they **converge**.

The **rank** of a thread within its warp is known as its **lane**, computed as:

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

The **warp shuffle functions** enable a thread to directly read values from the **registers of another thread** in the same warp. This avoids slower shared or global memory accesses. For the tree-structured sum, the relevant function is:

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

The **mask** specifies which threads are participating in the shuffle. Each bit corresponds to a thread's **lane ID**. For safe execution, all participating threads must have **converged**—that is, reached the call together—before any executes the shuffle. In most cases, **all threads** in the warp participate, so we define:

```
unsigned mask = 0xffffffff;
```

Here, 0x indicates a hexadecimal literal, and 0xffffffff corresponds to **32 binary 1s**, meaning **all 32 threads** are active in the operation.

When a thread with **lane l** calls _shfl_down_sync, it receives the value of var from the thread with:

```
lane = l + diff;
```

Since diff is an unsigned integer (non-negative), the operation retrieves data from a **higher-ranked thread**, hence the name **"shuffle down."**

The width parameter specifies the number of threads participating in the operation, but as it defaults to warpSize, it is usually omitted.

There are, however, some **potential pitfalls**:

- If thread *l* calls __shfl_down_sync but **thread l + diff** does not, the returned value is **undefined**.
- If **l + diff ≥ warpSize**, the call returns the value already stored in var on thread *l*.
- If the **thread block size** is **not a multiple of warpSize**, the last warp will have fewer than 32 threads. Suppose the last warp has *w* threads (0 < w < warpSize); then if **l + diff ≥ w**, the result is also undefined.

To ensure well-defined and efficient behavior, it is best to:

- Have **all threads** in a warp call __shfl_down_sync.
- Choose a **block size** that is a **multiple of warpSize (32)**.

This guarantees that every warp participates fully in the operation and prevents undefined results during register-level communication.

5.11.4 Implementing tree-structured global sum with a warp shuffle

A **tree-structured global sum** can be efficiently implemented within a warp using registers and the warp shuffle function. The implementation in CUDA is shown below:

```
__device__ float Warp_sum(float var) {
    unsigned mask = 0xffffffff;
    for (int diff = warpSize / 2; diff > 0; diff = diff / 2)
        var += __shfl_down_sync(mask, var, diff);
    return var;
} /* Warp_sum */
```

This function progressively reduces the number of active threads performing additions by half in each iteration, forming a **binary tree structure** of sums. As illustrated in **Fig. 6.5**, the process can be visualized using a smaller warp size of 8 for clarity.

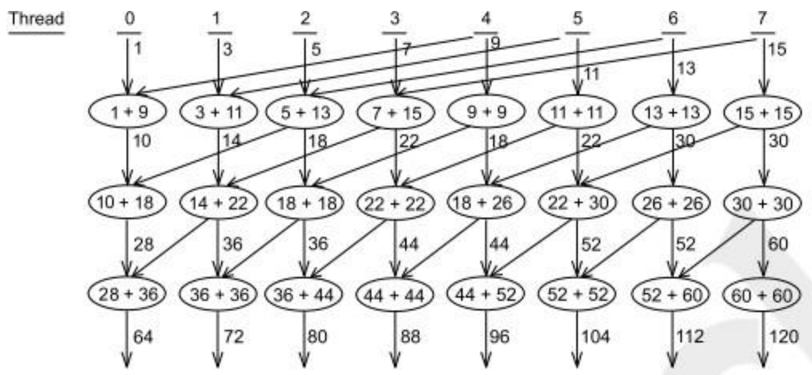


FIGURE 6.5

Tree-structured sum using warp shuffle.

At each step, the __shfl_down_sync function shifts the values of var from threads with higher lane IDs to those with lower ones, and the lower-lane threads accumulate these values. For instance, when diff = 4, threads with **lane IDs 0–3** receive values from threads **4–7**. However, for threads with **lane IDs 4–7**, since l + 4 ≥ 8, the function simply returns their own var value. Hence, their values get doubled at that stage. This behavior repeats in later iterations—for example, at diff = 2 for lanes 6 and 7, and at diff = 1 for lane 7—where only one arrow enters the summing node in **Fig. 6.5**.

From a **practical standpoint**, this implementation produces the **correct total sum only on the thread with lane ID 0**. The remaining threads will hold partial or redundant sums. If the final result needs to be available to **all threads** within the warp, an alternative function such as __shfl_xor can be used.

6.11.5 Shared memory and an alternative to the warp shuffle

If the GPU's compute capability is below 3.0, warp shuffle functions cannot be used since threads cannot access one another's registers directly. Instead, threads in the same block can communicate through **shared memory**, which, though slower than register access, can yield similar performance when properly managed.

Because threads in a warp execute synchronously, shared memory can mimic the behavior of warp shuffle operations. The following function demonstrates this:

```
__device__ float Shared_mem_sum(float shared_vals[]) {
    int my_lane = threadIdx.x % warpSize;
    for (int diff = warpSize / 2; diff > 0; diff /= 2) {
        int source = (my_lane + diff) % warpSize; // Ensure 0 <= source < warpSize
        shared_vals[my_lane] += shared_vals[source];
    }
    return shared_vals[my_lane];
}
```

All threads within a warp should invoke this function, with shared_vals located in the shared memory of the SM executing the warp. Since threads operate in **SIMD fashion**, updates occur in lockstep, eliminating race conditions—each thread reads from shared_vals[source] before writing to shared_vals[my_lane].

This approach is known as a **dissemination sum** (or dissemination reduction) rather than a traditional tree-structured sum. **Figure 6.6** illustrates the series of copy and addition operations that occur, though it omits the direct contributions of each thread for clarity. Every thread retrieves a value from another thread in each iteration, and by the end, all threads possess the same final sum—not just thread 0.

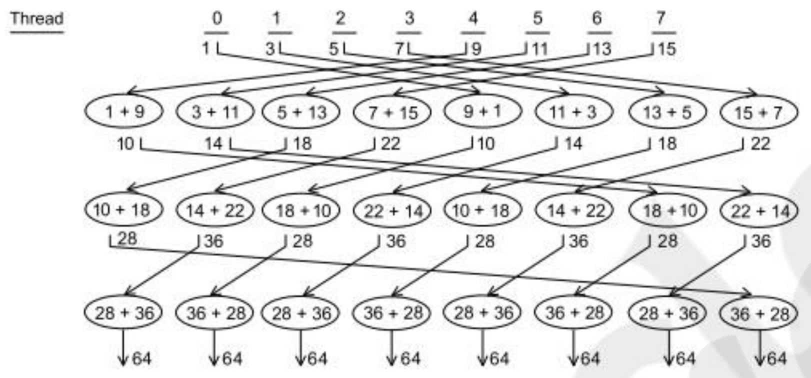


FIGURE 6.6

Dissemination sum using shared memory.

Although this full-warp summation is unnecessary for the trapezoidal rule, it can be advantageous in other algorithms. When a warp is active, all its threads either execute the same instruction or remain idle, so uniform instruction execution does not impose additional cost.

The Shared_mem_sum function itself does not require shared memory; its argument shared_vals could reside in either **global** or **shared memory**. However, performance improves when it is defined as a __shared__variable within the kernel, for instance:

```
__shared__ float shared_vals[32];
```

This reserves 32 float locations in the shared memory of each SM for its corresponding thread block.

If the shared memory size is not known at compile time, it can be declared dynamically:

```
extern __shared__ float shared_vals[];
```

When launching the kernel, the third parameter in the triple-angle brackets specifies the required shared memory size in bytes. For example, in a trapezoidal rule kernel Dev_trap, it can be invoked as:

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

This allocates th_per_blk floats in the shared memory for each thread block, ensuring efficient summation across threads.

5.12 Implementation of trapezoidal rule with warpSize thread blocks

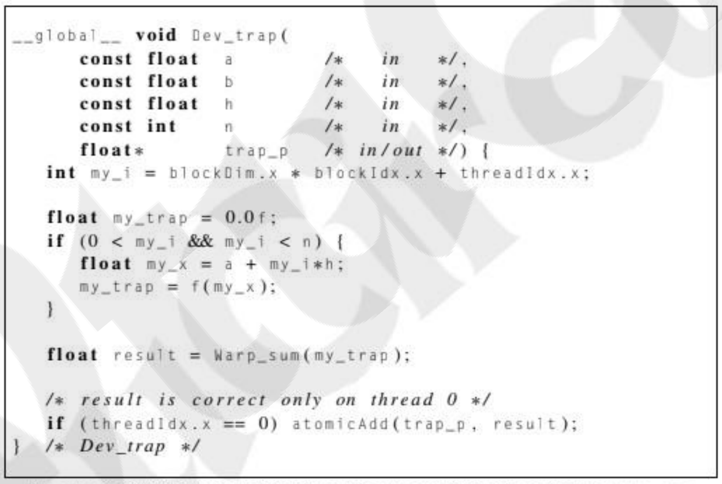
After exploring the use of **efficient summation techniques**, **warps**, **warp shuffles**, and **shared memory**, we can now develop improved CUDA implementations of the **trapezoidal rule**. In both implementations, each **thread block** consists of warpSize threads, and a **tree-structured sum** is employed to combine the results within each warp. Once the threads in a warp compute their respective function values and accumulate their local results, the **thread with lane ID 0** adds the warp's partial sum to the global total using the atomicAdd operation.

5.12.1 Host Code

For both the **warp shuffle** and **shared memory** versions, the **host code** remains nearly identical to that of the initial CUDA implementation. The main distinction is that there is **no th_per_blk variable**, as each thread block is now assumed to contain exactly warpSize threads. This simplifies the host configuration and ensures that each block aligns perfectly with the hardware's warp execution model, allowing the improved summation strategies to operate efficiently.

5.12.2 Kernel with warp shuffle

In **Program 6.13**, the kernel improves upon the earlier trapezoidal rule implementation by incorporating **warp-level summation** instead of having each thread update the global sum individually. The variable my_trap is initialized just as in the previous version (Program 6.12), representing the partial contribution computed by each thread.



```
__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 */
```

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

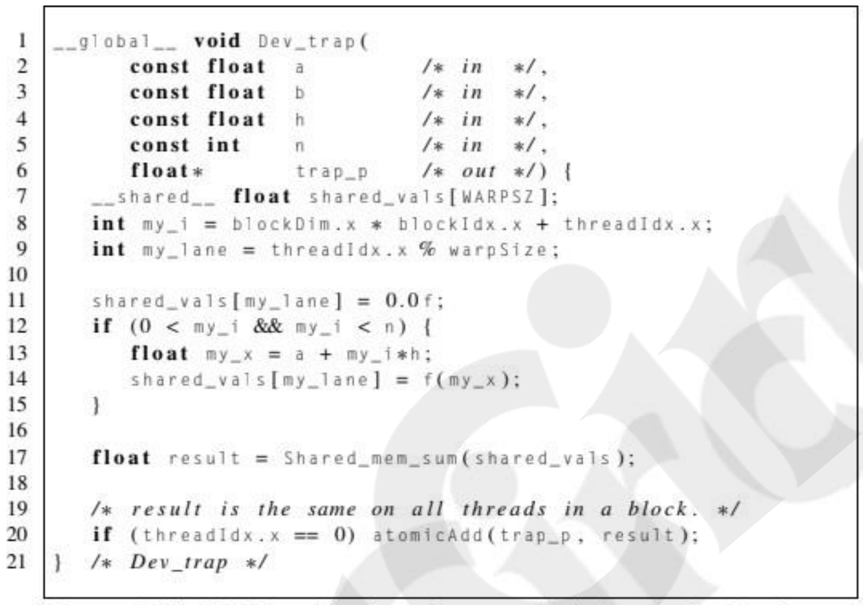
However, rather than directly adding each thread's contribution to the global sum (*trap_p), all threads within a **warp** (or equivalently here, a **thread block**) first call the **Warp_sum** function. This function performs a **tree-structured reduction** across the threads in the warp, producing a single sum value — the **warp sum**.

After this reduction step, only the **thread with lane ID 0** (the first thread in the warp) updates the global accumulator by adding the warp sum to *trap_p. Because multiple thread blocks can execute concurrently and attempt to update *trap_p at the same time, a **race condition** would arise if these updates were not synchronized.

To prevent this, the kernel employs **atomicAdd**, ensuring that each warp's contribution is safely and correctly accumulated into the global total without interference from other warps or thread blocks. This atomic operation guarantees data consistency while allowing all warps to execute in parallel.

5.12.3 Kernel with shared memory

In **Program 6.14**, the kernel uses **shared memory** instead of warp shuffle for summing thread results within a warp. It declares a shared array using __shared__ float shared_vals[WARPSZ]; where WARPSZ is defined as #define WARPSZ 32 since warpSize isn't known at compile time.



```
__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] = 0.0f;
   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);

   /* result is the same on all threads in a block. */
   if (threadIdx.x == 0) atomicAdd(trap_p, result);
}  /* Dev_trap */
```

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

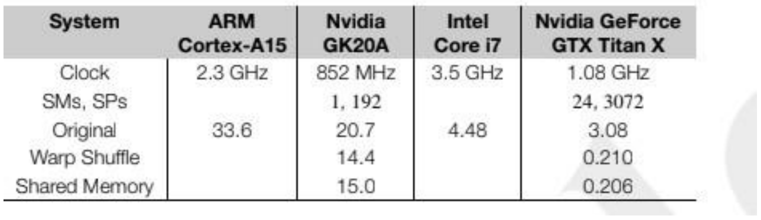
Each thread stores its partial sum in shared_vals, and the **Shared_mem_sum** function performs the reduction. Finally, **lane 0** adds the warp's total to the global sum using atomicAdd to avoid race conditions. This version functions like the warp shuffle implementation but uses shared memory for intra-warp communication.

5.12.4 Performance

To evaluate performance (Table 6.8), the same integration problem as before (Table 6.5) is used: $f(x) = x^2 + 1$over the interval $[-3,3]$with $2^{20} = 1{,}048{,}576$trapezoids.

Since each thread block has 32 threads, the setup uses **32,768 thread blocks (32 × 32,768 = 1,048,576)**. This configuration allows comparison of different implementations—original, warp shuffle, and shared memory—for performance efficiency.

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

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

In CUDA programming, a **race condition** occurs when multiple threads try to access or modify the same data simultaneously, and the final result depends on the order of execution.

In the case discussed, the race condition arises when **warp 0** tries to compute the total of the warp sums in a block **before** all other warps have finished their individual sums. For example, if warp 0 finishes early and starts reading warp 1's partial sum before warp 1 is done, the result will be **incorrect**.

5.13.1 __syncthreads

CUDA solves this problem using a **barrier synchronization function** called __syncthreads();.
This function ensures that **all threads in a block** reach the same point before any of them proceed further. When __syncthreads() is used, warp 0 will wait until all other warps have completed their work, thus eliminating the race condition.

The correct pseudocode for computing the block sum is:

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

After inserting __syncthreads(), the addition of warp sums begins **only after** every warp has finished computing its partial sum.

Two important points to remember:

1. **All threads in a block must execute __syncthreads()** — if only a subset of threads reach it (for example, due to an if condition), those threads will wait forever, causing the kernel to hang.
2. **_syncthreads() works only within a block**, not across multiple blocks. Threads in different blocks execute independently and cannot synchronize using this function.

Hence, __syncthreads() is essential for correct coordination of threads within a block, especially when sharing intermediate results through shared memory.

5.13.2 More shared memory

When implementing the pseudocode in actual CUDA code, an additional detail must be handled: **how warp 0 accesses the results computed by other warps after the __syncthreads() call.**

Warp shuffle instructions allow threads to share register values, but **only within the same warp**. Threads in **different warps cannot directly access each other's registers**. Hence, the final block-level reduction cannot rely on warp shuffle alone.

The solution is to use **shared memory**, which can be accessed by all threads in a block. Each warp can store its partial sum into a shared memory array. After synchronization, warp 0 can read these values to compute the total block sum.

```
__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
```

5.13.3 Shared memory warp sums

When using shared memory instead of warp shuffles, we need enough space for all threads in a block. So we declare:

```cuda
#define MAX_BLKSZ 1024
__shared__ float thread_calcs[MAX_BLKSZ];
```

Each warp uses a subarray of `thread_calcs` for its calculations:

```c
float *shared_vals = thread_calcs + my_warp * warpSize;
shared_vals[my_lane] = f(my_x);
float my_result = Shared_mem_sum(shared_vals);
```

A race condition occurs if all warps try to store their warp sums in the same shared region (e.g., thread_calcs[w]). To avoid this, synchronization is needed:

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

However, frequent __syncthreads() calls reduce performance. A better solution is for each warp to store its sum in the **first element of its own subarray**:

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

This is safe because each warp writes to a different part of shared memory, so no overlap or race condition occurs.

5.13.4 Shared Memory Banks

Shared memory in each Streaming Multiprocessor (SM) is divided into **32 banks** (16 for older GPUs). Each bank can be accessed by one thread per warp simultaneously. If two or more threads access **different locations within the same bank**, the accesses are **serialized**, reducing performance.

Table 6.9 illustrates this structure: consecutive elements of thread_calcs are distributed across the 32 banks. When threads access elements like 0, 32, 64, …, 992, all these map to the **same bank**, causing serialization and slower performance.

- If each thread in a warp accesses a **different bank**, access is parallel.
- If multiple threads access **different locations** in the same bank, access is serialized.
- If multiple threads read the **same location**, the read is broadcast efficiently.

To optimize access, warp sums should be stored in **contiguous memory** (e.g., a row of the bank layout).

Hence, we declare two shared arrays to separate thread and warp sums:

```
__shared__ float thread_calcs[MAX_BLKSZ];
__shared__ float warp_sum_arr[WARPSZ];
float *shared_vals = thread_calcs + my_warp * warpSize;
float my_result = Shared_mem_sum(shared_vals);
if (my_lane == 0) warp_sum_arr[my_warp] = my_result;
__syncthreads();
```

5.13.5 Finishing Up

After all warps compute their sums, **warp 0** adds up the values in warp_sum_arr. Finally, **thread 0** adds the block's total to the global sum using atomicAdd:

```
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);
```

This ensures correctness even when the number of warps in a block is less than 32.

5.13.6 Performance

The shared memory and warp shuffle versions both improve performance by reducing the number of calls to atomicAdd.

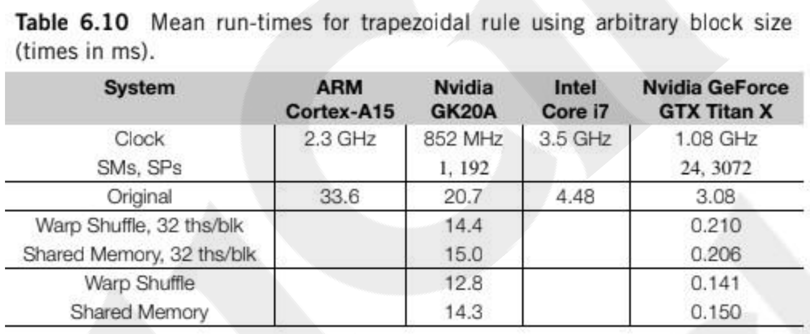


Table 6.10 Mean run-times for trapezoidal rule using arbitrary block size (times in ms).

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