Topics in Parallel and Distributed Computing: Introducing Algorithms, Programming, and Performance within Undergraduate Curricula*†‡

Chapter 8 – The Realm of Graphical Processing Unit (GPU) Computing

Vivek K. Pallipuram[1] and Jinzhu Gao[2]

[1]University of the Pacific, 3601 Pacific Avenue, Stockton, CA, vpallipuramkrishnamani@pacific.edu
[2]University of the Pacific, 3601 Pacific Avenue, Stockton, jgao@pacific.edu

*How to cite this book: Prasad, Gupta, Rosenberg, Sussman, and Weems. Topics in Parallel and Distributed Computing: Enhancing the Undergraduate Curriculum: Performance, Concurrency, and Programming on Modern Platforms, Springer International Publishing, 2018, ISBN : 978-3-319-93108-1, Pages: 337.

†Free preprint version of this book: https://grid.cs.gsu.edu/~tcpp/curriculum/?q=cder_book_2

‡Free preprint version of volume one: https://grid.cs.gsu.edu/~tcpp/curriculum/?q=cedr_book

Abstract

The goal of the chapter is to introduce the upper-level Computer Engineering/-Computer Science undergraduate (UG) students to general-purpose graphical processing unit (GPGPU) computing. The specific focus of the chapter is on GPGPU computing using the Compute Unified Device Architecture (CUDA) C framework due to the following three reasons: 1) Nvidia GPUs are ubiquitous in high-performance computing, 2) CUDA is relatively easy to understand versus OpenCL, especially for UG students with limited heterogeneous device programming experience, and 3) CUDA experience simplifies learning OpenCL and OpenACC. The chapter consists of nine pedagogical sections with several active-learning exercises to effectively engage students with the text. The chapter opens with an introduction to GPGPU computing. The chapter sections include: 1) Data parallelism; 2) CUDA program structure; 3) CUDA compilation flow; 4) CUDA thread organization; 5) Kernel: Execution configuration and kernel structure; 6) CUDA memory organization; 7) CUDA optimizations; 8) Case study: Image convolution on GPUs; and 9) GPU computing: The future. The authors believe that the chapter layout facilitates effective student-learning by starting from the basics of GPGPU computing and then leading up to the advanced concepts. With this chapter, the authors aim to equip students with the necessary skills to undertake graduate-level courses on GPU programming and make a strong start with undergraduate research.

Relevant core courses: Computer Systems Architecture and Advanced Computer Systems courses.

Relevant PDC topics: Table i lists the relevant PDC concepts covered along with their Bloom levels.

Learning outcomes: By the end of this chapter, students will be able to:

- Explain CUDA concepts including thread management, memory management, and device management.
- Identify performance bottlenecks in CUDA programs and calculate performance achieved in floating-point operations per second (FLOPS).

Table i: PDC concepts across chapter sections and their Bloom levels.

PDC Concept	Chapter Section								
	8.1	8.2	8.3	8.4	8.5	8.6	8.7	8.8	8.9
Data parallelism	C								
GPGPU devices		C	A	A	A	A	A	A	
nvcc compiler			A						
Thread management				A	A				
Parallel patterns							A	A	
Performance evaluation							A		
Performance optimization							A	A	
CUDA		A		A	A	A	A	A	
Advancements in GPU computing									K

- Develop CUDA programs and apply optimizations pertaining to memory hierarchy, instructions, and execution configuration.

Context for use: The book chapter is envisioned for upper-level Computer Science/Computer Engineering undergraduate courses/electives on systems, advanced computer systems architecture, and high-performance computing with GPUs. The book chapter is also intended as a "quick start" guide to general-purpose GPU programming in undergraduate research.

The general-purpose graphical processing units (commonly referred to as GPGPUs) are throughput-oriented devices. Unlike the conventional central processing units (CPUs) that employ a significant portion of the silicon wafer for caches, GPGPU devices devote a large chunk of the chip real-estate to computing logic. Consequently, GPGPU devices today feature several hundreds of cores dedicated to performing accelerated computing. To unleash the tremendous power in these computing cores, programmers must create several hundreds of thousands of threads. This task warrants programmers devise creative techniques for task decomposition, data partitioning, and synchronization. The GPGPU computing includes an additional challenge of CPU-GPGPU device communication, which stems from the fact that CPU and GPGPU device memories are typically disjoint. In most GPGPU programs, the CPU (host) prepares and transfers the data to the GPGPU device via the Peripheral Interconnect Express (PCIe) bus for computations. Once the GPGPU device finishes all of the computations, it sends the processed data back to the CPU host via the PCIe bus. A few recent architectures from AMD feature accelerated processing units (APUs) that integrate CPU and GPU in a single chip. AMD calls this approach as heterogeneous unified memory access (hUMA) where the CPU and GPU memory spaces are unified, thereby avoiding any explicit data transfers between the CPU host and GPU device. However, such integration leads to CPU-GPU competition for the on-chip resources, leading to limited performance benefits. This chapter builds the GPGPU programming concepts using the disjoint CPU-GPGPU memory model.

To enable programmers to perform general purpose computing with GPUs, NVIDIA introduced the Compute Unified Device Architecture (CUDA) [1] in 2006, ultimately replacing the notion of "express-it-as-graphics" approach to GPU computing. CUDA is appropriately classified as a parallel computing platform and programming model – it helps programmers to develop GPGPU programs written in common languages such as C, C++, and Fortran by providing an elegant set of keywords and language extensions. Additionally, CUDA provides useful libraries such as the CUDA Basic Linear Algebra Subroutines (cuBLAS) library [2] for GPGPU accelerated linear algebra routines and the CUDA Deep Neural Network (cuDNN) library [3] for GPGPU accelerated primitives for deep neural networks. At the time of this writing, CUDA is in its current avatar CUDA 9 and is freely available for Linux, Windows, and Mac OSX.

GPGPU programming has continued to evolve ever since the introduction of CUDA. Open Computing Language (OpenCL [4]) was released in 2009 as a royalty-free standard for parallel programming on diverse architectures such as GPGPUs, multi-core CPUs, field programmable gate arrays (FPGAs), and dig-

ital signal processors (DSPs). Using a set of platform specific modifications, OpenCL allows programmers to adapt their codes for execution across a variety of heterogeneous platforms. Both CUDA and OpenCL tend to be verbose, for instance CUDA traditionally requires programmers to perform explicit data transfers between the CPU host and GPGPU device. The CPU host explicitly calls the GPGPU device functions (called kernels) to execute the parallel tasks. OpenCL, with its cross-platform requirements, further requires programmers to explicitly create device-related task queues. To reduce such burden on programmers, OpenACC [5] standard was officially released in 2013 as a paradigm to simplify CPU-GPGPU programming. OpenACC offers compiler directives (for example, `#pragma acc`) that are strategically placed across the source code to automatically map computations to the GPGPU device. The software advancements are not only limited to GPGPU programming models – software libraries such as Thrust [6] accelerate GPGPU code development by providing helper classes and functions for common algorithms such as sort, scan, reduction, and transformations, enabling programmers to focus on the high-level details of the application.

This chapter introduces students to the basics of GPGPU computing using the CUDA programming model. Section 8.1 introduces the concept of data parallelism, which is critical for GPGPU computing. Section 8.2 explains the typical structure of a CUDA program. Section 8.3 describes the compilation flow of CUDA programs. Sections 8.4 and 8.5 describe the CUDA thread organization and CUDA kernel configuration, respectively. Section 8.6 details the GPGPU memory organization as viewed by a CUDA program. Section 8.7 expounds several CUDA optimization techniques employed by programmers to maximize the application performance. The chapter concludes in Section 8.8 on convolution with GPGPU devices as a case study. By the end of this chapter, students will be able to explain computation mapping to CUDA threads, write GPGPU device kernels, and employ optimization strategies to achieve high application performance using GPGPU devices.

8.1 Data parallelism

Several scientific applications today process large amounts of data. Some example applications include molecular dynamics, deep neural networks, and image processing, among others. A sequential scan of the data for such applications on a conventional CPU may incur significant application runtime. Fortunately, several scientific applications offer data parallelism, meaning the data can be partitioned

into several chunks that can be executed independent of each other. The data parallelism is the primary source of scalability for many programs. In a molecular dynamics simulation, the electrostatic potential of atoms at grid points is evaluated in parallel. In a neural network simulation, the voltages of firing neurons at a given neuron layer are independent, and therefore can be evaluated in parallel. Several image-processing applications offer data parallelism via independent evaluation of pixel attributes. Life teaches us several lessons – including data parallelism! When the professor allows students to collaborate for an assignment, students divide work (equally) with each other and complete the assignment in a short time. Similarly, teaching assistants divide the grading work among themselves to reduce the grading time.

Active Learning Exercise 1 – Identify five common activities in day-to-day life that express significant data parallelism.

GPGPU devices work extremely well with applications that offer significant data parallelism. In fact, the very primitive job of a GPU device, i.e. graphics rendering, is extremely data parallel. Consider a simple example of vector-vector addition to illustrate the concept of data parallelism. Figure 8.1 shows the addition of two vectors A and B; the result is stored in vector C. The corresponding elements of vectors A and B are added using processing elements, PEs. Clearly, each processing element, PE_i, works independently of any other processing element. Therefore, each data element of vector C can be evaluated in parallel, thereby utilizing data parallelism inherent in this operation.

Matrix-matrix multiplication is another frequently used mathematical operation that offers significant data parallelism. Consider the multiplication of two matrices, $A_{m\times n}$ and $B_{n\times p}$; the result is stored in the matrix $C_{m\times p}$. Each element c_{ij} of $C_{m\times p}$ is evaluated by computing the scalar product (also called the inner product in the context of Euclidean space) between the i^{th} row of $A_{m\times n}$ and j^{th} column of matrix $B_{n\times p}$. Equation 8.1 summarizes c_{ij} computation.

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \times b_{kj} \tag{8.1}$$

A careful inspection of the above equation reveals that computation of any c_{ij} is independent of the others; therefore, c_{ij} can be computed in parallel. Matrix-matrix multiplication is an interesting operation because it can be parallelized in a variety of ways. For example, one can create $m \times p$ processing elements (PEs) where each PE_{ij} computes a specific matrix element, c_{ij}. Consider another example, where the PEs perform partial product computations and then add the partial

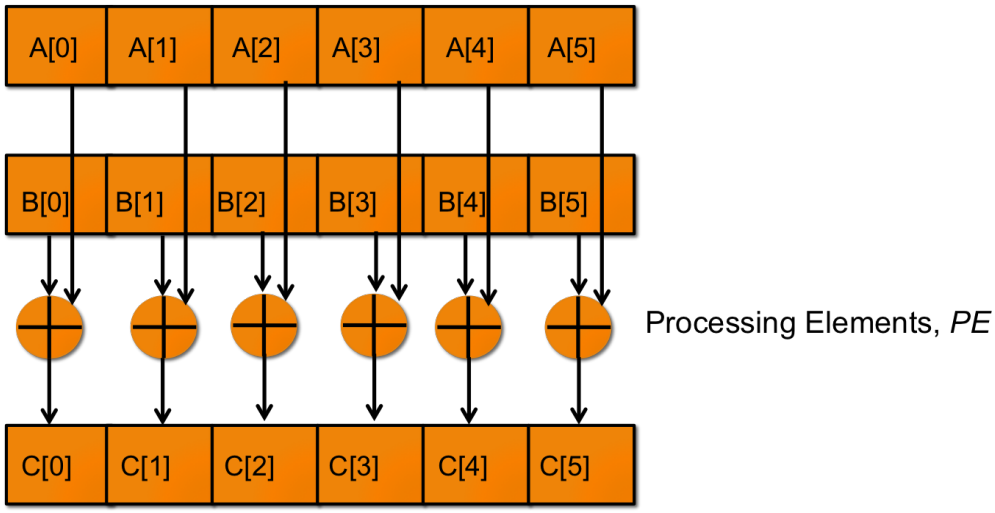


Figure 8.1: Addition of two vectors A and B to elucidate data parallelism. The processing elements, *PEs*, work independently to evaluate elements in C.

product results from other pertinent *PE*s to obtain the final result, c_{ij}. Ahead in this chapter, we study the parallelization of matrix-matrix multiplication on the GPGPU device.

There are several computationally-intensive mathematical operations used in engineering and science that offer data-parallelism. Some examples include reduction, prefix sum, scan, sorting, among many others. Not surprisingly, many scientific applications are composed of these computationally-intensive operations. By parallelizing these operations, programmers can achieve significantly high performance for their scientific codes and simulations.

Active Learning Exercise 2 – Perform a research on the following operations and explain how they offer data-parallelism: reduction, prefix sum, scan, and sorting.

8.2 CUDA program structure

In this section, we explore the structure of common CUDA programs. First, we explore a simple real-world example and then transfer the intuition to CUDA programs. Consider the example of multiple graders sharing the grading workload for a large class. Let us assume that the instructor collects the student assignments and distributes them equally to all of the graders. There are multiple scenarios

that can arise in this case. In Scenario *A*, the graders complete the grading job easily without any doubts and/or clarifications with respect to grading. In this scenario, the instructor gets the graded assignments expeditiously. In Scenario *B*, the graders may have questions on grading and they visit the instructor's office for clarification. Due to this instructor-grader communication, the grading is slower than Scenario *A*. In another Scenario *C*, the graders may choose to communicate with each other and avoid long trips to the instructor's office, thereby finishing the job faster than Scenario *B*. The structure of typical CUDA programs is no different than the structure of grading scenarios – in what follows, we describe the layout of a typical CUDA program.

Figure 8.2 illustrates the structure of a typical CUDA program, which has two primary interleaved sections namely, the host portion (highlighted in green) and the device portion (highlighted in orange). Depending on the application, these interleaved sections may be repeated several times. A single CPU thread executes the host portion, while the GPGPU device executes the device portion of the CUDA program. At the start of the program, the CPU host portion prepares the data to be executed on the GPGPU device. After the data preparation, the CPU host transfers the data to the GPGPU device memory via host-to-device (H2D) transfer operation, which is performed over the PCIe bus. After the GPGPU device portion finishes operating on the data, the processed data is transferred back to the CPU host memory via device-to-host (D2H) operation. A CUDA program may contain several interleaved host and device portions (similar to the multiple graders case discussed above); however a prudent programmer must be wary of communication costs incurred due to frequent H2D and D2H transfers. To maximize the performance of CUDA programs, it is recommended to minimize the host-device communications.

Let us investigate the structure of our first CUDA program. Listing 8.1 provides the complete CUDA program for vector-vector addition. Line 1 includes the `cuda.h` header file that provides GPGPU device-related functions. Inside the `main()` function, note the host and device pointer variables declaration in lines 14 and 15, respectively. It is recommended to provide the `h_` prefix for the host pointer variables and `d_` prefix for the device pointer variables. These prefixes enable programmers to avoid the accidental de-referencing of device pointers by the host and vice-versa, which cause the programs to break with error messages. The host portion of the CUDA program prepares the data for the GPGPU device execution (see Lines 18 through 29) and allocates the host and device memories for computations (Lines 19 though 25). For seamless programming, CUDA provides the `cudaMalloc()` function (similar to host's `malloc()` function) for

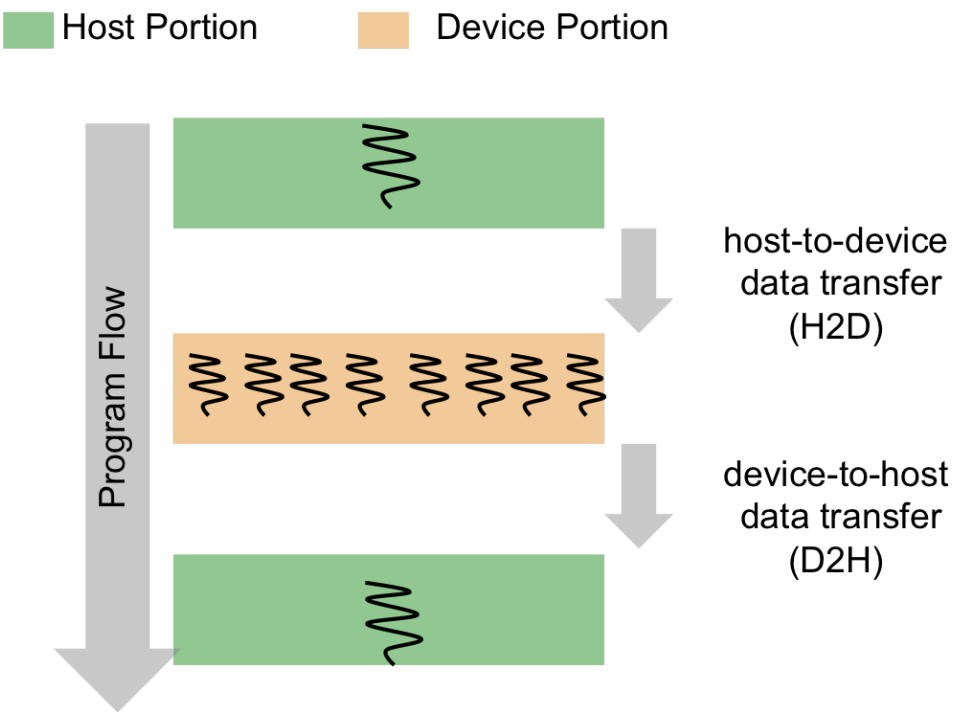


Figure 8.2: Program flow of a typical CUDA program interleaved with host portion (executed by a single CPU thread) and device portion (executed by several thousands of GPU threads). The host-to-device (H2D) and device-to-host (D2H) communications occur between the interleaved portions denoting data transfers from CPU-to-GPGPU and GPGPU-to-CPU, respectively.

allocating device pointers in the GPGPU device memory.

Once the host portion of the code finishes the necessary preprocessing steps, it initiates a host-to-device data transfer via the `cudaMemcpy()` function call (Lines 31 and 32). The `cudaMemcpy()` function inputs the destination pointer, source pointer, number of bytes to be transferred, and the data transfer direction (host-to-device, device-to-host, etc.). Readers are encouraged to pay special attention to `cudaMemcpy()` function and its parameters. Incorrect function parameters can also lead to incorrect referencing of pointers. In Listing 8.1 on lines 31 and 32, the destination pointer arguments are the device pointers `d_a` and `d_b`, respectively, the source pointer arguments are the host pointers `h_a` and `h_b`, respectively, and the data transfer direction is specified by the `cudaMemcpyHostToDevice` flag, denoting host-to-device data transfer.

Listing 8.1: An example CUDA program illustrating vector-vector addition. Note the interleaved CPU-host and GPGPU device portions. Host-device communications occur between the interleaved host and device portions of the CUDA program.

```
1. #include <cuda.h>
2. #include <stdio.h>
3. //Device kernel
4. __global__
5. void gpu_kernel(int *d_a, int *d_b,int *d_c, int vec_size){
6.    int tid = threadIdx.x + blockIdx.x*blockDim.x;
7.    if(tid <vec_size) {
8.       d_c[tid] = d_a[tid] + d_b[tid];
9.    }
10. } //end device kernel
11. int main(int argc, char **argv){
12.   //declare variables
13.    int i,vec_size;
14.    int *h_a,*h_b,*h_c; //data pointers for host-section
15.    int *d_a,*d_b,*d_c; //data pointers for device-section
16.   //Host-Section prepares the data
17.   vec_size=1000;
18. //Host-portion prepares the host data and allocates device pointers
19.   h_a=(int *)malloc(sizeof(int)*vec_size);
20.   h_b=(int *)malloc(sizeof(int)*vec_size);
21.   h_c=(int *)malloc(sizeof(int)*vec_size);
22.   //Allocate GPGPU device pointers
23.   cudaMalloc((void **)&d_a,sizeof(int)*vec_size);
24.   cudaMalloc((void **)&d_b,sizeof(int)*vec_size);
25.   cudaMalloc((void **)&d_c,sizeof(int)*vec_size);
26.   //Host-portion prepares the data
27.   for (i=0; i<vec_size; i++)   {
28.     h_a[i]=i; h_b[i]=i;
29.   }
30.   //CPU host transfers the data to GPGPU device memory
31.   cudaMemcpy(d_a,h_a,sizeof(int)*vec_size,cudaMemcpyHostToDevice);
32.   cudaMemcpy(d_b,h_b,sizeof(int)*vec_size,cudaMemcpyHostToDevice);
33.   //CPU host invokes the GPGPU device portion
34.   gpu_kernel<<<1, 1000>>>(d_a,d_b,d_c,vec_size);
35.   //GPGPU device transfers the processed data to CPU host
36.   cudaMemcpy(h_c,d_c,sizeof(int)*vec_size,cudaMemcpyDeviceToHost);
37.   //CPU host-portion resumes operation
38.   for(i=0; i<vec_size; i++){
39.     printf(``\n C[%d]=%d'',i,h_c[i]);
40.   }
41.   free(h_a);
42.   free(h_b);
43.   free(h_c);
44.   cudaFree(d_a);
45.   cudaFree(d_b);
46.   cudaFree(d_c);
47.   return 0;
48. }
```

After transferring the data to the GPGPU device, the host portion invokes the GPGPU device kernel in Line 34. A device kernel is a GPGPU device function that is callable from the host and executed by the GPGPU device. A device kernel

invocation is also referred to as a *kernel launch*. The calling name (`gpu_kernel`) specifies the name of the device kernel. The angular brackets (`<<< >>>`) specify the GPGPU device execution configuration, which mainly consists of the number of thread blocks and the number of threads per block to operate on the input data. We discuss threads and thread blocks in detail in Section 8.4. In this example when the `gpu_kernel` is launched, one thread block containing 1000 CUDA threads are created that execute the kernel function concurrently. More details on GPGPU device kernels and execution configuration appear in Section 8.5. The lines 4-9 are executed as the device portion of the code on the GPGPU device. The `__global__` keyword specifies that the following function (`gpu_kernel` in our case) is a device kernel. The in-built variables `threadIdx`, `blockIdx`, and `blockDim` in Line 6 enable programmers to access the threads' global indices. In this program, a thread with index `tid` operates on `tid`-th element of the vectors *A*, *B*, and *C*. It should be noted that GPGPU device kernel calls are asynchronous, meaning that after the kernel launch, the control immediately returns to the host portion. In this program after the kernel launch, the host portion invokes the `cudaMemcpy()` function (Line 36), which waits for all of the GPGPU threads to finish the execution. After the GPGPU device finishes execution, the control returns to line 36 where the device transfers the processed data (vector *C*) to the host. Note that in this device-to-host transfer, the host pointer (`h_c`) is the destination, the device pointer (`d_c`) is the source, and the direction of the data transfer is device-to-host denoted by the `cudaMemcpyDeviceToHost` flag. The lines 44 through 46 release the device memory variables via the `cudaFree()` function call.

8.3 CUDA compilation flow

Now that we understand the structure of CUDA programs, let us study how CUDA programs are compiled and a single executable is generated in a Linux-based environment. NVIDIA's `nvcc` compiler facilitates the splitting, compilation, preprocessing, and merging of CUDA source files to create an application's executable. Although the `nvcc` compiler enables transparent code compilation, an understanding of the compilation flow can enable further performance improvement. The `nvcc` compiler in the Linux environment recognizes a selected set of input files given in Table 8.2. In what follows, we study a high-level compilation flow of CUDA source programs.

Figure 8.3 provides a high-level abstraction of the CUDA compilation pro-

Table 8.2: List of input files recognized by the `nvcc` compiler in Linux-based environment.

Input file type	Description
.cu	CUDA source file containing host and device portions
.c	C source file
.cpp, .cc, .cxx	C++ source file
.gpu	Intermediate device-code only file
.o	Object file
.a	Library file
.so	Shared object files (not included in executable)
.res	Resource file

cess. The `nvcc` compiler, in conjunction with a compatible host code compiler such as `gcc/g++`, splits the compilation of CUDA source programs into two trajectories namely, the host trajectory and the device trajectory. These trajectories are not completely disjoint and often interact with each other via intermediate 'stub' functions. The host trajectory extracts the host code, host stub functions (functions that set up the kernel launch when the device kernel is invoked by the host), and compiles the host code to produce the `.o` object file. The device trajectory includes multiple steps such as device code extraction, host stub extraction, and device code optimization. The `nvopenacc` command inputs the intermediate compilation files (`.cpp3.i`) to produce the virtual architecture assembly file (`.ptx`) that contains a generic device instruction set. Next, the `ptxas` assembly command generates the `.cubin` file: the real architecture binary for a specific GPGPU device. The fatbinary stage combines multiple `.cubin` files (each targeting a different GPGPU device) into a `.fatbin` binary file. This binary file is ultimately linked with the host `.o` object file to create the final executable file, `a.out`. When `a.out` is executed, an appropriate `.cubin` file is selected from `.fatbin` for the target GPGPU device.

The CUDA toolkit documentation [7] provides a highly detailed explanation of the compilation process. The `nvcc` compiler also offers programmers with several compiler switches to control the code generation. Here, we only discuss two important switches: `--gpu-architecture` and `--gpu-code`. These switches allow for the GPGPU device architecture evolution. Before describing the role of these compiler switches, let us define the term *Compute Capability*.

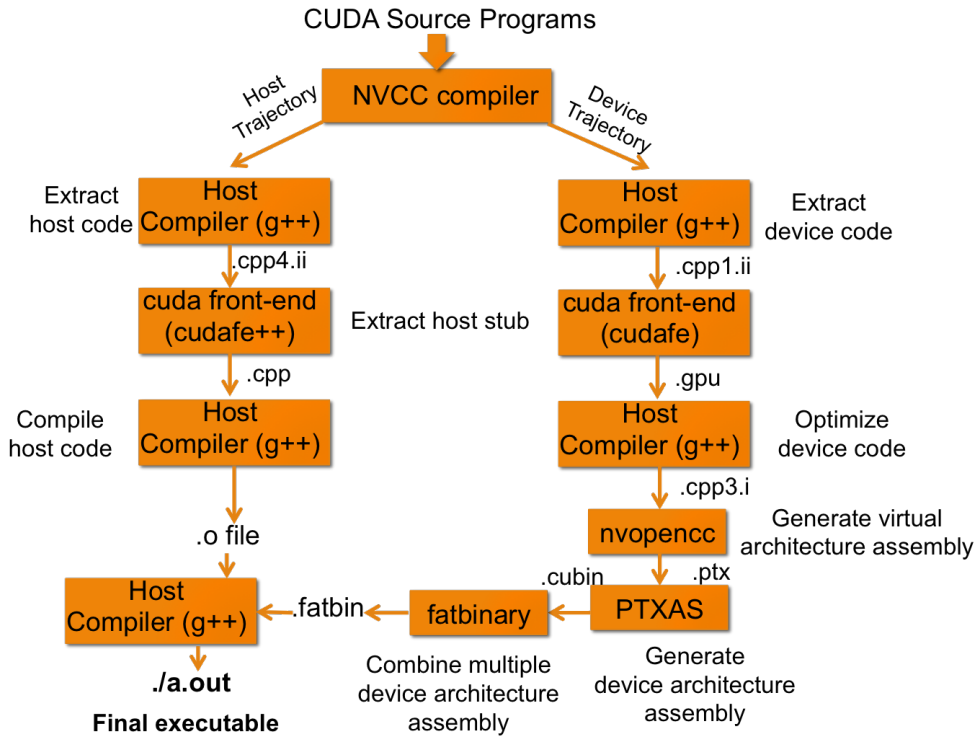


Figure 8.3: A high-level abstraction of `nvcc` compilation process. The `nvcc` compiler breaks the compilation process into two trajectories: host trajectory and device trajectory.

The Compute Capability of a device is represented by a version number that identifies the supported hardware features of the GPGPU device. The Compute Capability is used during the compilation process to determine the set of instructions for the target GPGPU device. The purpose of the above-mentioned `nvcc` compiler switches is as follows.

`--gpu-architecture` (short: `-arch`): This switch enables the selection of a virtual architecture, thereby controlling the output of the `nvopencc` command. A virtual architecture is a *generic* set of instructions for the virtual GPGPU device with the desired compute capabilities. By itself, the virtual architecture does not represent a specific GPGPU device. Some example values of `--gpu-architecture` switch are: `compute_20` (Fermi support); `compute_30` (Kepler support); `compute_35` (recursion via dynamic parallelism); `compute_50` (Maxwell support).

`--gpu-code` (short: `-code`): The switch enables the selection of a specific

Table 8.3: Examples of code generation using `--gpu-architecture` and `--gpu-code` switches.

Example	**Description**
`nvcc vector.cu --gpu-architecture=compute_30 --gpu-code=sm_30,sm_35`	The `fatbinary` includes two cubins; one cubin corresponding to each architecture.
`nvcc vector.cu --gpu-architecture=compute_30 --gpu-code=compute_30, sm_30,sm_35`	The same as the above with the inclusion of PTX assembly in the `fatbinary`.
`nvcc vector.cu --gpu-architecture=compute_30 --gpu-code= sm_20,sm_30`	Fails because `sm_20` is lower than the virtual architecture `compute_30`

GPGPU device (the actual GPU architecture). Some examples values include: `sm_20` (Fermi support); `sm_30` (Kepler support); `sm_35` (recursion via dynamic parallelism); `sm_50` (Maxwell support).

In what follows, we outline the general guidelines used to set values of the above mentioned compiler switches for different types of code generation. The `--gpu-architecture` switch takes a specific value, whereas the `--gpu-code` switch can be set to multiple architectures. In such a case, `.cubin` files are generated for each architecture and included in the `fatbinary`. The `--gpu-code` switch can include a single virtual architecture, which causes the corresponding PTX code to be added to the `fatbinary`. The NVIDIA documentation suggests keeping the value of `--gpu-architecture` switch as low as possible to maximize the number of actual GPGPU devices. The `--gpu-code` switch should preferably be higher than the selected virtual architecture. Table 8.3 provides several compilation examples for code generation. We encourage readers to peruse the Nvidia software development kit (SDK) for sample Makefiles and adapt them for their respective applications and GPGPU devices.

Active Learning Exercise 3 – Write a compilation command for generating a `fatbinary` with PTX included for Fermi and Kepler architectures.

8.4 CUDA thread organization

A CUDA program follows Single Program, Multiple Data (SPMD) methodology where several thousands of threads work concurrently to execute the same kernel function on different data elements. However, different groups of threads may be executing different sections of the same CUDA kernel. To enable CUDA threads to access the relevant data elements upon which to operate, it is imperative to fully understand the CUDA thread organization. The CUDA threads are organized in a two-level hierarchy of *grids* and *blocks*. A grid is a three-dimensional collection of one or more blocks and a block is a three-dimensional collection of several threads. When a kernel function is called, a grid containing multiple thread blocks is launched on the GPGPU device (Figure 8.4). As shown in the same figure, when the kernel function `Kernel1` is called, a two-dimensional grid of thread blocks (2 blocks each in x and y dimensions) is launched on the GPGPU device. In this example, each thread block is a two-dimensional arrangement of threads with two threads in both the x and y dimensions. The `Kernel2` function call launches a CUDA grid with two thread blocks, where each thread block is a one-dimensional arrangement of five threads. For illustration purposes, the above examples work with only four or five threads per block. Readers should note that GPGPU devices require a minimum number of threads per block depending on the Compute Capability.

First, let us investigate CUDA grids. As mentioned earlier, each grid is a three-dimensional arrangement of thread blocks. When the kernel function is launched, the first parameter in execution configuration, <<<**dimGrid, ..**>>>, specifies the dimensions of the CUDA grid. The size of grid dimensions depends on the Compute Capability of the GPGPU device. In CUDA programs, the dimensions of the grids can be set using the C structure, `dim3`, which consists of three fields: `x`, `y`, and `z` for x, y, and z dimensions, respectively. By setting the dimensions of CUDA grids in the execution configuration, we automatically set the values of x, y, and z fields of the predefined variable, `gridDim`. This variable is used in the kernel function to access the number of blocks in a given grid dimension. The blocks in each dimension are then accessed via the predefined variable, `blockIdx`, which also contains three fields: `x`, `y`, and `z`. The variable `blockIdx.x` takes on values ranging from 0 to `gridDim.x-1`; `blockIdx.y` takes on values ranging from 0 to `gridDim.y-1`; and `blockIdx.z` takes on values ranging from 0 to `gridDim.z-1`. Table 8.4 provides examples of CUDA grid initialization using the `dim3` structure and illustrates the values of `gridDim` and `blockIdx` variables. Note that the unused dimensions in the `dim3` structure

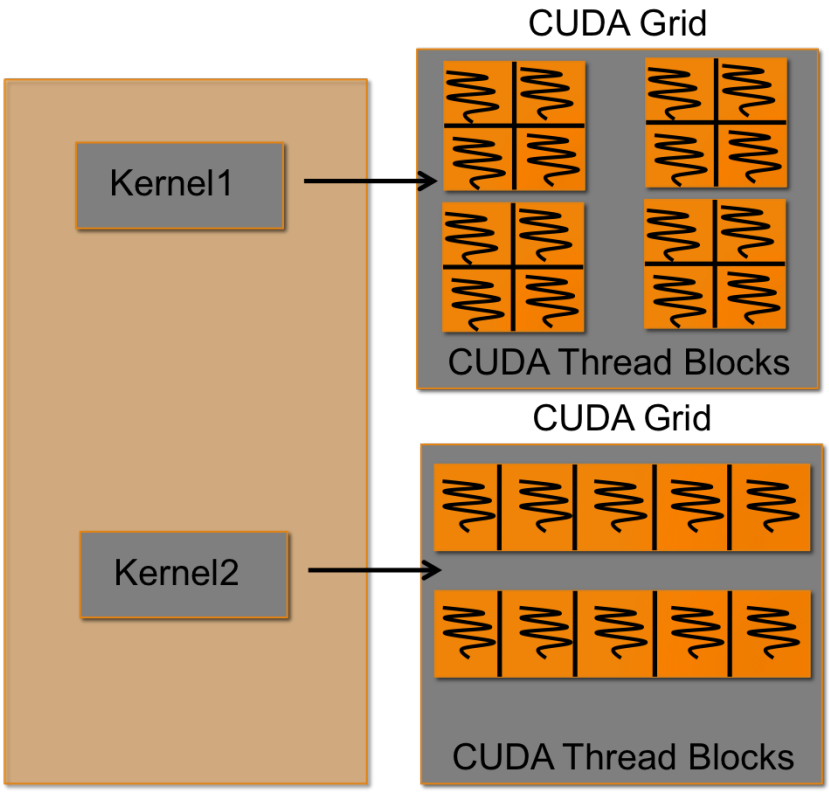


Figure 8.4: Two examples of CUDA grids and thread blocks. When `Kernel1` is called, it launches a 2×2 grid of 2×2 thread blocks. When `Kernel2` is called, it launches a 1D grid with two 1D thread blocks with each block containing 5 threads.

are set to one.

The dimensions of a CUDA grid can also be set at runtime. For instance, if a programmer requires 256 threads per block to work on `n` elements, the dimensions of the grid can be set as:

`<<<dimGrid(round_up(n,256)),..>>>`. Note that `round_up()` function is required to launch enough thread blocks to operate on all of the `n` elements.

Active Learning Exercise 4 – Initialize a three-dimensional CUDA grid with two blocks in each dimension. Give the values of pertinent predefined variables.

Next, we turn our attention to CUDA thread blocks. As mentioned before, the CUDA thread blocks are three-dimensional arrangements of threads. The second parameter in the execution configuration, `<<<dimGrid,`**`dimBlock`**`,..>>>`, specifies the dimensions of a single thread block. Similar to grids, the thread

Table 8.4: Examples of CUDA grid initialization using `dim3` structure. The corresponding values (range of values) of `gridDim` and `blockIdx` variables are shown.

Example	Description	`gridDim` variable			`blockIdx` variable		
		x	y	z	x	y	z
`dim3 dimGrid1(32,1,1)`	1D grid with 32 thread-blocks	32	1	1	0-31	0	0
`dim3 dimGrid2(16,16,1)`	2D grid with 16 blocks in x and y dimensions	16	16	1	0-15	0-15	0
`dim3 dimGrid3(16,16,2)`	3D grid with 16 blocks in x and y dimensions and 2 blocks in z dimension	16	16	2	0-15	0-15	0-1

block dimensions can also be set using the `dim3` structure. It should be noted that the total number of threads in a block should not exceed 1024. Once the block dimensions are set, the `x`, `y`, and `z` fields of the in-built variable, `blockDim` are initialized. Each field of `blockDim` variable denotes the number of threads in x, y, and z dimensions. Each thread in a given thread block is then accessed using the predefined variable, `threadIdx`. Akin to the `blockIdx` variable, the `threadIdx` variable has three fields namely, `threadIdx.x` varying from 0 to `blockDim.x-1`, `threadIdx.y` varying from 0 to `blockDim.y-1`, and `threadIdx.z` varying from 0 to `blockDim.z-1`. Table 8.5 provides examples of block dimension initialization and the corresponding values of `blockDim` and `threadIdx` variables.

Active Learning Exercise 5 – Initialize a 2D CUDA block with 16 threads in each dimension. Give the values of pertinent predefined variables.

As discussed before, the maximum grid and block dimensions depend on the Compute Capability of the GPGPU device. It is always a good idea to verify these values for newer architectures. Table 8.6 provides the maximum device specific values for Compute Capability 3.x, 5.x, and 6.x devices.

Active Learning Exercise 6 – Investigate the device specific values of earlier compute capabilities, i.e. 1.x and 2.x. Also provide one GPGPU device from these compute capabilities. What are the significant changes in device specific

Table 8.5: Examples of CUDA block initialization using `dim3` structure. The corresponding values (range of values) of `blockDim` and `threadIdx` variables are shown.

Example	Description	`blockDim` variable			`threadIdx` variable		
		x	y	z	x	y	z
`dim3 dimblock1(32,1,1);`	1D block with 32 threads	32	1	1	0-31	0	0
`dim3 dimblock2(32,32,1);`	2D block with 32 threads in x and y dimensions	32	32	1	0-31	0-31	0
`dim3 dimblock3(32,32,2);`	Incorrect. The number of threads in the block exceeds 1024.	-	-	-	-	-	-

values for Compute Capability 2.x onwards? Make a note on how these changes influence the GPGPU programming.

8.5 Kernel: Execution configuration and kernel structure

As readers may recall, several thousands of threads created by the programmer in a CUDA program concurrently execute a special device function, the kernel. The host portion of the CUDA program asynchronously calls the CUDA kernel, meaning that the control immediately returns to the host portion after the kernel launch. During the kernel execution, the host portion may perform some computations (thereby overlapping computations) or may choose to wait for the GPGPU device to finish operating on the data. An example of kernel launch is as follows:

```
gpu_kernel <<<dimGrid, dimBlock>>> (arg1, arg2,..,argN);
```

In the above statement, the GPGPU device kernel named `gpu_kernel` is executed by all of the threads created in the CUDA program. The number of threads created is a function of the kernel execution configuration specified by the `dimGrid` and `dimBlock` (`dim3` type) variables configured by the programmer (see Tables 8.4 and 8.5 for examples). As discussed in the foregoing section,

Table 8.6: Limitations on device specific parameters for Compute Capability 3.x, 5.x, and 6.x devices.

Device Parameter	**Maximum Number**
Maximum number of grid dimensions	3
Grid maximum in x dimension	$2^{31} - 1$
Grid maximum in y and z dimensions	$2^{16} - 1$
Maximum number of block dimensions	3
Block maximum in x and y dimensions	1024
Block maximum in z dimension	64
Maximum threads per block	1024
Example GPGPU device (3.x)	Kepler GK110
Example GPGPU device (5.x)	Maxwell GM200
Example GPGPU device (6.x)	Pascal GP102

the `dimGrid` variable specifies the number of CUDA blocks arranged in x, y, and z dimensions of a CUDA grid, whereas the `dimBlock` variable specifies the number of CUDA threads arranged in x, y, and z dimensions in a CUDA block. A general procedure for setting an execution configuration is follows.

1. Set the thread block dimensions and the number of threads in each dimension such that the total number of threads in a block does not exceed 1024. Pay attention to GPGPU device specific limits (see Table 8.6).

2. Calculate the number of thread blocks required in each grid dimension.

Once the execution configuration is set and the kernel is launched, it is customary for each thread to 'know' its local and global thread identification numbers (IDs). It is via these thread IDs that different threads access their respective portions of the data. As discussed in Section 8.4, threads can access their IDs inside the device kernel function using in-built variables: `gridDim`, `blockDim`, `blockIdx`, and `threadIdx`. These variables are set when the execution configuration is passed to the kernel during the kernel launch. The methodology of setting the execution configuration usually depends on the type of parallel patterns in an application. Simple parallel patterns such as vector-vector addition, prefix sum, etc. may only require one-dimensional execution configuration. Whereas more complex patterns such as matrix-matrix multiplication, two-dimensional image convolution, etc. intuitively lend themselves to two-dimensional execution configuration. More complex applications that operate on three-dimensional data are parallelized using a three-dimensional execution configuration. In what follows, we use two example parallel patterns illustrating one-dimensional and two-dimensional execution configurations, namely vector-vector addition and matrix-matrix multiplication. We study how the execution configuration is set and the threads are accessed inside the device kernel function for these two parallel patterns. These examples help us build our intuition for one- and two-dimensional grids and blocks, which can be easily extended to three-dimensional execution configuration.

Consider addition of two vectors A and B, each containing n elements. The result of addition is stored in vector C as illustrated by Figure 8.1. We use 1D blocks and grids for this case, given that our working arrays A, B, and C are one-dimensional arrays. An example execution configuration with 256 threads per block appears in Listing 8.2.

Listing 8.2: The example illustrates an execution configuration with 256 threads per block for vector-vector addition. The example also shows how a thread accesses its global index/identifier (ID) in the CUDA grid.

```
//Auxiliary C function for rounding up
int round_up(int numerator,int denominator) {
return (numerator+denominator-1)/denominator;
}

//Inside main

// Step 1: Set the block configuration
1. dim3 dimBlock(256, 1, 1);
//Step 2: Set the grid configuration
2. dim3 dimGrid (round_up(n,256), 1, 1);
//GPU kernel call
3. gpu_kernel <<<dimGrid, dimBlock>>>(A, B, C);
:
:
//Inside gpu_kernel function (device portion)
:
//The local thread ID in a given block
A. local_tid = threadIdx.x;
//The global thread ID in the entire grid
B. global_tid = local_tid + blockIdx.x*blockDim.x;
:
//Array access
AA. C[global_tid] = A[global_tid] + B[global_tid];
```

In Listing 8.2, Line 1 sets the x dimension of the thread block to 256 and the remaining unused fields (y and z) are set to one. In Line 2, the x dimension of the grid is set to `round_up(n,256)`, whereas the unused y and z dimensions are set to 1. The rounding up operation (using `round_up()`) is performed to create enough number of thread blocks to execute all of the n data elements. Inside the `gpu_kernel` function, Line A performs the access of the local thread ID, i.e. the thread's ID in its block. Line B shows how a thread accesses its global thread ID. In general, the global thread ID in any dimension follows the formula: `global_tid = local_tid + offset`. In this case, the offset equals `blockIdx.x*blockDim.x` and local ID equals `threadIdx.x`. Each thread then accesses a unique element of vectors A, B, and C using the global thread ID (`global_tid`) in Line AA. Figure 8.5 illustrates the global thread ID access discussed above.

Next, we consider the example of matrix-matrix multiplication to illustrate two-dimensional execution configuration. For simplicity, assume multiplication of two 2D matrices $A_{n\times n}$ and $B_{n\times n}$ of dimensions $n \times n$ each. The result of this multiplication is stored in another 2D matrix of the same dimensions, $C_{n\times n}$. For the purpose of illustration, assume 16×16 as the thread block dimensions. Read-

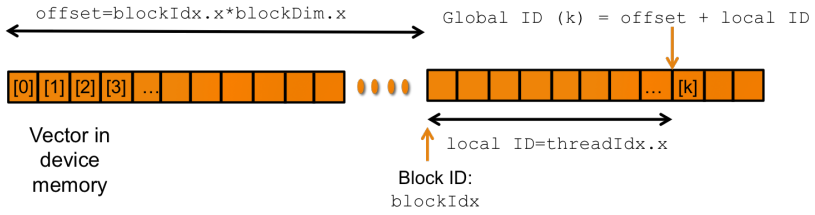


Figure 8.5: The illustration shows how a thread accesses its global ID and the corresponding data element in the vector.

ers should recall that the number of threads per block should not exceed 1024. The `dim3` type variables, `dimGrid` and `dimBlock`, are configured as shown in Listing 8.3.

Listing 8.3: Configuration of `dimGrid` and `dimBlock` in the host portion; and access of local and global thread IDs in the device portion.

```
//Preparing the execution configuration inside host portion of the code
// Step 1: Set the block configuration
1. dim3 dimBlock(16, 16, 1);
//Step 2: Set the grid configuration
2. dim3 dimGrid (round_up(n,16), round_up(n,16), 1);
//GPU kernel call
3. gpu_kernel <<<dimGrid, dimBlock>>>(A, B, C);
:
:
//Inside gpu_kernel function (device portion)
:
//The local thread ID in x-dimension in a given block
A. local_tidx = threadIdx.x;
//The local thread ID in y-dimension in a given block
B. local_tidy=threadIdx.y;
//The global thread ID in x-dimension in the entire grid
C. global_tidx = local_tidx + blockIdx.x*blockDim.x;
//The global thread ID in y-dimension in the entire grid
D. global_tidy = local_tidy + blockIdx.y*blockDim.y;
:
//Array access
AA. a=A[global_tidx][global_tidy];b=B[global_tidx][global_tidy];
```

In the example shown in Listing 8.3, a `dim3` structure (`dimBlock`) for 2D CUDA block is declared with 16 threads in x and y dimensions, respectively; the unused z dimension is set to 1. Because the matrices are square with n elements in x and y dimensions, the CUDA grid consists of `round_up(n,16)` number of CUDA blocks in x and y dimensions; the unused z dimension is set to 1 (Line 2). Inside the `gpu_kernel`, the local and global thread IDs in x and y dimensions are accessed as shown in lines A through D. The global element access using

the global thread IDs is elucidated in Line AA. Figure 8.6 illustrates the above discussed concept for two-dimensional thread ID access.

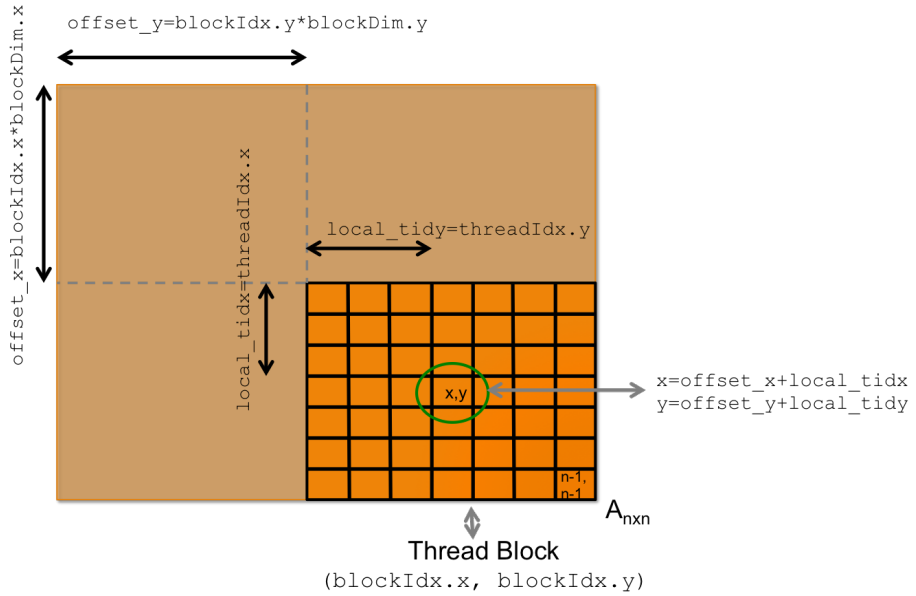


Figure 8.6: The illustration shows how a thread accesses its global 2D ID (x,y) and the corresponding data element (x,y) in a two-dimensional matrix, $A_{n \times n}$.

In the foregoing examples and parallel patterns similar to them, readers should ensure that the threads cover all of the data elements and the number of idle threads is minimized. For instance, consider an example of addition of two vectors with 1000 elements each. A choice of 256 threads per block results in four thread blocks, thereby creating 1024 threads for the entire application. Because the threads with global IDs 0 through 999 operate on the corresponding data elements 0 through 999, the threads with IDs 1000 through 1023 remain idle. Similarly, a choice of 200 threads per block results in 5 thread blocks with no idle threads. However, there is more to execution configuration than simply creating sufficient number of threads. The number of threads per block and thread blocks affect the number of concurrent thread groups (a group of 32 concurrent threads is called a warp) active on a streaming multiprocessor. This concept is discussed in detail in Section 8.6.

Active Learning Exercise 7 – Create a 2D grid with 2D blocks for operation on an image of size 480×512. Elucidate, how each thread accesses its ID and its

corresponding pixel element (x,y). How can you extend this process for a color image $480 \times 512 \times 3$ where the third dimension corresponds to the red, green, and blue (RGB) color channels?

8.6 CUDA memory organization

The GPGPU devices are throughput-oriented architectures, favoring compute-logic units over memory units. The GPGPU device's main memory (also called the *device memory*) is usually separate from the GPGPU device. Consequently, most of the CUDA programs observe a performance bottleneck due to frequent device memory accesses. Therefore, programmers pursuing high-performance on GPGPU devices must have a deep understanding of the device memory hierarchy. A sound understanding of the CUDA memory hierarchy enables programmers to perform optimizations effectively. In what follows, we discuss the device memory hierarchy with respect to the CUDA programming model.

Figure 8.7 shows an abstract representation of a CUDA GPGPU device with its streaming multiprocessors interacting with the device memory. Henceforth, we refer to this memory hierarchy as the CUDA memory hierarchy.

As shown in Figure 8.7, a GPGPU device contains multiple streaming processors (SMs), each containing multiple CUDA cores. In a typical CUDA program, the thread blocks are launched on the SMs while the CUDA cores execute the threads in a thread block. The CUDA memory hierarchy follows a pyramid fashion from the fastest but smallest memory units to the slowest but largest memory units as under:

- **On-chip Registers ($\approx$ 32 K registers per SM)** – In a SM, each CUDA core has exclusive access to its own set of registers. The register accesses are blazingly fast, each access taking only one clock cycle. The lifetime of registers is the lifetime of a thread. The automatic variables in the CUDA kernel are allotted registers depending on the device's Compute Capability. The leftover registers spill into the device's local memory, which resides in the off-chip device memory.

- **On-chip Shared memory ($\approx$ 64 KB per SM)** – Further away from the registers is the shared memory shared by all of the CUDA cores in a SM. The accesses to shared memory are also fast; an access typically takes ≈ 30 clock cycles. The shared memory persists for the lifetime of a thread block.

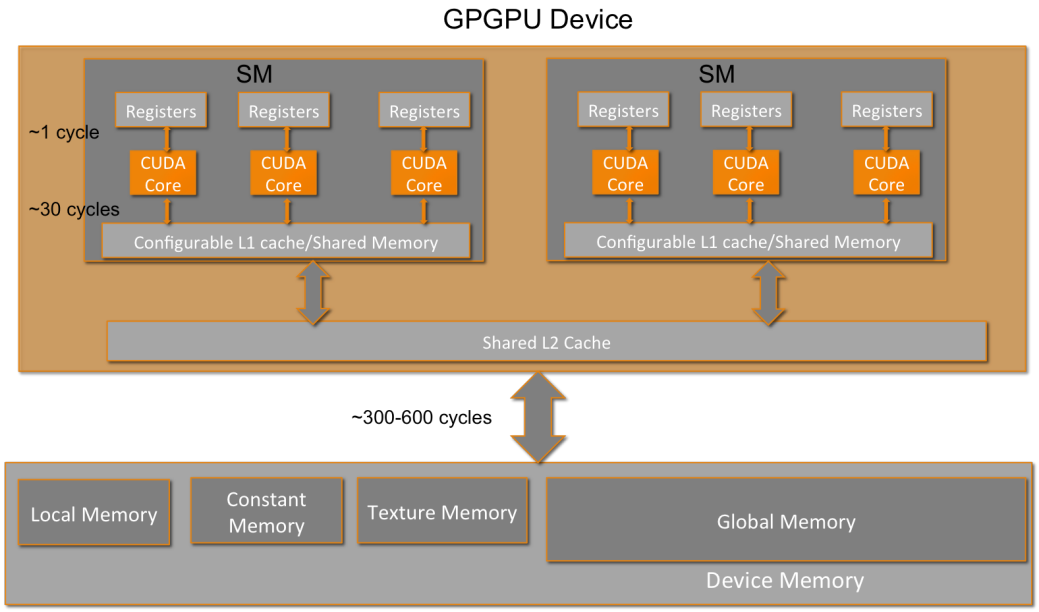


Figure 8.7: The CUDA memory hierarchy: At the lowest level, CUDA cores inside SMs have access to fast registers. All of the CUDA cores in a given SM have shared access to L1 cache/shared memory (fast but slower than registers). All the SMs share the L2 cache (if present). The farthest memory unit from the GPGPU device is the device memory, which consists of special memory units including local memory, cached constant memory, texture memory, and global memory.

- **Off-chip Device Memory (typically several GB)** – The largest and perhaps the most important memory unit of all is the GPGPU device memory, which resides in the off-chip random access memory (RAM). The device memory further consists of sub-units including:
  - **Local memory** for storing 'spilled' register variables.
  - **Cached constant memory** for storing constant values.
  - **Texture memory** with specialized hardware for filtering operations.
  - **Global memory** accessible to the entire GPGPU device via CUDA memory transfer functions.

  Accesses to the device memory typically take 300 to 600 clock cycles. However, a CUDA program can obtain significant performance boost due to

L1/L2 caches in recent GPGPU architectures. The device memory persists for the lifetime of the entire program.

In what follows, we explore registers, shared memory, constant memory, and the global memory in detail. The texture memory is operated via the Texture Object APIs and its usefulness is limited in general-purpose computing. Therefore, we skip the discussion on texture memory, although readers are encouraged to explore texture memory discussed in the CUDA programming guide [7].

8.6.1 Registers

As shown in Figure 8.7, each streaming multiprocessor has a set of on-chip registers that provide fast data access for various operations, which would otherwise consume several clock cycles due to frequent device memory accesses. Upon compilation with the `nvcc` compiler, the automatic variables declared in a CUDA kernel are stored in registers. However, not all automatic variables reap the benefits of registers because the GPGPU device's Compute Capability limits the maximum number of registers per streaming multiprocessor. If the number of requested registers in a CUDA kernel exceeds the device's capability, the leftover variables spill into the local memory (in off-chip device memory). Thereafter, any subsequent accesses to these variables may consume several hundreds of clock cycles. With recent advancements in the GPGPU device architecture and inclusion of caches, this performance artifact can be alleviated, however it is application-specific.

The number of registers used by threads in a CUDA kernel in conjunction with the number of threads per block also has a major performance implication – to what extent are the SMs occupied? The GPGPU devices realize parallelism via *warps*, a group of 32 concurrent threads. All of the threads in a warp execute the same instruction. Although, different warps may be executing different instructions of the same kernel. A streaming multiprocessor can have several active warps that can execute concurrently – when a set of warps executes memory instructions, the other set of warps performs useful computations. This level of concurrency amortizes the global memory latency. The *multiprocessor occupancy* is defined as the ratio of the number of active warps on SM to the maximum number of warps that can reside on a SM. Consequently, this ratio can at most be equal to 1 and a high value of multiprocessor occupancy is desirable to ensure high concurrency.

With the above background, let us study how the number of registers per thread and the number of threads per block affect the multiprocessor occupancy. Consider the Kepler K20Xm GPGPU device architecture, which belongs to Compute Capability 3.5. For this device, the maximum number of registers per SM is equal to 65536 and the maximum number of warps per SM is equal to 64. Using the `nvcc` compiler's `Xptxas` switch, we can determine the number of registers used and the amount of spill into the local memory. An illustration appears in Listing 8.4 where we compile a CUDA program, `convolve.cu`. As shown in the listing, the total number of registers per thread is 23 and there is no spill into the device's local memory.

Listing 8.4: An illustration of `nvcc` compiler's `Xptxas` option to determine the number of registers used and the amount of register spill into the local memory.

```
bash-4.2# nvcc -Xptxas -v -arch=sm_35 convolve.cu
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z8convolvePiiiPfiS_' for 'sm_35'
ptxas info    : Function properties for _Z8convolvePiiiPfiS_
0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 23 registers, 360 bytes cmem[0]
```

The multiprocessor occupancy for a given kernel is obtained via Equations 8.2 through 8.5.

$$registers\_per\_block = registers\_per\_thread \times threads\_per\_block \tag{8.2}$$

$$total\_blocks = \frac{(max\_registers\_per\_SM)}{(registers\_per\_block)} \tag{8.3}$$

$$resident\_warps = min(maximum\_warps, \frac{total\_blocks \times threads\_per\_block}{32}) \tag{8.4}$$

$$occupancy = \frac{resident\_warps}{maximum\_warps} \tag{8.5}$$

For the example in Listing 8.4, let us assume that the CUDA kernel is launched with 256 threads per block. The total number of registers per block is: $23 \times 256 = 588$ registers. For this example, a SM in theory can execute a total of 11 blocks. The total number of resident warps is $min(64, \frac{11 \times 256}{32}) = 64$, thereby yielding multiprocessor occupancy equal to 1. Equations 8.6 through 8.11 show

the calculations for multiprocessor occupancy if the threads in the above example were to use 100 registers.

$$registers\_per\_thread = 100; threads\_per\_block = 256 \tag{8.6}$$

$$registers\_per\_SM = 65536; maximum\_warps = 64 \tag{8.7}$$

$$registers\_per\_block = 100 \times 256 = 25600 \tag{8.8}$$

$$total\_blocks = \left\lfloor \frac{65536}{25600} \right\rfloor = 2 \tag{8.9}$$

$$resident\_warps = min(64, \frac{2 \times 256}{32}) = 16 \tag{8.10}$$

$$occupancy = \frac{16}{64} = 25\% \tag{8.11}$$

NVIDIA's CUDA occupancy calculator facilitates the occupancy calculations and elucidates the impact of varying thread block size and register count per thread on the multiprocessor occupancy. We discuss the occupancy calculator in detail in Section 8.7.

Active Learning Exercise 8 – For a Compute Capability device 3.0, the `nvcc` compiler reports a usage of 50 registers per thread. If the thread block size is 512, what is the multiprocessor occupancy? Make sure to use the NVIDIA GPU data for the device related constants (maximum registers per SM, warp size, maximum number of warps per SM, etc.). Will the occupancy be any better if the kernel were to use 128 threads per block?

8.6.2 Shared memory

NVIDIA GPGPU devices offer 64 KB on-chip shared memory that is used to cache frequently accessed data. The shared memory is slower than registers ($\approx$ 30 cycles per access versus 1 cycle per access for registers). However unlike registers, shared memory is accessible to all the threads in a thread block. The shared memory space is commonly used for thread collaboration and synchronization. These accesses, if performed via global memory, would typically consume several hundreds of clock cycles, thereby reducing the performance.

The kernel functions should be 'aware of' whether the variables are located in the device memory or in the shared memory. Programmers can statically allocate shared memory inside the kernel using the `__shared__` qualifier. Some examples of static shared memory allocation appear in Table 8.7. In the first example, a simple shared memory variable, `a` is declared. Example 2 shows how a 2D shared memory variable is declared inside a kernel function. All of the threads in a thread block have access to this 2D shared memory variable. Example 2 also shows how local threads in a thread block load the corresponding global data element into this shared variable. The last example shows an incorrect way of dynamically allocating a shared memory variable.

It is also possible to dynamically allocate variables in the shared memory. The third parameter of execution configuration (the first two parameters are for specifying the dimensions of grid and thread blocks, respectively) specifies the size of the shared memory to be dynamically allocated inside the kernel function. Additionally, the dynamic shared memory variable inside the kernel function is declared with the `extern` qualifier. For example, consider that the `BLOCKSIZE` parameter is determined at runtime – in this case, example 3 in Table 8.7 for allocating array `A` will not work. Programmers can specify the size of the shared memory in the execution configuration during the kernel call as shown in Listing 8.5.

Note that it is also possible to perform multiple dynamic shared memory allocations by specifying the combined size of required arrays in the execution configuration. Inside the kernel function, a single shared memory array is used with appropriate offsets (using array sizes) to access the individual shared memory arrays.

Listing 8.5: An illustration of dynamic shared memory allocation by specifying the amount of memory to be allocated in the execution configuration. The corresponding shared memory variable declaration has `extern` qualifier.

```
__global__void kernel(kernel-args) {
:
extern __shared__ float A[];
:
}
int main () {
:       kernel<<<dimGrid, dimBlock, sizeof(float)*BLOCKSIZE>>>(kernel-args);
:
}
```

Next, we study how threads within a thread block synchronize their accesses to the shared memory for thread collaboration. The threads in a thread block can synchronize via the `__syncthreads()` function, which provides a barrier for all of

Table 8.7: Examples of CUDA shared memory declaration.

Example	Syntax	Description
1	`__shared__ float a;`	The variable `a` is allocated in shared memory and is accessible to all threads inside a thread block.
2	`__shared__ float A[BLOCKSIZE][BLOCKSIZE]; //All threads load a value tidx=threadIdx.x; tidy=threadIdx.y; global_tidx= tidx+blockIdx.x *blockDim.x; global_tidy= tidy+blockIdx.y *blockDim.y; A[tidx][tidy]= global_A[global_tidx] [global_tidy];`	A two-dimensional array `A` is declared in the shared memory. The dimensions are `BLOCKSIZE` x `BLOCKSIZE` where `BLOCKSIZE` is the number of threads per block. All of the threads inside the thread block can access this array. This type of allocation is usually performed when each thread inside a thread block loads a value from the device global memory to shared memory, thereby optimizing the global memory bandwidth.
3	`__shared__ float *A; A=(float *)malloc (sizeof(float) *BLOCKSIZE);`	Incorrect because array `A` is not static. See text for dynamic shared memory allocation.

the threads in a thread block. Unless all the threads in a thread block finish executing the code preceding the `__syncthreads()`, the execution does not proceed ahead. This concept is illustrated by Figure 8.8. More on `__syncthreads()` function appears in Section 8.7 where we discuss shared memory optimization for algorithms that re-use the data (matrix-matrix multiplication for instance).

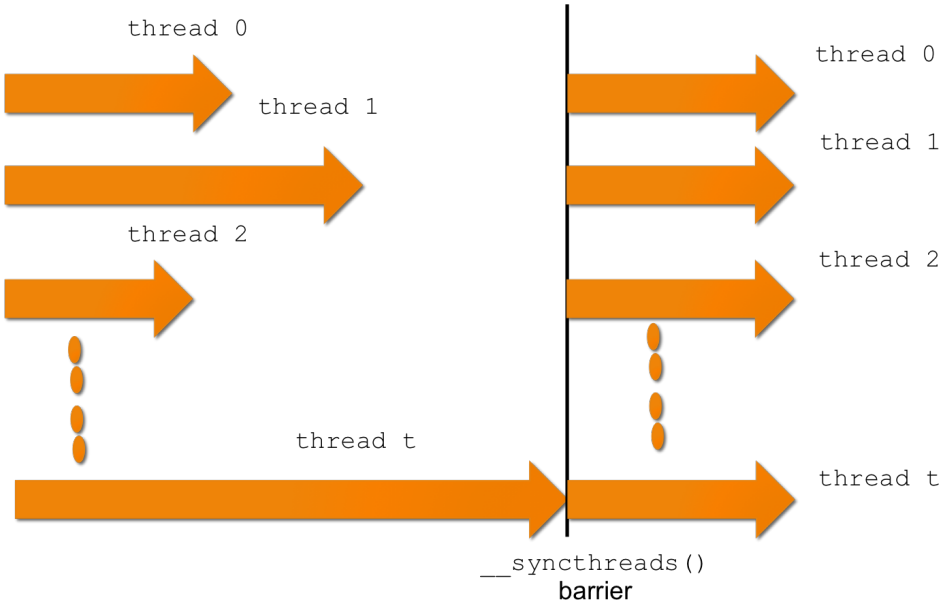


Figure 8.8: Threads `0` to `t` inside a thread block synchronizing via the `__syncthreads()` function. All of the preceding statements before the `__syncthreads()` statement must be executed by all the threads in a thread block.

Active Learning Exercise 9 – Declare a `BLOCKSIZE` sized shared memory variable called `mask` inside of a CUDA kernel. Outline the methodology for allocating shared memory space for the shared variable, `mask`.

Active Learning Exercise 10 – In the foregoing section, we mentioned a method of allocating multiple shared memory variables inside a CUDA kernel. The methodology is as follows: a) Specify the overall shared memory size in bytes in the execution configuration. This step is similar to the dynamic shared memory allocation method. b) Declare a single `extern __shared__` variable in the CUDA kernel. c) Using the individual shared variable sizes as offsets, access the appropriate base addresses using the shared variable declared in Step b. Employ the outlined methodology to reserve a shared memory space for three variables:

`float A` (k elements), `float B` (l elements), and `float C` (m elements).

In addition to shared memory, there are other mechanisms that enable threads to communicate with each other. The preceding discussion examines how threads within a block synchronize using the shared memory and `__synchthreads()` function. The threads within a warp can also synchronize and/or communicate via warp vote functions and warp shuffle functions. As readers may recall, a warp is a group of 32 concurrent threads.

The vote functions allow *active* threads within a warp to perform reduce-and-broadcast operation. The active threads within a warp are all threads that are in the intended path of warp execution. The threads that are not in this path are disabled (inactive). The vote functions allow active threads to compare an input integer from each participating thread to zero. The result of comparison is then broadcast to all of the participating threads in the warp. The warp voting functions are as follows.

- `__all(int input)`: All participating threads compare `input` with zero. The function returns a non-zero value if and only if all active threads evaluate the `input` as non-zero.
- `__any(int input)`: The function is similar to `__any(input)`, however the function returns a non-zero value if and only if any one of the active threads evaluates the `input` as non-zero.
- `__ballot(int input)`: The function compares the `input` to zero on all active threads and returns an integer whose N^{th} bit is set when the N^{th} thread of the warp evaluates the `input` as non-zero.

The shuffle functions (`__shfl()`) allow all active threads within a warp to exchange data while avoiding shared memory all together. At a time, threads exchange 4 bytes of data; exchanges of 8 byte data is performed by calling shuffle functions multiple times. The exchanges are performed with respect to a thread's *lane ID*, which is an integer number from 0 to $warpSize - 1$. Some of the shuffle functions are as follows:

- `__shfl(int var, int srcLane,int width=warpSize)`: This function allows an active thread to look up the value of variable `var` in the source thread whose ID is given by `srcLane`. If the `width` is less than `warpSize` then each subsection of the warp acts as a separate entity with starting lane ID of 0. If `srcLane` is outside the $[0 : width - 1]$, then the function calculates the source as $srcLane\%width$.

- `__shfl_up(int var,unsigned int delta,int width=warpSize)`: The function calculates the lane ID of the source thread by subtracting `delta` from the current thread's lane ID and returns the value `var` held by the source thread. If the `width` is less than `warpSize` then each subsection of the warp acts as a separate entity with starting lane ID of 0. The source index does not wrap around the value of `width`, therefore lower `delta` lanes are unchanged.

- `__shfl_down(int var,unsigned int delta,int width=warpSize)`: This function is similar to `__shfl_up()` function, except that `__shfl_up()` computes the source lane ID by adding `delta` to the current thread's lane ID. Similar to `__shfl_up()`, the function does not wrap around for upper values of `delta`.

- `__shfl_xor(int var,int laneMask,int width=warpSize)`: This function calculates the source's lane ID by performing bitwise-XOR of the caller's lane ID and `laneMask`. The value held by the resulting source is returned into `var`. If `width` is less than `warpSize`, then each group of `width` threads is able to access elements from earlier groups of threads. However, if a group attempts to access later groups' elements, then the function returns their own value of the variable, `var`.

The warp vote and shuffle functions typically find their application when programmers wish to perform reduction or scan operations. Note that our discussion thus far comprised intra-block and intra-warp synchronizations. The synchronization between two blocks can only be accomplished via global memory accesses, which consumes significant amount of time. Programmers must pay attention to the type of applications they are porting to the GPGPU devices – applications that involve significant memory accesses and frequent global memory synchronization may perform better on the CPU host instead on the GPGPU device.

8.6.3 Constant memory

The constant memory resides in the device memory and is cached. This memory space is used for storing any constant values frequently accessed by the kernel function, which would otherwise consume several clock-cycles if done via the device global memory. The constant memory is also useful for passing immutable arguments to the kernel function. The current GPGPU architectures provide L1 and L2 caches for global memory, making the constant memory less lucrative.

However, constant memory can provide performance boost for earlier GPGPU architectures. To declare constant memory variables inside a `.cu` file, programmers must declare global variables with `__constant__` prefix. For example,

```
__constant__ float pi=3.14159;
```

The host portion (CPU) is capable of changing a constant memory variable since a constant variable is constant only with respect to the GPGPU device. The host performs any changes to the constant memory via `cudaMemcpyToSymbol()` function:

```
template <class T> cudaError_t cudaMemcpyToSymbol (
const T & symbol, // Destination address
const void &src, //source address
size_t count, //the number of bytes to copy
size_t offset, //Offset from the start of symbol
enum cudaMemcpyKind kind); //kind is cudaMemcpyHostToDevice
```

Active Learning Exercise 11 - Consider a host variable `h_Cosine`, a one-dimensional vector of constant size, `Bins`, initialized with cosine function values at `Bins` number of angles between 0 to 2π. Declare a constant memory variable `d_Cosine` of a fixed size equal to `Bins`. Perform a host-to-device copy from `h_Cosine` to `d_Cosine`.

8.6.4 Global memory

In Section 8.2, we explored how to manage the device global memory using `cudaMalloc` and `cudaMemcpy` functions. In this section, we study these functions in more depth. The device global memory is easily the most important unit with respect to the CUDA architecture. It is the largest memory unit where all (or at least, most) of the data for GPGPU processing is stored. Because this memory unit is located in the off-chip RAM, frequent accesses to the device global memory constitutes one of the major performance limiting factors in GPGPU computing. As discussed before, the CPU host and GPGPU device memories are usually disjoint. The host portion of a CUDA program explicitly allocates the device global memory for device variables. Throughout the program, the host portion communicates with the GPGPU device by copying data to-and-from the device global memory. In what follows, we discuss CUDA functions that enable programmers to allocate the device memory variables and perform host-device communications.

C programmers are already aware of the procedure for allocating and deallocating memory regions using the `malloc()` and `free()` functions, respectively. The CUDA programming model provides simple C extensions to facilitate

device global memory management using the `cudaMalloc()` and `cudaFree()` functions. The syntaxes appear under.

```
//cudaMalloc: host portion allocates device global memory for device variables
cudaError_t cudaMalloc(
void **devPtr, //Host pointer address that will store the
//allocated device memory s address
size_t size) //size number of bytes to be allocated in device memory


//cudaFree: host portion 'frees' the device global memory
cudaError_t cudaFree( void *devPtr);
//The host pointer address storing the allocate device memory's
//address to be freed
```

The data transfer between the host portion of the code and device portion of the code is performed via the `cudaMemcpy()` function as follows:

```
//cudaMemcpy: Data transfer between the host and GPGPU device
cudaMemcpy(
void *dst_ptr, //destination address
const void *src, //source address
size_t count,//number of bytes to be transferred
cudaMemcpyKind kind) //enum type kind where kind can be
//cudaMemcpyHostToHost (0), cudaMemcpyHostToDevice (1),
//cudaMemcpyDeviceToHost (2), cudaMemcpyDeviceToDevice (3)
```

Readers are encouraged to exercise caution with de-referencing the device pointers inside the host portion, which can prove fatal for the CUDA program. Seasoned CUDA programmers avoid such mistakes by adding `h_` prefix for the host pointers and `d_` prefix for the device pointers. Additionally, readers are strongly encouraged to free the allocated device global memory pointers because the GPGPU device does not have a smart operating system for garbage collection. A complete reboot may be the only way to recover the lost device global memory.

8.7 CUDA optimizations

The CUDA programming model is not known for straight-forward GPGPU application development. A naïve and sloppy CUDA program may provide little to no performance benefits at all! To develop an efficient CUDA application, programmers must be highly intimate with the device architecture to reap its complete benefits. Fortunately, researchers have meticulously studied different applications on GPGPU architectures to provide a generic set of strategies to perform GPGPU program optimization. Although, the strategies may vary from one application to another. In general, CUDA provides three primary optimization strategies namely, Memory-level optimization, Execution Configuration-level optimization,

and Instruction-level optimization. In addition, CUDA also offers program structure optimization via unified memory. In what follows, we discuss each of these optimization strategies.

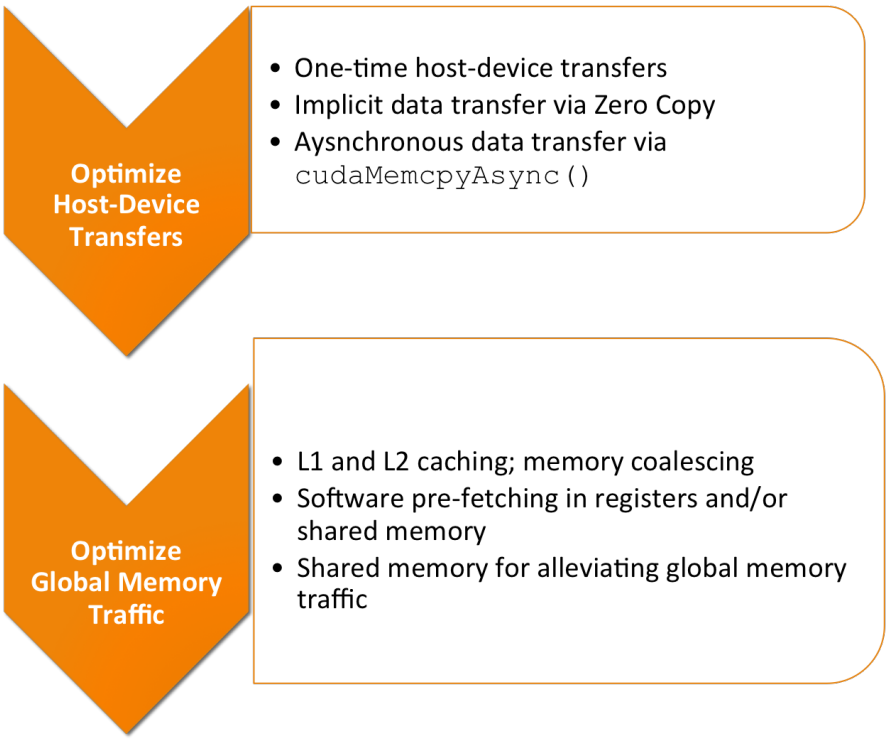


Figure 8.9: A list of commonly used memory-level optimization strategies to alleviate host-device and global memory traffic.

8.7.1 Memory-level optimization

While CUDA programming model provides several memory-level optimizations, we discuss memory optimization strategies to alleviate common performance bottlenecks arising due to host-device transfers and global memory traffic. These memory-level optimization strategies are listed in Figure 8.9.

Memory-level optimization: Host-device transfers – One memory optimization strategy is to reduce the frequent transfers between the host and the device since the host-to-device bandwidth is usually an order of magnitude lower than the device-to-device bandwidth. It is highly beneficial to transfer all of the relevant

data to the device memory for processing (even if it requires multiple kernel calls) and later transfer the data back to the host memory once all of the operations are finished.

Overlapping the kernel execution with data transfers using Zero-Copy can further optimize the host-device bandwidth. In this technique, the data transfers are performed implicitly as needed by the device kernel code. To enable Zero-Copy, the GPGPU device should support the host-mapped memory. The CUDA programming model provides `cudaHostAlloc()` and `cudaFreeHost()` functions to allocate and free the paged host memory. The mapping of the paged host memory into the address space of the device memory is performed by passing `cudaHostAllocMapped` parameter to `cudaHostAlloc()` function. The GPGPU device kernel implicitly accesses this mapped memory via the device pointer returned by the `cudaHostGetDevicePointer()` function. The functions for managing the page mapped memory are as follows.

Listing 8.6: Functions for zero-copy between host and device.

```
cudaError_t cudaHostAlloc(
void **ptr, //Host pointer to be paged
size_t size, //Size of the paged memory in bytes
unsigned int flags); //cudaHostAllocMapped for zero copy.



cudaHostGetDevicePointer(
void ** devptr, //Device pointer for GPGPU to access the paged memory
void *hostptr, //The host pointer of the paged memory
unsigned int flags); //flags is meant for any extensions, zero for now
```

Listing 8.7 illustrates Zero-Copy between the CPU host and the GPGPU device. The readers are also encouraged to read about `cudaMemcpyAsync()` [7] function for asynchronous host-device data transfers.

Listing 8.7: Illustration of Zero-Copy between the CPU host and GPGPU device. The memory copy is performed implicitly whenever the device accesses the host mapped memory via the device pointer (`d_nfire`) returned by `cudaHostGetDevicePointer()` function.

```
int main(){
:
//host vector to be page mapped
char *h_nfire;
//device pointer for the mapped memory
char *d_nfire;
cudaAllocHost((void **)&h_nfire,sizeof(char)*num_neurons,cudaHostAllocMapped));
cudaHostGetDevicePointer((void **)&d_nfire,(void *)h_nfire,0));
:
kernel<<<dimGrid,dimBlock>>>(d_nfire, num_neurons);
:
```

```
}
```

Memory-level optimization: Caching in L1 and L2 caches; and coalescing – The more recent GPGPU devices (Compute Capability 2 and higher) offer caches for global memory namely the L1 and L2 caches. For Compute Capability devices 2.x, by using the `nvcc` compiler flag `dlcm`, programmers can enable either both L1 and L2 caches by default (`-Xptxas dlcm=ca`) or L2 cache alone (`-Xptxas dlcm=cg`). A cache line is 128 bytes and is aligned with a 128-byte segment in the device memory. If both L1 and L2 caches are enabled, then the memory accesses are serviced via 128-byte transactions. If only L2 cache is enabled, then the memory accesses are serviced via 32-byte transactions. If the request size is 8 bytes, then the 128-byte transaction is broken into two requests, one for each half-warp. If the request size is 16 bytes, then the 128-byte transaction is broken into four requests, one for each quarter warp (8 threads). Each memory request is broken into cache line requests, which are serviced independently. The cache behavior for GPGPU devices is similar to general-purpose processors. If there is a cache hit, the request is served at the throughput of L1 or L2 cache. A cache miss results in a request that is serviced at the device global memory throughput.

Compute Capability 3.x, 5.x, and 6.x devices usually allow global memory caching in L2 cache alone. However some 3.5 and 3.7 devices allow programmers to opt for the L1 cache as well. The L2 caching for Compute Capability devices 3.x and above is similar to Compute Capability 2.x devices.

A progam optimally utilizes the global memory when the accesses lead to as many cache hits as possible. In such a case, threads within a warp complete memory accesses in fewer transactions. This optimized global memory access pattern is generally called *coalesced access*. The term *global memory coalescing* had significant importance to Compute Capability 1.x devices, where coalesced access rules were highly stringent. However, with the introduction of caches in recent GPGPU architectures, the term coalescing has become obscure. To achieve global memory 'coalescing' in recent GPGPU architectures, programmers must strive to write cache-friendly codes that perform aligned accesses. Similar to CPU architectures, good programming practices lead to optimal GPGPU codes.

Active Learning Exercise 12 – Perform research on Compute Capability 1.x devices; and write down the rules for coalesced global memory accesses.

Memory-level optimization: Software-prefetching using registers and shared memory – The device global memory is an order of magnitude slower than registers and shared memory. Programmers can use the register and shared memory

space for caching frequently used data from the device global memory. This technique is referred to as *software prefetching*; avid assembly language programmers among readers may already be aware of this technique.

Memory-level optimization: Shared memory to alleviate global memory traffic – The judicious use of shared memory space to reduce the global memory traffic is a highly important technique especially for algorithms that exploit data locality, matrix-matrix multiplication and several image processing applications for instance. Here, we discuss how the shared memory space can be used to enhance the global memory throughput using matrix-matrix multiplication as a case study. Readers should recall the concept of matrix-matrix multiplication: any two matrices $A_{m\times n}$ and $B_{n\times p}$ are multiplied to yield a matrix, $C_{n\times p}$. Any element c_{ij} in matrix $C_{n\times p}$ is obtained by computing the scalar product between the i^{th} row of matrix A and j^{th} column of matrix B. Let us first consider a naïve matrix-matrix multiplication and find out why it sub-optimally utilizes the global memory bandwidth. Listing 8.8 shows the naïve implementation that multiplies two matrices of equal dimensions (`width x width` each).

Listing 8.8: A naïve implementation of matrix-matrix multiplication kernel.

```
1. __global__ void
2. matrixmul_kernel(float *d_A, float *d_B, float *d_C,int width) {
3. int row, col, k=0;
4. float temp=0;
//thread accesses global row
5. row = threadIdx.x + blockIdx.x*blockDim.x;
//thread accesses global col
6. col = threadIdx.y + blockIdx.y*blockDim.y;
7. if(row <width && col < width){  //out of bound threads must not work
8.   temp=0;
9.   for (k=0;k<width;k++){
10.      temp+=d_A[row*width + k]*d_B[k*width + col];
11.  }
12. d_C[row*width+col]=temp;
13. }
14. }
```

A careful inspection of the kernel function in Listing 8.8 reveals that the performance bottleneck is in lines 9 and 10. Note that in each iteration of the `for` loop in Line 9, a thread performs two global memory loads (loads elements `d_A[row*width +k]` and `d_B[k*width + col]`, respectively) and performs two floating-point operations (multiplies the two loaded elements and adds the product with the `temp` variable). Let us define the term computation-to-global memory access (CGMA) ratio, which is the ratio of the total number of computations to the total number of global memory accesses. The CGMA ratio is often used to characterize a GPGPU kernel as a computation-bound kernel

or a communication-bound kernel. In our example of naïve matrix-matrix multiplication, the CGMA ratio is (2 floating-point operations per 2 floating-point accesses) equal to 1. This ratio is too small to reap the maximum benefits of a throughput-oriented architecture. For instance, if the GPGPU device memory has a bandwidth of 200 GB/sec, then the kernel in Listing 8.8 performs computations at the rate of 50 giga-floating point operations per second (GFLOPS). This computation throughput does not do justice to modern day GPGPU devices with peak performance as high as 10 TFLOPS for single-precision.

It is clear from the above example that the CGMA ratio for matrix-matrix multiplication needs to improve, possibly by boosting the global memory bandwidth. In what follows, we discuss 'tiled' matrix-matrix multiplication using shared memory, which enables us to improve the global memory bandwidth for this operation. Prior to delving into the GPGPU implementation, let us investigate the concept of 'tiling'. To perform matrix-matrix multiplication, the matrices can be broken into smaller tiles that are multiplied together to yield partial results. The partial results from pertinent tile-multiplication are then added to obtain the final result.

For example, consider multiplication of two matrices, $M_{4\times4}$ and $N_{4\times4}$; the result is stored in the matrix, $P_{4\times4}$ (see Figure 8.10). The matrix P can be broken into four tiles where tile-1 comprises elements $P_{0,0}$, $P_{0,1}$, $P_{1,0}$, and $P_{1,1}$; tile-2 comprises elements $P_{0,2}$, $P_{0,3}$, $P_{1,2}$, and $P_{1,3}$, and so on. Consider the evaluation of tile-1 elements; Figure 8.10 shows the tile-1 elements of matrix P enclosed in the yellow box. The tile-1 elements are evaluated in two steps: In the first step, the red tiles over matrices M and N (see Figure 8.10) are multiplied together to yield the partial result for tile-1 elements $P_{0,0}$ through $P_{1,1}$ (see partial results highlighted in red). In the second step, the tile over matrix M moves to the right (now highlighted in green in Figure 8.11) and the tile over matrix N moves down (now highlighted in green in Figure 8.11) to compute the next set of partial results (also highlighted in green in Figure 8.11). The partial results from the above two steps are added to produce the complete result for tile-1 elements. This tile movement is in agreement with the concept of matrix-matrix multiplication where we compute the scalar product between the rows of the first matrix (M in this case) and the columns of the second matrix (N in this case). The evaluation of the other tiles is similar to this tile-1 example. Readers are encouraged to compute the results for the remaining tiles for practice.

In general, how does tiling help with parallelization of matrix-matrix multiplication? To obtain an answer to this question, consider a multi-threaded computing architecture (see Figure 8.12) that stores the operand matrices in the off-chip

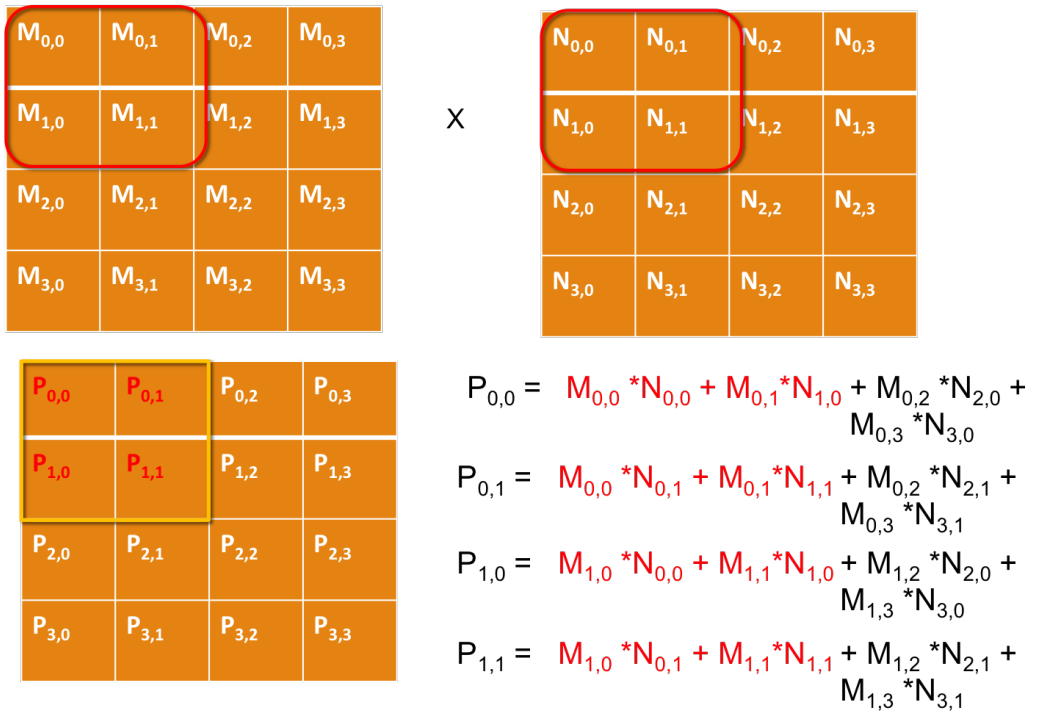


Figure 8.10: The tiles in matrices M and N (highlighted in red) multiply to yield the partial results for the tile in matrix P (highlighted in yellow box.)

memory, which resides far away from the computing architecture. Consequently, accesses to this off-chip memory is slow. Let us assume that this architecture is also equipped with on-chip shared memory that provides faster access versus the off-chip memory. The architecture contains four processing elements (*PE*s) that share the on-chip memory. For the foregoing example of multiplying matrices $M_{4\times4}$ and $N_{4\times4}$, envision the following scenario. Each one of the four *PE*s loads a red tile element from matrices *M* and *N* into the shared memory as depicted in Figure 8.12 (top). PE_1 loads $M_{0,0}$ and $N_{0,0}$; PE_2 loads $M_{0,1}$ and $N_{0,1}$; and so on. After this collaborative loading, the shared memory now contains the red tiles from *M* and *N* for the computation of the first set of partial result. Each *PE* computes its partial result via shared memory look-up: PE_1 computes the partial result for $P_{0,0}$, PE_2 computes the partial result for $P_{0,1}$ and so on. Similarly, the *PE*s cooperatively load the green tile elements (see Figure 8.12 bottom) to evaluate the second set of partial result. This collaborative loading has clearly reduced the number of trips to the farther, off-chip memory, thereby providing tremendous benefits. Do

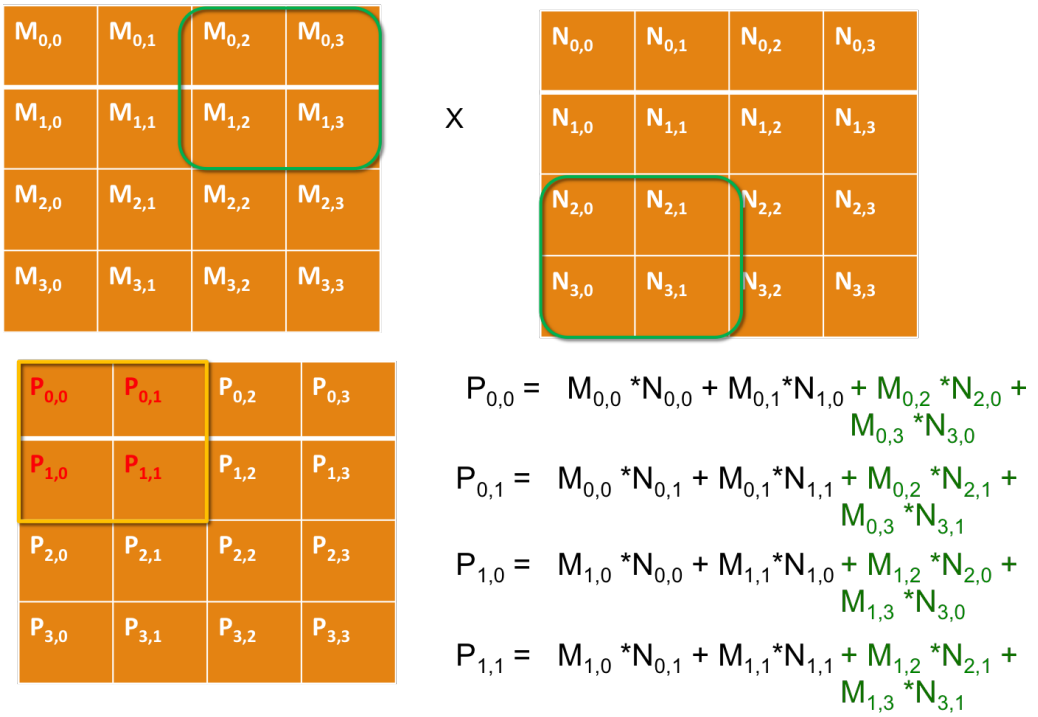


Figure 8.11: The tiles in matrices M and N (highlighted in red) multiply to yield the partial results for the tile in matrix P (highlighted in yellow box).

we have an architecture that facilitates this tiling operation? GPGPU devices are great fit!

Listing 8.9: The shared memory implementation of matrix-matrix multiplication, also called as tiled matrix-matrix multiplication.

```
0. #define TILEWIDTH 16
1. __global__ void
2. matrixmul_kernel(float *d_A, float *d_B, float *d_C,int width) {
3. __shared__ float Ashared[TILEWIDTH][TILEWIDTH];
//shared memory to load shared tile from matrix A
4. __shared__ float Bshared[TILEWIDTH][TILEWIDTH];
//shared memory to load shared tile from matrix B
5. int bx=blockIdx.x, by=blockIdx.y;
6. int tx=threadIdx.x, ty=threadIdx.y;
7. int row=bx*TILEWIDTH+tx;
8. int col=by*TILEWIDTH+ty;
9. float temp=0;
10.//Loop over the tiles Ashared and Bshared to compute an element in d_C
11. for (int i=0;i < width/TILEWIDTH; i++){
//threads collaboratively load Ashared
12.     Ashared[tx][ty] = d_A[row*width + i*TILEWIDTH + ty];
```

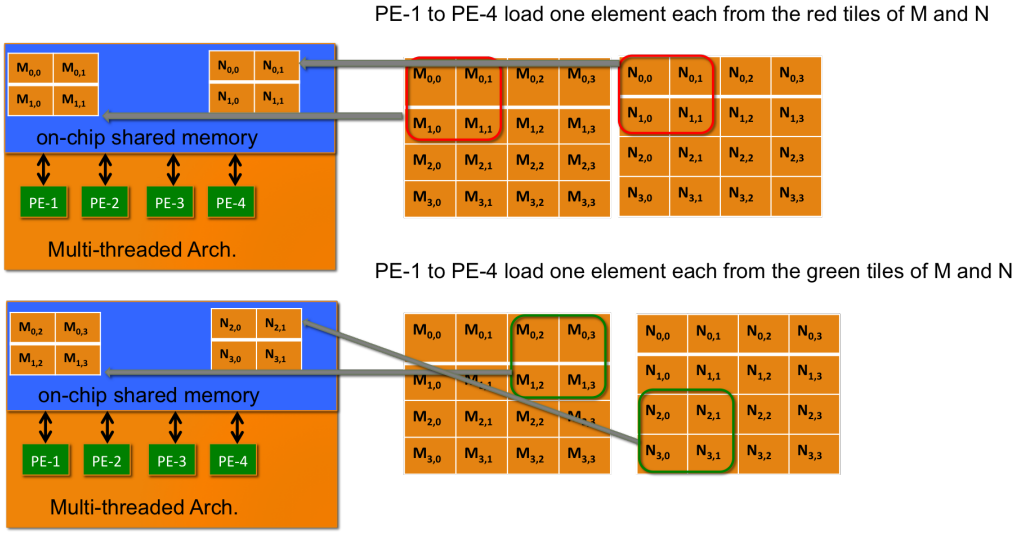


Figure 8.12: A general depiction of matrix-matrix multiplication on a multi-threaded architecture with shared memory.

```
//threads collaboratively load Bshared
13.     Bshared[tx][ty] = d_B[(i*TILEWIDTH+tx)*width + col];
14.     __syncthreads(); //wait for threads int the block to finish
15.//Loop over the tiles and perform computations
16.     for(int k=0;k<TILEWIDTH;k++){
17.         temp+=Ashared[tx][k]*Bshared[k][ty];
18.     }
19.     __syncthreads(); //wait for threads in the block to finish
20.  }
21.  d_C[row*width + col] = temp;
22. }
```

Listing 8.9 provides the kernel for the shared memory implementation. In Listing 8.9, Line 0 sets the width of the tile via `#define TILEWIDTH 16`. For simplicity, we assume that the program creates thread blocks of dimensions, `TILEWIDTH*TILEWIDTH`. Lines 3 and 4 statically declare two shared variables, `Ashared` and `Bshared`. Because these variables reside in the shared memory space, all the threads in a thread block will have access to these variables. Lines 5 and 6 store the thread block IDs (in x and y dimensions) and thread IDs (in x and y dimensions) in variables `bx`, `by`, `tx`, and `ty`, respectively. In Lines 7 and 8, each thread calculates the global row (`row`) and global column (`col`) indices of the target element in `d_C`.

Figure 8.13 shows the conceptual representation of calculating the matrix indices for tiled matrix multiplication. A `for` loop over counter, `i` in Line 11 per-

forms tile traversal over the matrices. Because the matrix `d_A` is traversed horizontally, the tile traversal requires an offset of `i*TILEWIDTH` in the horizontal direction for each iteration of the counter, `i`. A single thread with local ID (`tx`, `ty`) then accesses the element (`row`, `i*TILEWIDTH + ty`) in matrix `d_A` and loads it in the shared array `Ashared[tx][ty]` (Line 12). Similarly, the matrix `d_B` is traversed in vertical direction; therefore the tile traversal requires an offset of `i*TILEWIDTH` in vertical direction for each iteration of counter, `i`. Correspondingly, a thread with local ID (`tx`, `ty`) accesses the element (`i*TILEWIDTH + tx`, `col`) in matrix `d_B` and loads it in shared array, `Bshared[tx][ty]` (Line 13). Note that the threads in a thread block must wait for all the other participant threads to load their respective elements. This synchronization is provided by `__syncthreads()` in Line 14. After loading the shared arrays with relevant matrix elements, each thread evaluates its partial result in Lines 16 through 18. Line 19 provides the synchronization such that all the threads in a thread block finish their respective computations. At the end of the `for` loop (Line 20), each thread loads the complete result of its respective element (`row`,`col`) in matrix, `d_C` (Line 21).

Readers should carefully observe that each thread performs exactly two global loads, one for each matrix in lines 12 and 13. After these global loads, each thread performs `TILEWIDTH` multiplications and `TILEWIDTH` additions (i.e., `TILEWIDTHx2` floating-point operations) in lines 16-18. Therefore, this kernel performs `TILEWIDTH` floating-point computations for every floating-point global memory access, thereby providing `TILEWIDTH` times boost to the CGMA ratio (recall that CGMA ratio for the naïve implementation is 1). On a GPGPU device with 200 GB/s global memory bandwidth, the kernel provides a performance of $\frac{200 \quad GB/sec}{4B \quad per \quad floating-point} \times (TILEWIDTH = 16) = 800$ GFLOPS!

Active Learning Exercise 13 – A kernel performs 100 floating-point operations for every 10 floating-point global memory accesses. What is the CGMA ratio for this kernel? Assuming that the GPGPU device has a global memory bandwidth equal to 150 GB/sec, what is the kernel performance in GFLOPS?

With our previous discussion on the use of shared memory to alleviate the global memory congestion, it is clear that judicious use of the shared memory can provide substantial performance boost for CUDA programs. However, programmers should be aware of a potential shared memory performance bottleneck called the *bank conflict*. In GPGPU devices, the shared memory is divided into 32 banks such that successive 32-bit words are stored in successive banks. A bank conflict arises when multiple threads within a warp access the same bank. Whenever a

bank conflict arises, the accesses to the shared memory bank are serialized. An n-way bank conflict arises when `n` threads in a warp access the same bank – such accesses are completed in `n` serial steps. If two threads access the addresses within the same 32-bit word, then the word is broadcast to the threads, thereby avoiding a bank conflict. Similarly, a bank conflict is avoided when all the threads in a warp or a half-warp access the same word. In such a case, the word is broadcast to the entire warp or the half-warp. Bank conflicts usually arise when threads access the shared memory with some stride, `s`. For example:

```
extern __shared__ float Ashared[];
data=Ashared[Base+s*thread_id];
```

Figure 8.14 shows examples of two stride accesses, `s`=1 and `s`=2. As shown in the same figure, the shared memory is divided into 32 banks with successive words stored in successive banks. The bank-0 stores words 0 and 32, bank-1 stores, 1 and 33, and so on. For stride `s`=1, each thread (0 through 31) in a warp accesses a unique bank (0 through 31), therefore there is no bank conflict in this case. However, for stride `s`=2, the threads in the first half-warp (0-15) have a two-way bank conflict with the threads in the second half-warp (16-31). For example, the thread with ID equal to 0 (belonging to the first half-warp) accesses a word at offset 0 from the base address (stored in bank-0) and the thread with ID equal to 16 (belonging to the second half-warp) accesses a word at offset 32 from the base address (also stored in bank-0), leading to a two-way bank conflict.

Active Learning Exercise 14 – This activity summarizes our understanding of the CGMA ratio and shared memory bank conflict. Assume that the kernel given in Listing 8.10 is executed on a GPGPU device with global memory bandwidth equal to 200 GB/sec. Calculate the CGMA ratio and the performance achieved by this kernel in GFLOPS. Notice the use of two shared memory variables, `fire` and `fired`. Is there a potential for bank conflict(s)? Why or why not?

Listing 8.10: Kernel code for Active Learning Exercise 14.

```
1. __global__ void kernel(float *level1_I,float *level1_v,
float *level1_u,mytype *L1_firings, mytype2 *myfire,int Ne){
2.       extern __shared__ bool fire[];
3.      __shared__ bool fired;
4.       int k = threadIdx.x + blockIdx.x*blockDim.x;
5.      int j= threadIdx.x;
6.       auto float level1v, level1u;
7.      if(j==0)
8.                   fired =1;
9.       __syncthreads();
10.       if(k<Ne){
11.              level1v = level1_v[k];
12.              level1u = level1_u[k];
13.              if (level1v>30) {
```

```
14.                             L1_firings[k]=0;
15.                             level1v=-55;
16.                             level1u=level1u+4;
17.                 }
18.                 level1v=level1v+0.5*(level1v*(0.04*level1v+5)
                            +140-level1u+level1_I[k]);
19.                 level1u=level1u+0.02*(0.2*(level1v)-level1u);
20.                 level1_v[k] = level1v; level1_u[k] = level1u;
21.                 fire[j] = L1_firings[k];
22.                 fired&=fire[j];
23.                 __syncthreads();
24.     }
25. }
```

8.7.2 Execution configuration-level optimization

This level of optimization targets the parameters appearing in the kernel execution configuration (<<< >>>) and serves two primary performance objectives: 1) maximize the multiprocessor occupancy and 2) enable concurrent execution via streams. In what follows, we discuss these two performance objectives.

Maximizing multiprocessor occupancy – As discussed in Section 8.6, on-chip, fast memories such as registers and shared memory can provide tremendous performance boost. However, the catch lies in their limited quantity, which is dependent on the device's Compute Capability. The limited number of registers and shared memory limits the number of thread blocks (and therefore, the number of warps) that can reside on a streaming multiprocessor (SM), affecting the multiprocessor occupancy. Readers should recall that the multiprocessor occupancy is the ratio of the total number of warps residing on an SM to the maximum number of warps that can reside on an SM. While a high multiprocessor occupancy does not always imply high performance, nonetheless it is a good measure of concurrency. Therefore, CUDA programmers must strive to create grids and thread blocks for kernels such that the multiprocessor occupancy is generally high. Although this process may involve some experimentation with multiple execution configurations.

How can I achieve high multiprocessor occupancy, whilst not spending time performing meticulous calculations as shown in Equations 8.6 to 8.11? NVIDIA has a wonderful and simple tool called the CUDA occupancy calculator [7] to perform all of this mathematical work! The CUDA occupancy calculator allows users to select the Compute Capability and shared memory configuration for their GPGPU devices. Once these device configurations are selected, the CUDA occupancy calculator automatically fills the device related constants such as active

threads per SM, active warps per SM, etc. The programmer then provides kernel information including the number of registers per thread (identified using the `Xptxas nvcc` switch discussed in Section 8.6), the amount of shared memory per block, and the number of threads per block information to the occupancy calculator. After receiving the above pertinent kernel information, the occupancy calculator provides the multiprocessor occupancy value (in percentage) and graphically displays the impact of varying block size, shared memory usage per block, and register count per thread on the multiprocessor occupancy.

For the CUDA kernel in Listing 8.10, let us assume that the target architecture belongs to Compute Capability 3.5 and the shared memory configuration is 16 KB (48 KB for L1 cache). The `nvcc` compilation with `Xptxas` option for this kernel yields 20 registers per thread. If we assume a thread block size equal to 192 and shared memory per block equal to 192 bytes, then CUDA occupancy calculator provides us with multiprocessor occupancy value equal to 94%. Figure 8.15 shows the impact of varying block size, shared memory usage, and register count on occupancy, as given by the occupancy calculator. These figures suggest that for the thread block size equal to 256, we can expect the occupancy to reach 100%.

Readers are also encouraged to explore CUDA APIs such as `cudaOccupancyMaxActiveBlocksPerMultiprocessor` [7] for calculating the multiprocessor occupancy for CUDA kernels.

Active Learning Exercise 15 – Analyze the multiprocessor occupancy for the tiled matrix-matrix multiplication example. Assuming Compute Capability devices 3 and 5, use the CUDA occupancy calculator to obtain the multiprocessor occupancy values for thread block sizes: 128, 256, 512, and 1024.

Concurrent execution using streams – Readers should recall that frequent host-device transfers are significant bottlenecks that appear in CUDA programs. The CUDA streams provide a way to hide the data transfer latency by overlapping the memory transfers with kernel invocations. A *stream* consists of a sequence of instructions that execute in-order; these sequences include host-device transfers, memory allocations, and kernel invocations. For devices with Compute Capability 2.0 and above, streams enable programmers to perform device-level concurrency – while all of the instruction sequences within a stream execute in-order, multiple streams may have instruction sequences executing out-of-order. Therefore, instruction sequences from different streams can be issued concurrently. For instance, when a single stream performs kernel invocation, the other stream completes any data transfer operation. It should be noted that relative execution order of instruction sequences across streams is unknown.

CUDA streams are of type `cudaStream_t` type and generally follow the

coding sequence given under:

- **Stream creation**: : `cudaStreamCreate()` function call creates a stream:

  ```
  cudaError_t cudaStreamCreate(cudaStream_t *stream);
  ```

- **Stream use in asynchronous data transfer**: A stream can also perform asynchronous data transfers using `cudaMemcpyAsync()` function as follows:

  ```
  cudaError_t cudaMemcpyAsync(void *dst, const void *src, size_t
  count, enum cudaMemcpyKind kind, cudaStream_t stream);
  ```

  It should be noted that the host memory should be pinned for the above usage.

- **Stream use in execution configuration**: A kernel invocation is assigned to a stream by specifying the stream in execution configuration as under:

  ```
  kernel <<<dimGrid,dimBlock,SharedMemory,stream>>>(<kernel-args>);
  ```

- **Stream Destruction**: After use, the stream is destroyed using the `cudaStreamDestroy()` function. This function is blocking and only returns when all of the instruction sequences within a stream are completed.

Listing 8.11 provides a self-explaining code snippet elucidating the above described sequence.

Listing 8.11: Illustration of two concurrent streams following the sequences: stream creation, asynchronous data transfer, kernel invocation, and stream destruction.

```
//Creating two streams
1. int size=1024; //1024 data items per stream
2. :
3. cudaStream_t stream[2];
4. //Allocate host and device memory
5. float *h_data[2],*d_data[2];
//one host-device pair for each stream
6. for(i=0;i<2;i++) {
7.    cudaMallocHost((void**)&h_data[i],sizeof(float)*size);
8.    cudaMalloc((void**)&d_data[i],sizeof(float)*size);
9. }
10. //Perform initialization
11. :
12. //Follow the stream sequences except for destruction
13. for(i=0;i<2;i++) {
14.    cudaStreamCreate(&stream[i]); //create stream i
// ith stream initializes async. host-to-device transfer
15.    cudaMemcpyAsync(d_data[i],h_data[i],sizeof(float)*size,
```

```
                                    cudaMemcpyHostToDevice, stream[i]);
// ith stream invokes the kernel
16.    kernel<<<dimGrid, dimBlock, shared, stream[i]>>>(d_data[i], size);
17.cudaMemcpyAsync(h_data[i], d_data[i], sizeof(float)*size,
                                    cudaMemcpyDeviceToHost, stream[i]);
//ith stream initializes async. device-to-host transfer
18. }
19. //Streams synchronize. Blocks until streams finish
20. cudaStreamDestroy(stream[0]);
21. cudaStreamDestroy(stream[1]);
22. //free pointers
23. }
```

Active Learning Exercise 16 – Write a CUDA program that creates `n` streams to perform vector-vector addition. Hint: The i^{th} stream operates on the data starting from `&d_A[i*data_per_stream]` and `&d_B[i*data_per_stream]`.

8.7.3 Instruction-level optimization

This level of optimization targets the optimization of arithmetic instructions and branching statements in a CUDA kernel. The arithmetic operations can be easily optimized using `fast math` [1] functions. The branching statement optimization, however, requires meticulous handling of statements to avoid an artifact known as *divergent warps*. Readers should recall that all of the threads within a warp execute the same instruction. A warp is divergent if the threads inside a warp follow different execution paths (for example, first half-warp satisfies the `if` statement while the second half-warp satisfies the `else` statement). In such a case, divergent paths are serialized, which results in reduced performance. To illustrate this concept, we discuss an important parallel pattern called reduction, which derives a single value by applying an operation (addition, for instance) to all of the elements in an array. Listing 8.12 provides the code snippet of a reduction kernel (Variant 1), which is prone to producing divergent warps. Readers are encouraged to verify that the code will produce the correct reduction result.

Listing 8.12: Reduction kernel snippet (Variant 1) that produces divergent warps.

```
1. __shared__ float partialSum[BLOCKSIZE];
2. :
3. int t = threadIdx.x;
4. for(int stride = 1; stride < blockDim.x; stride*=2){
5.       __syncthreads();
6.       if(t%(2*stride)==0)
7.               partialSum[t]+=partialSum[t+stride];
8.}
```

To analyze this example, let us assume that our hypothetical GPGPU device supports 8 threads per warp. Further assume that reduction is performed using

blocks of size 32 threads. Figure 8.16 illustrates the participating threads (highlighted in black) within a warp in each iteration of the `for` loop (stride varies from 1 to 16). As seen in the same figure, there is at least one divergent warp in each iteration of the `for` loop. Specifically, strides 1, 2, and 4 include four divergent warps each; whereas strides 8 and 16 include two and one divergent warps, respectively. The entire execution of the `for` loop leads to $4+4+4+2+1=15$ divergent warps. As discussed before, divergent warps are serialized, thereby reducing the performance.

Listing 8.13 provides the code snippet of a reduction kernel (Variant 2 that reduces the number of divergent warps). Figure 8.17 illustrates the participating threads within a warp in each iteration of the `for` loop (`stride` varies from 16 to 1).

Listing 8.13: Reduction kernel snippet (Variant 2) that reduces the number of divergent warps.

```
1. __shared__ float partialSum[BLOCKSIZE];
2. int t = threadIdx.x;
3. for(int stride = blockDim.x/2; stride >= 1; stride /=2){
4.   __syncthreads();
5.   if(t< stride)
6.         partialSum[t]+=partialSum[t+stride];
7. }
```

As seen in Figure 8.17, none of the 8-thread warps are divergent in the first two iterations of the `for` loop. The divergent warps (one each) occur only in the last three iterations, thereby leading to a total of three divergent warps (versus 15 divergent warps in Variant 1). Therefore, Variant 2 provides higher performance versus Variant 1.

Active Learning Exercise 17 – Assume that there are 256 threads per block; calculate the total number of divergent warps for Variant 1 and Variant 2 of the reduction kernel. Is the scenario any better for 512 threads per block?

8.7.4 Program Structure Optimization: Unified Memory

In our programs so far, we performed explicit (with the exception of Zero-Copy) data transfers between the CPU host and GPGPU device via `cudaMemcpy` function. Needless to say, this process may be very lengthy and highly error-prone for large programs. *Unified memory* is a nice feature introduced in CUDA 6.0 that enables programmers to perform implicit data transfers between the host and the device. Unified memory introduces the concept of *managed memory* wherein the memory is allocated on both the host and the device under the supervision of the

device driver. The device driver ensures that these two sets of data remain coherent throughout the program execution. In essence, the user just maintains a single pointer for both the CPU host and GPGPU device. A data transfer is implicitly triggered before the kernel launch and another one immediately after the kernel termination. Readers should note that the unified memory operation is similar to explicit host-device transfers, with the exception that the device driver automatically manages data transfers in unified memory. Unified memory alleviates programmers with the burden of meticulous host-device transfer management, allowing them to write shorter codes and focus more on the program logic. Unified memory should not be confused with Zero-Copy where the data transfer is triggered whenever the device kernel accesses the data. Figure 8.18 summarizes the difference in 'developer's view' between an explicit data transfer (shown on the left) and unified memory data transfer (shown on the right).

Programmers can allocate managed memory in two ways:

1. Dynamically via the `cudaMallocManaged()` function call.
2. Statically by declaring global variable with the prefix: `__managed__`.

The syntax for `cudaMallocManaged()` is as follows.

```
template <class T> cudaMallocManaged(
T **dev_ptr, //address of the memory pointer
 size_t bytes, //size in bytes of the required memory
 unsigned flags) //Either cudaMemAttachGlobal for all kernels to access or
// cudaMemAttachHost to make the variable local to declaring host
// and kernels invoked by the declaring host.
```

Listing 8.14 illustrates unified memory using vector-vector addition as example. While the kernel construction is the same as Listing 8.1, notice the changes in the `main()` function. Using `cudaMallocManaged()`, lines 4-6 allocate the space for variables `a`, `b`, and `c` on both the CPU host and GPGPU device. The host input is initialized in lines 8-10 and the device output is evaluated by the kernel call in Line 12. Prior to accessing the modified values of the variables, programmers must ensure that the kernel has terminated. This check is done via `cudaDeviceSynchronize()` function call in Line 13. The variables `a`, `b`, and `c` are freed via `cudaFree()` function call in lines 15-17.

Listing 8.14: Vector-vector addition code snippet illustrating unified memory.

```
1. int main(int argc, char **argv) {
2.   int *a,*b,*c;
3.   int vec_size=1000, i;
4.   cudaMallocManaged(&a, vec_size*sizeof(int));
```

```
5.   cudaMallocManaged(&b, vec_size*sizeof(int));
6.   cudaMallocManaged(&c, vec_size*sizeof(int));
7.  //Host-portion prepares the data
8.  for (i=0; i<vec_size; i++) {
9.    a[i]=i; b[i]=i;
10. }
11. //Run the GPU Kernel
12. gpu_kernel<<<1,vec_size>>>(a,b,c,vec_size);
13. cudaDeviceSynchronize(); //Wait for the GPU to finish execution.
14. //Free pointers
15. cudaFree(a);
16. cudaFree(b);
17. cudaFree(c);
18. return 0;
19. }
```

The example in Listing 8.14 shows substantial simplification of the vector-vector addition code structure using the unified memory concept. Although, programmers must note that unified memory is not a performance optimization. Proficient CUDA programmers with a command on explicit host-device transfers and Zero-Copy optimization technique can achieve high-performance for their applications.

In this section, we discussed several optimization strategies that CUDA programmers can employ to achieve significant application performance. It is worth noting that the choice of optimization varies across applications. While the techniques covered here are quite comprehensive, we have not fully exhausted the list of possible strategies. For instance, dynamic parallelism allows a CUDA kernel to create child kernels, thereby avoiding kernel synchronization in the host portion and any host-device transfers. The HPC community continually augments the optimization strategy list via exhaustive research efforts. Readers are encouraged to stay abreast with scientific publications. Several applications share 'parallelization logic' that helps programmers avoid re-inventing the wheel.

8.8 Case study: Image convolution on GPUs

In this section, we study a parallel pattern that commonly arises in various scientific applications namely, the convolution. The convolution algorithm frequently occurs in signal processing contexts such as audio processing, video processing, and image filtering, among others. For example, images are convolved with convolution kernels (henceforth referred to as convolution masks to avoid ambiguity with the CUDA kernel) to detect sharp edges. The output of a linear time invariant (LTI) design is obtained via convolution of the input signal with the impulse

response of the LTI design. The convolution operation has two interesting aspects that make it highly lucrative for the GPGPU device. First, the convolution operation is highly data parallel – different elements of the input data can be evaluated independent of the other elements. Second, the convolution operation on a large input (a large image or an audio signal for instance) leads to significantly large number of operations. In what follows, we first provide a brief mathematical background on this highly important mathematical operation. Then, we explore how the convolution operation can be effectively deployed on GPGPU devices.

Convolution is a mathematical array operation (denoted with asterisk, *) where each output element (`P[j]`) is a weighted sum of neighboring elements of the target input element (`N[j]`). The weights are defined by an input array called, the convolution mask. The weighted sum, `P[j]`, is calculated by aligning the center of the convolution mask over the target element, `N[j]`. The input mask usually consists of odd number of elements so that equal numbers of neighboring elements surround the target element in all directions.

Let us consolidate our understanding of the convolution operation via an example. For simplicity, let us assume that we need to convolve an array of eight elements, `N`, with a convolution mask of five elements, `M`. Figure 8.19 illustrates the convolution procedure. Notice the evaluation of element, `P[2]` (top) – the center of the mask (`M[2]`) is aligned with the target input element `N[2]` (dark gray); next the overlapping elements of `P` and `M` are multiplied and the products are added to obtain the weighted sum:

$P[2] = N[0] \times M[0] + N[1] \times M[1] + N[2] \times M[2] + N[3] \times M[3] + N[4] \times M[4]$

Notice the evaluation procedure of the element, `P[1]` (Figure 8.19 bottom). Similar to evaluation of `P[2]`, the center of the mask, `M[2]` is aligned with the target input element `N[1]` (highlighted in dark gray). However, the mask element, `M[0]` flows out of array, `N`. In such a case, the overflowing elements of the mask are multiplied with 'ghost elements', g_i, which are customarily set to zero. The element, `P[1]` in this case is evaluated as:

$g1 = 0$

$P[1] = g1 \times M[0] + N[0] \times M[1] + N[1] \times M[2] + N[2] \times M[3] + N[3] \times M[4]$

This process is performed on all of the array elements to obtain the convolution output, `P`.

The convolution operation can also be extended to higher dimensions. Figure 8.20 shows the convolution of a two-dimensional matrix, $N_{5\times5}$ with a two-dimensional convolution mask, $M_{5\times5}$. Consider the evaluation of element, `P[1][1]`. As shown in Figure 8.20, the center of the convolution mask, `M[1][1]`, aligns with the target element, `N[1][1]`. The overlapping elements of matrices `M` and

N are then multiplied and the products are added to obtain the weighted sum as:

$P[1][1] = M[0][0] \times N[0][0] + M[0][1] \times N[0][1] + M[0][2] \times N[0][2] +$
$M[1][0] \times N[1][0] + M[1][1] \times N[1][1] + M[1][2] \times N[1][2] +$
$M[2][0] \times N[2][0] + M[2][1] \times N[2][1] + M[2][2] \times N[2][2]$

Notice the evaluation of element P[0][1] as shown in the same figure with mask element, M[1][1] aligned with the target element, N[0][1]. The mask elements M[0][0], M[0][1], and M[0][2] flow beyond the bounds of matrix, N. Therefore, the overflowing mask elements are multiplied with ghost elements, g_1, g_2, g_3, which are all set to zero. The element P[0][1] is evaluated as:

$g1 = g2 = g3 = 0$
$P[0][1] = M[0][0] \times g1 + M[0][1] \times g2 + M[0][2] \times g3 +$
$M[1][0] \times N[0][0] + M[1][1] \times N[0][1] + M[1][2] \times N[0][2] +$
$M[2][0] \times N[1][0] + M[1][1] \times N[2][1] + M[2][2] \times N[1][2]$

The above process is performed on all of the array elements to obtain the convolution output, P. As illustrated through examples in Figures 8.19 and 8.20, it is clear that: a) convolution operation is highly data parallel; b) convolution operation can be computationally intensive for large input sizes; and c) programmers must pay special attention to boundary conditions, i.e. when the convolution mask elements flow beyond the bounds of the input data.

Active Learning Exercise 18 – Perform the convolution of the two vectors, A and B given as: $A = [-1, 0, 1]$ $B = [-3, -2, -1, 0, 1, 2, 3]$.

Now that we are mathematically equipped to perform the convolution operation, let us study how it can be performed on the GPGPU devices. For simplicity, let us perform one-dimensional convolution. The arguments for a CUDA convolution kernel include the following arrays: N (input), M (mask), and output, P. In addition, the kernel requires the width of array N, let this variable be width; and width of the convolution mask, let this variable be mask_width. A naïve implementation of the one-dimensional convolution kernel appears in Listing 8.15.

Listing 8.15: A naïve implementation of one-dimensional convolution kernel.

```
1. __global__ void kernel(float *N, float *M, float *P,
                          int width, int mask_width) {
2.   int tid = threadIdx.x + blockIdx.x*blockDim.x;
3.   int start_point = tid-mask_width/2;     //place the mask center on N[tid]
4.   float temp=0;
5.   for( int i=0; i<mask_width; i++) {       //loop over the mask
6.     if(start_point + i >=0 && start_point + i < width) //check boundary
7.       temp+=N[start_point +i]*M[i];
8.   }
9.   P[tid]=temp;
10.}
```

As seen in Listing 8.15, each thread obtains its global thread ID, `tid` in Line 2. Because the center of the mask is placed on the target element `N[tid]`, the starting element of the mask, `M[0]` is aligned with `N[tid - mask_width/2]`. Line 3 sets the starting point to `tid - mask_width/2`. Lines 5 through 8 perform the weighted sum calculation and finally, the answer is written to the global memory location, `P[tid]`(Line 9).

What are the performance bottlenecks for this naïve kernel? A careful inspection would yield two bottlenecks: 1) There is a control flow divergence due to Line 6 – threads within a warp may or may not satisfy the `if` statement; and 2) global memory is sub-optimally utilized. In each iteration of the `for` loop in Line 5, each thread performs two floating-point operations (one multiplication and one addition) for every two accesses of the global memory (access of the input array and the mask). Consequently, the CGMA ratio is only 1, yielding a fraction of the peak performance. The control flow divergence may not be a significant issue here because only a handful of threads process the ghost elements (mask size is usually much smaller than the thread block size). The global memory accesses are a significant source of performance bottleneck and therefore must be alleviated. One immediate remedy is to store the convolution mask in the constant memory. As discussed in Section 8.6, all of the threads in a kernel globally access the constant memory. Because the constant memory is immutable, the GPGPU device aggressively caches the constant memory variables, promoting performance. As an exercise, readers are left with the task of declaring constant memory for the convolution mask and use `cudaMemcpyToSymbol()` to copy the host mask pointer, `h_M` to the device constant memory, `M`.

A careful inspection of the naïve convolution kernel in Listing 8.15 also suggests that threads within a block tend to share the access to array elements. For instance in Figure 8.19, elements required to evaluate `P[2]` are `N[0]` through `N[4]`. Similarly, elements needed to evaluate `P[3]` are `N[1]` through `N[5]`. Therefore, consecutive threads in a warp evaluating elements `P[2]` and `P[3]` require common access to elements `N[2]` through `N[4]`. The threads in a block can access the shared computational elements via shared memory. Specifically, the threads in a block load their respective elements into the shared memory, reducing the number of trips to the global memory unlike the naïve convolution. Despite of this cooperative loading, some of the threads may need access to the elements loaded by the adjacent thread blocks. Additionally, some threads within a block may require access to ghost elements. This issue is illustrated in Figure 8.21. In the same figure, consider the thread blocks of size four threads, array `N` of size equal to 15, and a convolution mask of size equal to 5. The convolution

operation requires four blocks: block-0 operates on elements 0 through 3; block-1 operates on elements 4 through 7, and so on. Consider block-0 for example – the evaluation of elements 2 and 3 clearly require elements 4 and 5, which are loaded into the shared memory by threads in block-1. We refer to these elements as halo elements (highlighted in gray). In addition to halo elements, threads 0 and 1 need access to ghost elements (highlighted in vertical bars).

With the introduction of L2 caches in modern GPGPU devices, the access to the halo elements is greatly simplified; whereas the ghost elements can be tackled using the code logic. When the threads in block-1 load elements `N[4]` through `N[7]`, it is a reasonable assumption that these values will also be stored in the L2 cache. Consequently with high probability, block-0 can find its halo elements (`N[4]` and `N[5]`) in the L2 cache, thereby optimizing global memory accesses. Similarly, block-1 can also find the halo elements, 8 and 9 when block-2 threads load their respective elements (`N[8]` through `N[11]`) into the shared memory.

To summarize, an optimized CUDA kernel can alleviate the global memory traffic using three strategies: 1) by storing the convolution mask in the constant memory, which is aggressively cached, 2) by requiring threads in a block to load their respective elements into the shared memory; these elements will also be cached in L2, and 3) access the halo elements via L2 cache. The optimized CUDA kernel for convolution operation appears in Listing 8.16.

Listing 8.16: Optimized convolution kernel that makes use of: constant memory to cache the convolution mask, L2 cache to enable threads access the elements loaded by neighboring thread blocks, and shared memory for collaborative load of elements by threads in a block.

```
1. __global__ void convolution_kernel(float *N, float *P,
                                      int width, int mask_width) {
2.   int tid = threadIdx.x + blockIdx.x*blockDim.x;
3.   __shared__ float Nshared[BLOCKSIZE];
4.   Nshared[threadIdx.x]=N[tid]; //each thread loads its respective element in
//shared memory
5.   __syncthreads(); //make sure  all threads finish loading before proceeding
6.   int myblock_start = blockIdx.x*blockDim.x;
7.   int nextblock_start=(blockIdx.x+1)*blockDim.x;
8.   int start = tid - mask_width/2; //places the center of mask on N[tid]
9.   float temp=0;
10.   for (int i=0;i<mask_width;i++){ //loop over the mask
11.     int Nelement=start + i; //element overlapping with i
12.     if(Nelement >=0 && Nelement < width) { //boundary check
13.         if(Nelement >=myblock_start &&Nelement < nextblock_start){
//Nelement present in shared memory
14.             temp+=Nshared[threadIdx.x+i-mask_width/2]*M[i]; }
15.         else {
                //not in shared memory. Access using L2 cache
16.             temp+=N[Nelement]*M[i];
```

```
17.          }
18.        }
19.     }
20.   P[tid]=temp; //write the answer to global memory
21. }
```

In Listing 8.16, note that the convolution mask, `M` resides in the device constant memory (copied into the constant memory of the device by the host in host portion); therefore, it is not passed as an argument to the kernel. In line 4, each local thread (`threadIdx.x`) within a block cooperatively loads its respective global element `N[tid]`, where `tid` is equal to `threadIdx + blockIdx.x*blockDim.x`, into the shared memory, `Nshared` (see Lines 3 to 5). After the shared memory has been loaded by all of the threads within a block, the threads in a block identify the end points of their block (see Lines 6 and 7) and their respective start positions such that the center of the mask is centered at `N[tid]` (see Line 8). The computations occur from Line 10 through 19 – for each iteration of the mask counter, `i`, the thread obtains the position of the element in `N` (labeled as `Nelement`) that overlaps with mask element, `M[i]`. If this element is within the bounds of the thread block (calculated in Lines 6 and 7), then the `Nelement` is obtained from the shared memory variable, `Nshared` (see Lines 13 and 14). However, if `Nelement` lies outside of the block boundaries, then the corresponding element in `N` is obtained via a global memory access (see Lines 15 through 17). With high probability, this global memory location is cached in L2, therefore served with L2 cache throughput. The final computation result is written back to the global memory in Line 20.

Active Learning Exercise 19 – Extend the optimized 1D convolution kernel to perform 2D convolution. Assume modern GPGPU devices that allows for general L2 caching.

In Section 8.8, we discussed an interesting parallel pattern, the convolution, which appears frequently in several scientific applications and simulations. Due to its inherent data parallelism and computation-intensiveness, the convolution operation is a great fit for GPGPU devices. Readers are also encouraged to investigate other parallel patterns including prefix sums and sparse matrix multiplication for a comprehensive understanding of GPGPU device optimizations.

We conclude our discussion on the CUDA programming model. In this chapter, we discussed the CUDA thread model and CUDA memory hierarchy, which are critical to writing effective CUDA programs. We studied different optimization strategies to attain a significant fraction of the device's peak performance. We completed our discussion on CUDA with convolution as a case study, which highlights the importance of optimizations such as constant memory, shared mem-

ory, and general L2 caching. The exploration of CUDA optimizations is figuratively endless – several applications continue to emerge that are re-organized or re-written for GPGPU computing, thereby making it a truly disruptive technology.

8.9 GPU computing: The future

In summary, this chapter covers major topics in GPGPU computing using the CUDA framework for upper-level Computer Engineering/Computer Science undergraduate (UG) students. Starting with the concept of data parallelism, we explained in detail the CUDA program structure, compilation flow, thread organization, memory organization, and common CUDA optimizations. All of these concepts were put together in Section 8.8 where we discussed convolution on GPGPUs as a case study. We organized the previous eight sections in a way that promotes active learning, encourages students to apply their knowledge and skills immediately after learning, and prepares them for more advanced topics in HPC. We hope that, after studying this chapter and finishing all active learning exercises, the students will have a good understanding of GPGPU computing and will be able to program GPGPUs using the CUDA paradigm.

Over the years, with a humble start as graphics-rendering devices, GPUs have evolved into powerful devices that support tasks that are more general, more sophisticated, and more computationally intensive. After decades of competition in the GPU world, NVIDIA and AMD are the two major players left. Their GPUs have been used to build the world's fastest and greenest supercomputers. In April 2016, NVIDIA unveiled the world's first deep-learning supercomputer in a box. Supported by a group of AI industry leaders, the company's new products and technologies are focusing on deep learning, virtual reality and self-driving cars. Equipped with the NVIDIA Tesla P100 GPU, the servers can now deliver the performance of hundreds of CPU server nodes. Taking advantage of the new Pascal architecture, the updated NVIDIA SDK provides extensive supports in deep learning, accelerated computing, self-driving cars, design visualization, autonomous machines, gaming, and virtual reality. Supporting these key areas will definitely attract more researchers and developers to this exciting field and enable them to create efficient solutions for problems that were considered unsolvable before. In the coming years, the evolution of GPUs will follow this increasing trend in terms of GPU processing power, software capabilities, as well as the diversity of GPU-accelerated applications.

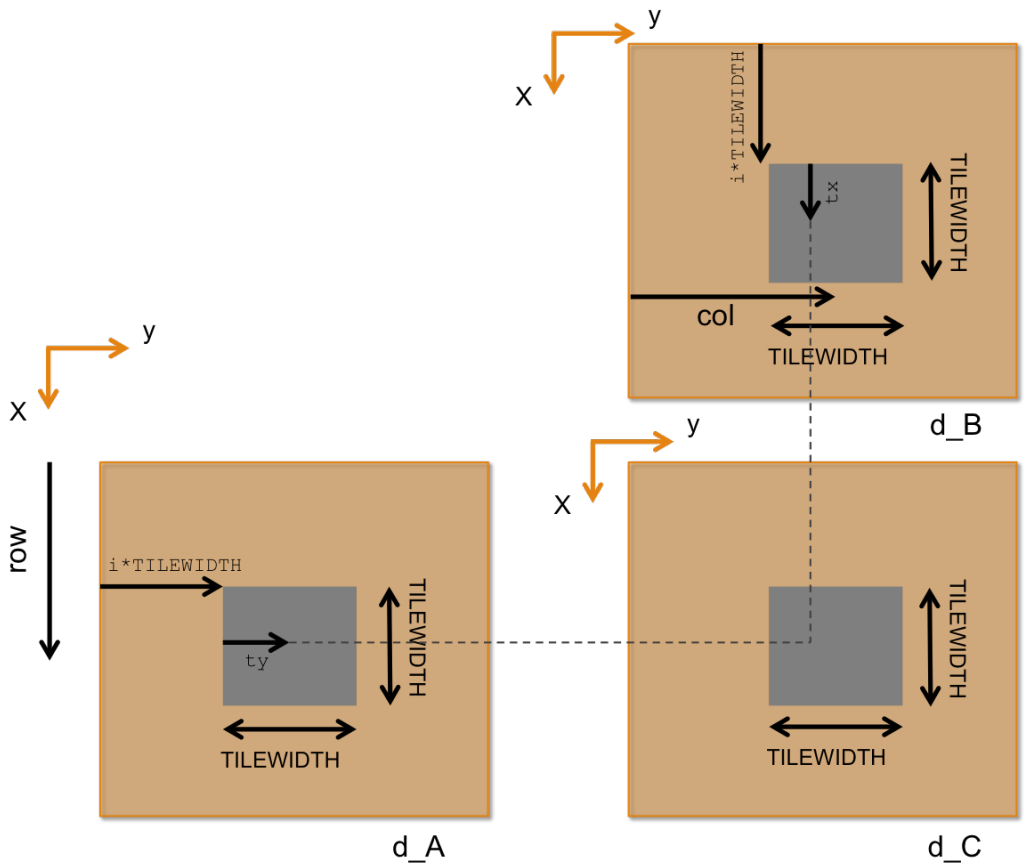


Figure 8.13: Conceptual representation of index calculation for tiled matrix multiplication.

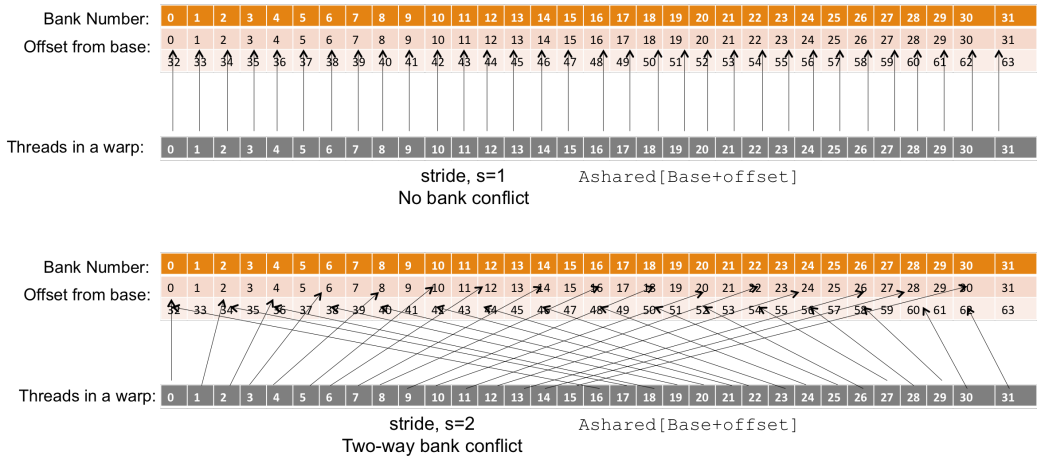
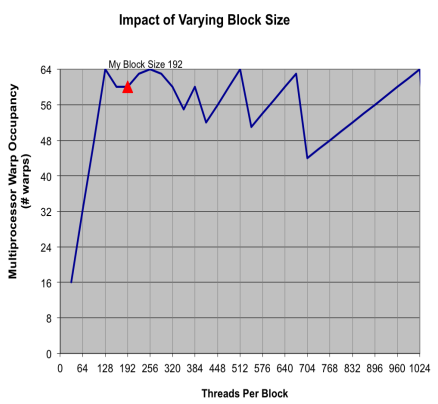
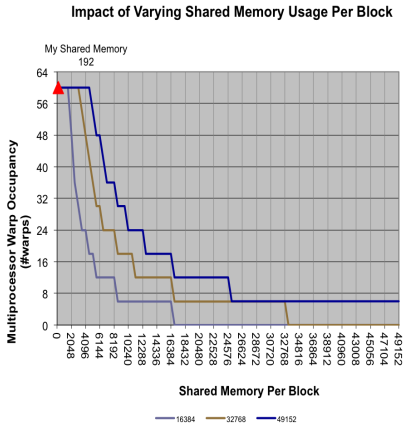


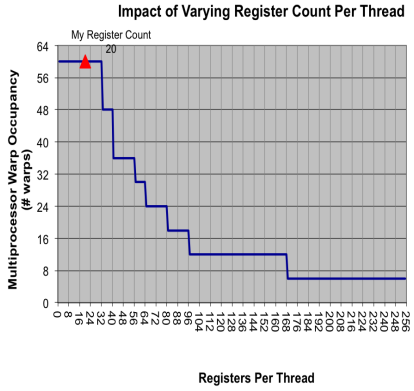
Figure 8.14: This illustration shows examples of two stride accesses, `s`=1 (top) and `s`=2 (bottom). For the stride access, `s`=1, each thread in a warp accesses a unique bank. For stride access, `s`=2, there is two-way bank conflict between the first half-warp and the second half-warp.



(a) Impact of block size on occupancy.



(b) Impact of shared memory on occupancy.



(c) Impact of register count on occupancy.

Figure 8.15: Impact of thread block size, shared memory per block usage, and register count per thread on multiprocessor occupancy.

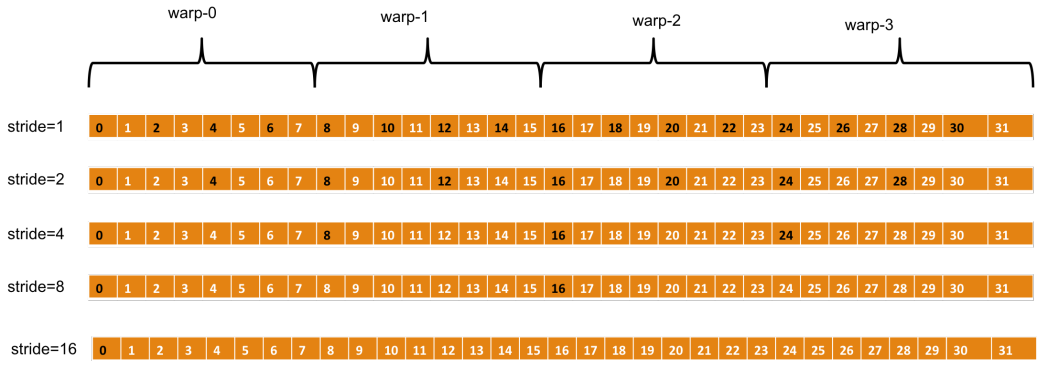


Figure 8.16: An illustration of participating threads (highlighted in black) within hypothetical warps of size equal to 8 threads. In each iteration of the `for` loop, there is at least one divergent warp.

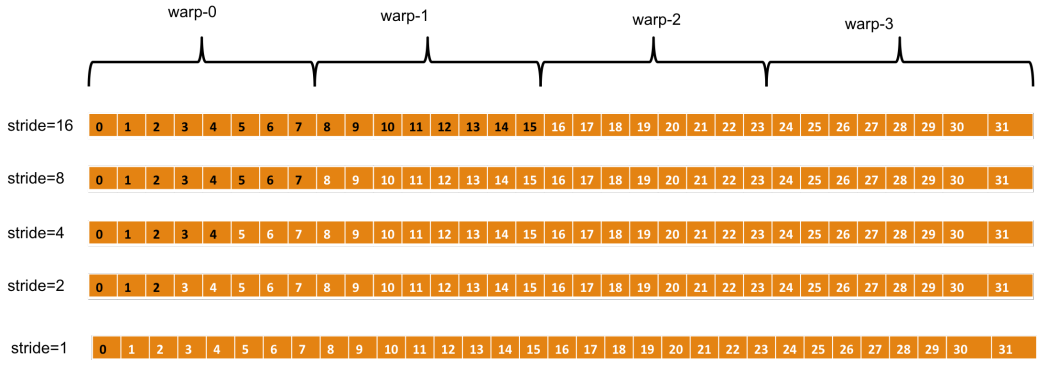


Figure 8.17: Illustration of participating threads (highlighted in black) within hypothetical warps of size equal to 8 threads. In first two iterations, none of the warps are divergent. Divergent warps (one each) occur in last three iterations.

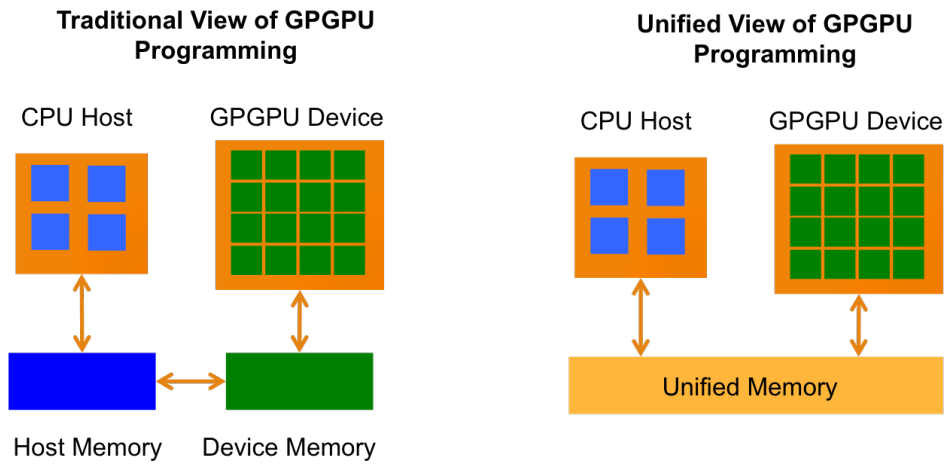


Figure 8.18: Difference in 'developer's view' between explicit data transfers and unified memory data transfers.

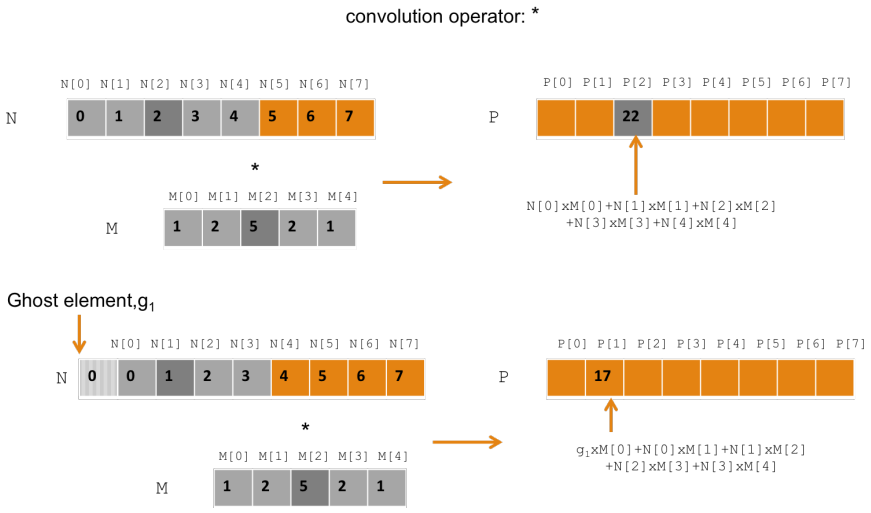


Figure 8.19: An illustration of one-dimensional convolution operation. The evaluation of element `P[2]` (see top) involves internal elements `N[0]` through `N[4]`. However, the evaluation of element `P[1]` (see bottom) requires a ghost element, g_1, which is customarily set to zero. The aligned elements are highlighted in gray and the center and target elements of `M` and `N` are highlighted in dark gray.

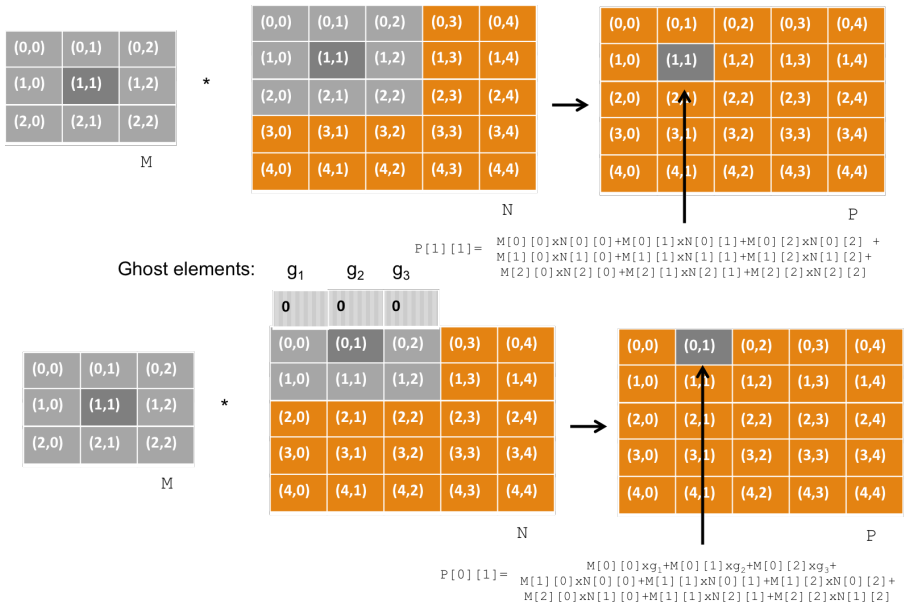


Figure 8.20: Illustration of two-dimensional convolution operation. The evaluation of element `P[1][1]` (see top) involves internal elements of `N` highlighted in gray (target element is highlighted in dark gray). However, the evaluation of element `P[0][1]` requires a ghost elements, g_1, g_2, g_3, which are customarily set to zero.

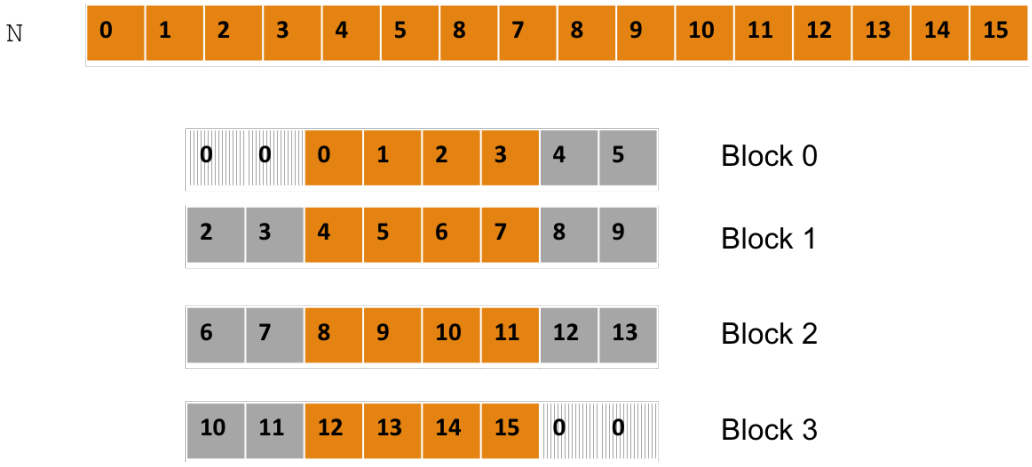


Figure 8.21: Illustration of thread blocks of size 4 requiring access to halo and ghost elements.

Bibliography

[1] CUDA zone.
https://developer.nvidia.com/cuda-zone.
LastAccessed11Feb.2018

[2] cuBLAS.
https://developer.nvidia.com/cublas.
LastAccessed11Feb.2018

[3] Nvidia cuDDN. GPU Accelerated Deep Learning.
https://developer.nvidia.com/cudnn.
LastAccessed11Feb.2018

[4] OpenCL overview.
https://www.khronos.org/opencl/.LastAccessed11Feb.
2018

[5] OpenACC. More sciene, less programming.
https://www.openacc.org/.LastAccessed11Feb.2018

[6] Thrust. https://developer.nvidia.com/thrust.
https://developer.nvidia.com/cudnn.
LastAccessed11Feb.2018

[7] Nvidia.
www.nvidia.com.LastAccessed11Feb.2018