

CUDA'S PROGRAMMING MODEL: THREADS, BLOCKS, AND GRIDS

In order to properly utilize a GPU, the program must be decomposed into a large number of threads that can run concurrently. GPU schedulers can execute these threads with minimum switching overhead and under a variety of configurations based on the available device capabilities.

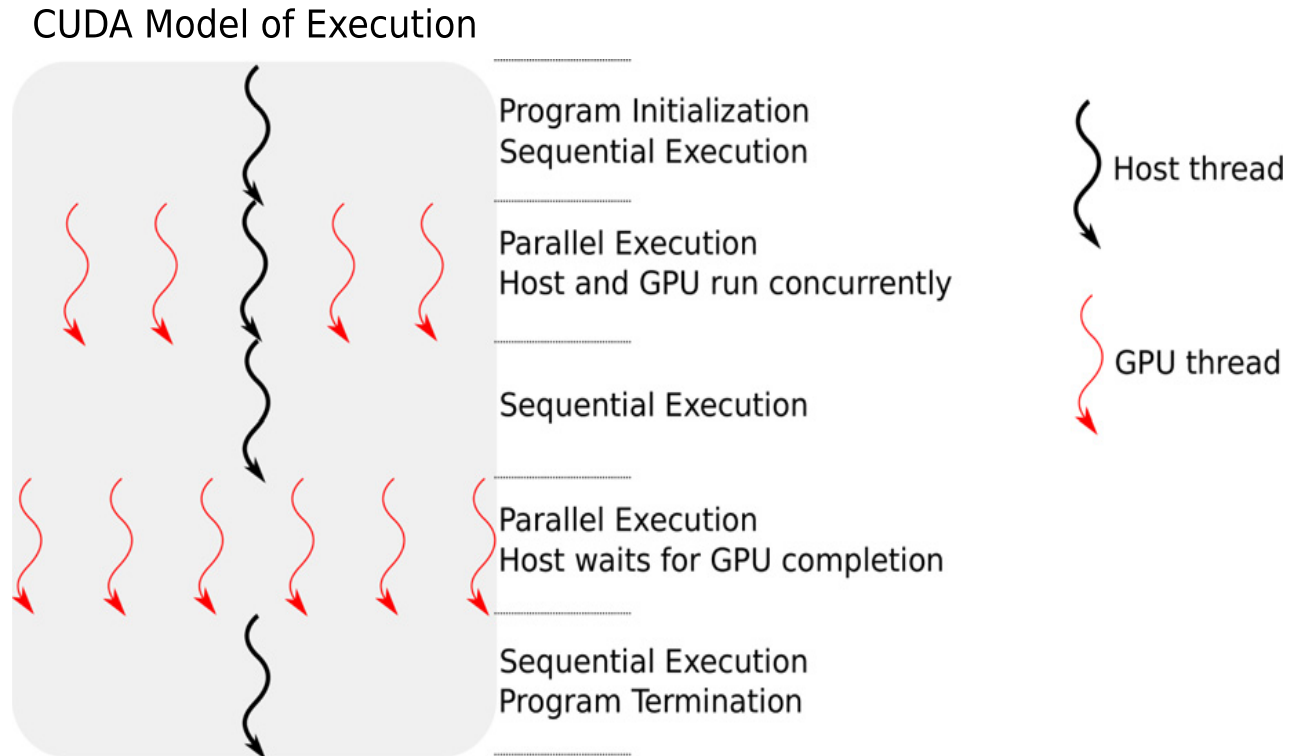


FIGURE 6.2

The GSLP CUDA execution model. The host machine may continue execution (default behavior) or may block, waiting for the completion of the GPU threads.

- Threads are organized in a hierarchy of two levels.
- At the lower level, threads are organized in blocks that can be of one, two or three dimensions.
- Blocks are then organized in grids of one, two, or three dimensions.
- The sizes of the blocks and grids are limited by the capabilities of the target device.

Table 6.1 Compute Capabilities and Associated Limits on Block and Grid sizes

Item	Compute Capability			
	1.x	2.x	3.x	5.x
Max. number of grid dimensions	2	3		
Grid maximum x-dimension	$2^{16} - 1$		$2^{31} - 1$	
Grid maximum y/z-dimension	$2^{16} - 1$			
Max. number of block dimensions	3			
Block max. x/y-dimension	512	1024		
Block max. z-dimension	64			
Max. threads per block	512	1024		
GPU example (GTX family chips)	8800	480	780	980

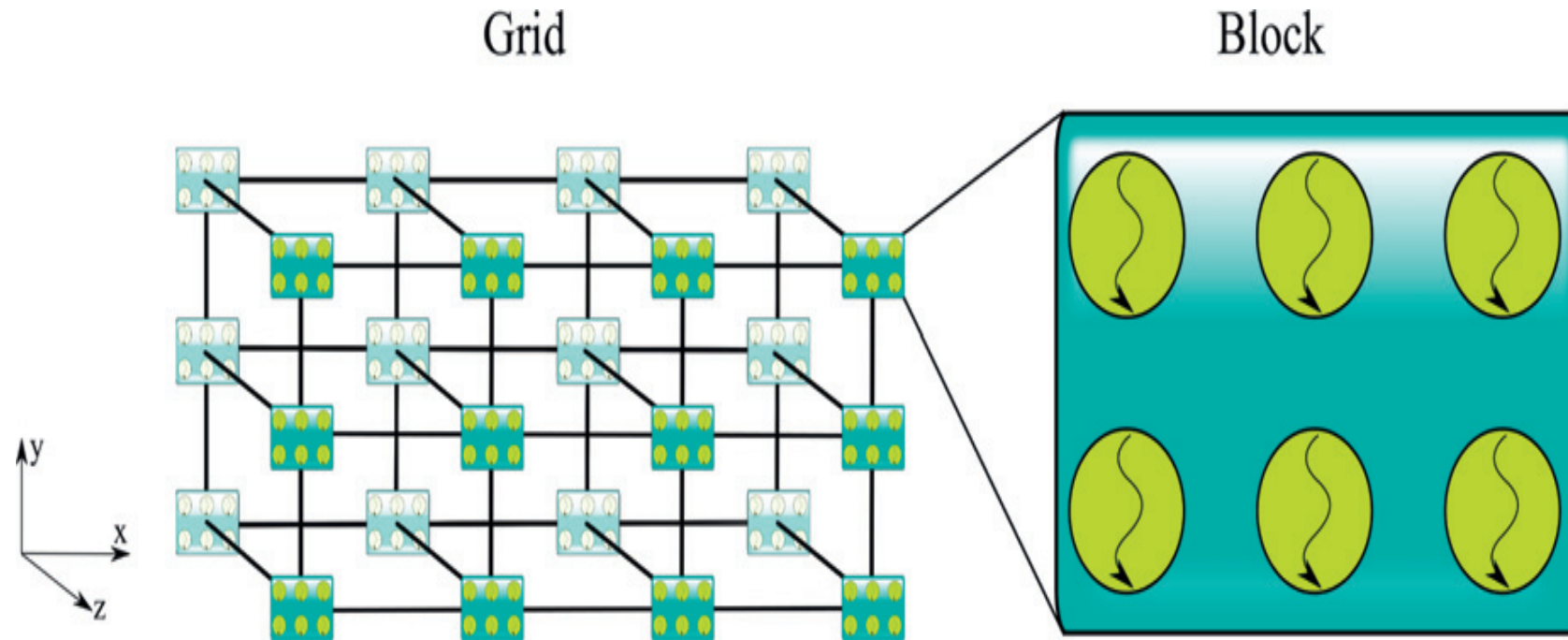


FIGURE 6.3

An example of the grid/block hierarchy used to describe a set of threads that will be spawned by CUDA. The figure illustrates a 4x3x2 grid made by 3x2 blocks. The grid connections are there only for illustrative purposes.

- The programmer has to provide a function that will be run by all the threads in a grid. This function in CUDA terminology is called a **kernel**.
- During the invocation of the kernel, one can specify the thread hierarchy with a special execution configuration syntax (<<< >>>).
- For example, running a kernel called foo() by a set of threads, like the one depicted in [Figure 6.3](#), would require the following lines:

```
dim3 block(3 ,2) ;
```

```
dim3 grid(4 ,3 ,2) ;
```

```
foo<<<grid, block>>>();
```

- The CUDA-supplied dim3 type represents an integer vector of three elements.
- If less than three components are specified, the rest are initialized by default to 1.
- In the special case where a 1D structure is desired, e.g., running a grid made up of five blocks, each consisting of 16 threads, the following shortcut is also possible:
- foo<<<5, 16>>>() ;

Many different grid-block combinations are possible. Some examples are shown

here:

```
dim3 b(3 ,3 ,3) ;
```

```
dim3 g(20 ,100) ;
```

```
foo<<<g, b>>>() ; // Run a 20x100 grid made of 3x3x3 blocks
```

```
foo<<<10, b>>>() ; // Run a 10-block grid , each block made by 3x3x3 ←  
threads
```

```
foo<<<g, 256>>>(); // Run a 20x100 grid , made of 256 threads
```

```
foo<<<g, 2048>>>();// An invalid example: maximum block size is ←  
1024 threads even for compute capability 5.x
```

```
foo<<<5, g>>>(); // Another invalid example , that specifies a ←  
block size of 20x100=2000 threads
```

Listing 6.1 shows the CUDA equivalent of a “Hello World” program.

```
1  // File: hello.cu
2  #include <stdio.h>
3  #include <cuda.h>
4
5  __global__ void hello()
6  {
7      printf("Hello world\n");
8  }
9
10 int main()
11 {
12     hello<<<1,10>>>();
13     cudaDeviceSynchronize();
14     return 1;
15 }
```

LISTING 6.1

A “Hello World” CUDA program.

- CUDA supports two more function specifiers: `__device__` and `__host__`.
- A `__device__` function can only be called from within a kernel, i.e., not from a host.
- A `__host__` function can only run on the host. The `__host__` is typically omitted unless used in combination with `__device__` to indicate that the function can run on both the host and the device.
- This means that two binary executables must be generated: one for execution on the CPU and one for execution on the GPU. The CUDA toolkit conveniently packages both inside the same executable file.

Each of the CUDA threads is aware of its position in the grid/block hierarchy via the following intrinsic/built-in structures, all having 3D components x , y , and z :

- `blockDim`: Contains the size of each block, e.g., (B_x, B_y, B_z) .
- `gridDim`: Contains the size of the grid, in blocks, e.g., (G_x, G_y, G_z) .
- `threadIdx`: The (x, y, z) position of the thread within a block, with $x \in [0, B_x - 1]$, $y \in [0, B_y - 1]$, and $z \in [0, B_z - 1]$.
- `blockIdx`: The (b_x, b_y, b_z) position of a thread's block within the grid, with $b_x \in [0, G_x - 1]$, $b_y \in [0, G_y - 1]$, and $b_z \in [0, G_z - 1]$.

- The purpose of making a thread aware of its position in the hierarchy is to allow it to identify its workload.
- **threadIdx** is not unique among threads, since there could be two or more threads in different blocks with the same index.
- Deriving a unique scalar ID for each of the threads would require the use of all of the preceding information. Each thread can be considered an element of a 6D array with the following definition⁸:

```
Thread t[gridDim.z][gridDim.y][gridDim.x][blockDim.z][blockDim.y][↵  
    blockDim.x];
```

Getting the offset of a particular thread from the beginning of the array would produce a unique scalar ID, as shown in [Listing 6.2](#).

```
int myID = ( blockIdx.z * gridDim.x * gridDim.y +  
            blockIdx.y * gridDim.x +  
            blockIdx.x ) * blockDim.x * blockDim.y * blockDim.z +  
            threadIdx.z * blockDim.x * blockDim.y +  
            threadIdx.y * blockDim.x +  
            threadIdx.x;
```

LISTING 6.2

Calculation of a unique ID for a thread in a grid of blocks.

CUDA'S EXECUTION MODEL: STREAMING MULTIPROCESSORS AND WARPS

- GPU cores are essentially vector processing units, capable of applying the same instruction on a large collection of operands. So, when a kernel is run on a GPU core, the same instruction sequence is synchronously executed by a large collection of processing units **called streaming processors, or SPs.**
- A group of SPs that execute under the control of a single control unit is **called a streaming multiprocessor, or SM.**
- A GPU can contain multiple SMs, each running each own kernel. Since each thread runs on its own SP, we will refer to SPs as cores (Nvidia documentation calls them CUDA cores), although a more purist approach would be to treat SMs as cores.
- Nvidia calls this execution model Single-Instruction, Multiple Threads (SIMT).

- SIMT is analogous to SIMD.
- The only major difference is that in SIMT the size of the “vector” on which the processing elements operate is determined by the software, i.e., the block size.
- The computational power of a GPU (and consequently, its target market) is largely determined, at least within the members of a family, by the number of SMs available.
- As an example, Table 6.2 lists a number of legacy, Fermi, Kepler, and Maxwell-class GPU offerings

Table 6.2 A sample list of GPU chips and their SM capabilities

GPU	Cores	Cores/SM	SM	Compute Capab.
GTX 980	2048	128	16	5.2
GTX Titan	2688	192	14	3.5
GTX 780	2304	192	12	3.5
GTX 770	1536	192	8	3.0
GTX 760	1152	192	6	3.0
GTX 680	1536	192	8	3.0
GTX 670	1344	192	7	3.0
GTX 580	512	32	16	2.0

- Threads are scheduled to run on an SM as a block.
- The threads in a block do not run concurrently, though.
- Instead they are executed in groups called warps.
- The size of a warp is hardware-specific.
- The current CUDA GPUs use a warp size of 32.
- At any time instance and based on the number of CUDA cores in an SM, we can have 32 threads active (one full active warp), 16 threads active (a half-warp active), or 8 threads active (a quarter-warp active).
- The benefit of interleaving the execution of warps (or their parts) is to hide the latency associated with memory access, which can be significantly high.

- An SM can switch seamlessly between warps (or half- or quarter-warps) as each thread gets its own set of registers.
- Each thread actually gets its own private execution context that is maintained on-chip.
- Each SM can have multiple warp schedulers, e.g., in Kepler there are four. This means that up to four independent instruction sequences from four warps can be issued simultaneously.
- Additionally, each warp scheduler can issue up to two instructions as long as they are independent, i.e., the outcome of one does not depend on the outcome of the other. This is known as instruction-level parallelism (ILP). As an example, let's consider the following code:
 - `a=a*b; d=b+e;`
 - These two statements can be executed concurrently, whereas the following example does not present such an opportunity due to the dependency between the two statements:
 - `a=a*b;`
 - `d=a+e; // needs the value of a`

- Once an SM completes the execution of all the threads in a block, it switches to a different block in the grid.
- In reality, each SM may have a large number of resident blocks and resident warps, i.e., executing concurrently.
- In compute capability 3.x devices, each SM has 192 CUDA cores.
- So, how can we maximize the utilization of an SM if only 32 threads are active at any time? Obviously the answer is that we cannot, unless we have multiple warps running on the SM.
- Each SM of a compute capability 3.x device has four warp schedulers, which means it can direct four different instruction sequences.
- This would still leave $192 - 4 * 32 = 64$ cores unused. These are utilized by the SM for running more warps in case there is a stall.

- In the Maxwell architecture, Nvidia has reduced the number of cores to 128 per SM, but they retain the four warp schedulers.
- Each scheduler is permanently assigned to a group of 32 cores, simplifying scheduling and improving efficiency, despite the overall reduction in cores.
- The number of resident kernels, blocks, and warps depends on the memory requirements of a kernel and the limits imposed by the compute capability of a device. These limits are shown in [Table 6.3](#).

Table 6.3 Compute Capabilities and Associated Limits on Kernel and Thread Scheduling

Item	Compute Capability					
	1.0, 1.1	1.2, 1.3	2.x	3.0	3.5	5.0
Concurrent kernels/device	1		16		32	
Max. resident blocks/SM	8			16		32
Max. resident warps/SM	24	32	48	64		
Max. resident threads/SM	768	1024	1536	2048		
32-bit registers/SM	8k	16k	32k	64k		
Max. registers/thread	128		63		255	

- As an example to the previous discussion, let's assume that we have a kernel that requires 48 registers per thread, and it is to be launched on a GTX 580 card, as a grid of 4x5x3 blocks, each 100 threads long.
- The registers demanded by each block are $100 * 48 = 4800$, which are below the 32k/SM available on this compute capability 2.0 card.
- The grid is made of $4 * 5 * 3 = 60$ blocks that need to be distributed to the 16 SMs of the card. Although Nvidia does not publish how the blocks are distributed to the SMs, we can safely assume that it should be a form of round-robin assignment.

- This means that there will be 12 SMs that will receive four blocks and six SMs that will receive three blocks. Obviously, this would be a source of inefficiency, since during the time the 12 SMs process the last of the four blocks they were assigned, the remaining four SMs are idle.
- Additionally, each of the 100-thread blocks would be split into $\lceil 100 / \text{warpSize} \rceil = 4$ warps, assuming $\text{warpSize}=32$. The first three warps would have 32 threads and the last would have four threads! So, during the execution of the last warp of each block,
- $32 - 4 / 32 = 87.5\%$ of the multiprocessors would be idle.

- These issues indicate that kernel design and deployment are critical for extracting the maximum performance from a GPU.
- A programmer can safely ignore the mechanics of the thread scheduling since no control can be exercised over when or in what sequence the threads in a kernel invocation will run. Care should be given only to the case where multiple threads modify the same memory location. Operation atomicity should be maintained, but
- this is a concern shared by multicore CPU software as well.

- There is however a significant reason why one needs to understand how threads and warps are executed, and that reason is performance.
- Threads in a warp may execute as one, but they operate on different data. So, what happens if the result of a conditional operation leads them to different paths? The answer is that all the divergent paths are evaluated (if threads branch into them) in sequence until the paths merge again. The threads that do not follow the path currently being executed are stalled. So, given the kernel in [Listing 6.4](#), the execution of a warp would be as shown in [Figure 6.4](#).

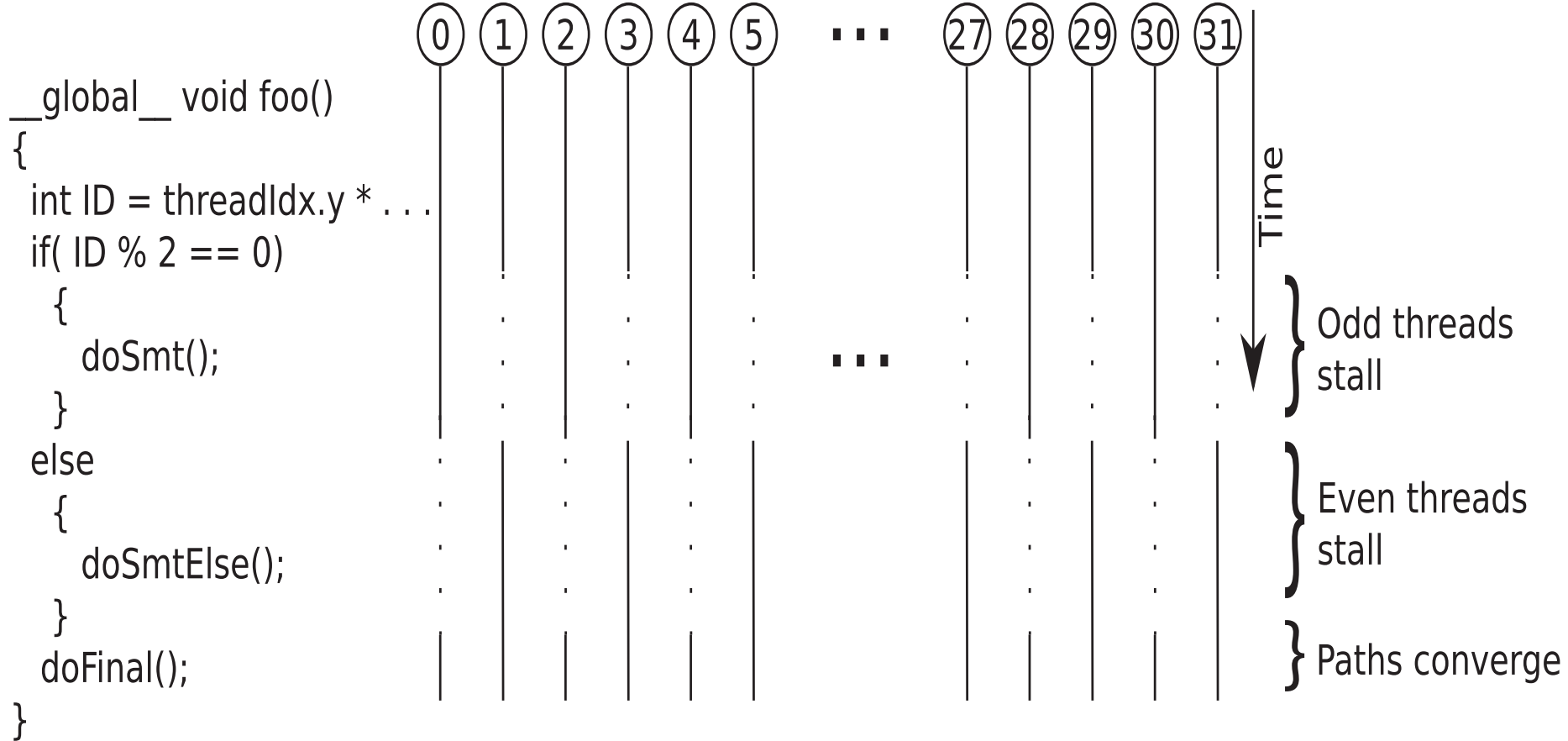


FIGURE 6.4

An illustration of the execution of the kernel in [Listing 6.4](#) by a warp of threads. The dotted lines indicate a stall.

```

__global__ void foo()
{
    int ID = threadIdx.y * blockDim.x + threadIdx.x;
    if( ID % 2 == 0)
    {
        doSmt();
    }
    else
    {
        doSmtElse();
    }
    doFinal();
}

```

LISTING 6.4

An example of a kernel that would slash the utilization of an SM in half by keeping half the threads in a warp stalled.

MEMORY HIERARCHY

- GPU memory is typically disjoint from the host's memory. So, passing chunks of data to a kernel in the form of a pointer to an array in the host's memory is not possible:
- `int *mydata= new int[N];`
- `... // populating the array`
- `foo<<<grid , block>>>(mydata , N) ; // NOT POSSIBLE !`
- Instead, data have to be explicitly copied from the host to the device and back once the required processing has been completed. This entails the allocation of two regions of memory for every array that needs to be processed on the device:
- 1. One region on the host. This can be allocated with the `new` operator or the `malloc` and `calloc` functions, as usual.
- 2. One region on the device. This can be allocated and released with the `cudaMalloc` and `cudaFree` functions.[13](#)

- In the listings that follow, in order to keep track of what kind of memory each
- pointer references, we prefix the pointers to host memory with h_ and the pointers to device memory with d_.
- The transfer of data between the two memory spaces can be done with the cudaMemcpy function:

```

// Allocate memory on the device.
cudaError_t cudaMalloc ( void** devPtr, // Host pointer address,
                                // where the address of
                                // the allocated device
                                // memory will be stored
                                size_t size ) // Size in bytes of the
                                                // requested memory block

// Frees memory on the device.
cudaError_t cudaFree ( void* devPtr ); // Parameter is the host
                                         // pointer address, returned
                                         // by cudaMalloc

// Copies data between host and device.
cudaError_t cudaMemcpy ( void* dst, // Destination block address
                        const void* src, // Source block address
                        size_t count, // Size in bytes
                        cudaMemcpyKind kind ) // Direction of copy.

```

cudaError_t is an enumerated type. If a CUDA function returns anything other than cudaSuccess (0), an error has occurred.

The `cudaMemcpyKind` parameter of `cudaMemcpy` is also an enumerated type. The `kind` parameter can take one of the following values:

- `cudaMemcpyHostToHost` = 0, Host to Host
- `cudaMemcpyHostToDevice` = 1, Host to Device
- `cudaMemcpyDeviceToHost` = 2, Device to Host
- `cudaMemcpyDeviceToDevice` = 3, Device to Device (for multi-GPU configurations)
- `cudaMemcpyDefault` = 4, used when Unified Virtual Address space capability is available (see [Section 6.7](#))

- GPUs follow a different paradigm from CPUs in the architectural design of their memory subsystem.
- In order for CPUs to operate at full speed, they need to have quick access to seemingly random data locations in main memory.
- Contemporary main memory technology (DDR3 RAM) is relatively slow, thus requiring the incorporation of big on-chip memory caches (where multiple levels of caches are a common feature).

- On the other hand, GPUs, as part of their job to filter and transform huge amounts of graphical information, need to process big collections of contiguous data that are to be read at once, without requiring that they are kept on chip for subsequent operations.
- This means that GPUs benefit from big data buses and they can do with small or no on-chip cache memories. This picture has changed slightly over more recent GPU designs, with on-chip memories getting bigger and incorporating caches in order to more efficiently support generic computation.

- The biggest discrepancy between CPU and GPU memory organizations lies in the fact that the GPU memory hierarchy is not transparent to the programmer.
- GPUs have faster on-chip memory, which occupies a separate address space than the off- chip one.
- CUDA programs can and should take advantage of this faster memory by moving frequently used data to it.
- CPU programs can be designed to exploit cache locality (e.g., by restricting the amount of data they work on at one time so that they all fit in the cache),
- but in the GPU world we have to explicitly manage data movement between the two types of memory.

- Data movement between the host and the device can only involve what is identified as global memory.
- GPUs also employ other types of memory, most of them residing on-chip and in separate address spaces. These, along with their typical performance characteristics, are illustrated in [Figure 6.8](#)

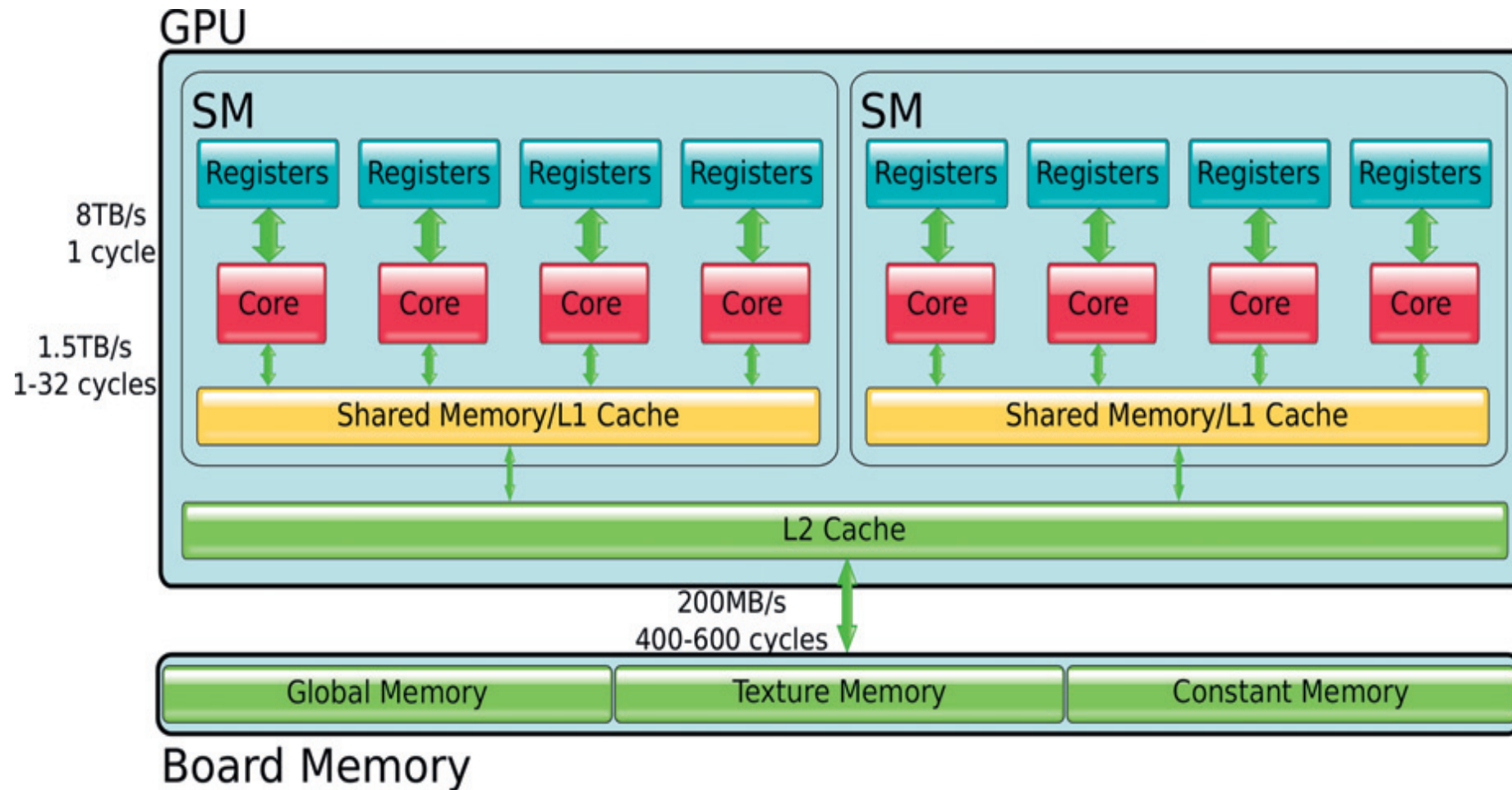


FIGURE 6.8

GPU memory hierarchy. Each bus is labeled with typical bandwidth and latency values.

- Each of the memory types has a unique set of characteristics that make it suitable for particular tasks:
- **Local memory/registers:** Used for holding automatic variables.
- **Shared memory:** Fast on-chip RAM that is used for holding frequently used data. The shared on-chip memory can be used for data exchange between the cores of the same SM.
- **Cache memory:** Cache memory is transparent to the programmer. In recent GPU generations (e.g., Fermi, Kepler), a fixed amount of fast on-chip RAM is divided between first-level cache (L1) and shared memory. The L2 cache is shared among the SMs.
- **Global memory:** Main part of the off-chip memory. High capacity, but relatively slow. The only part of the memory that is accessible to the host via the CUDA library functions.
- **Texture and surface memory:** Part of the off-chip memory. Its contents are handled by special hardware that permits the fast implementation of some filtering operations.
- **Constant memory:** Part of the off-chip memory. As its name suggests, it is read-only. However, it is cached on-chip, which means it can provide a performance boost.

Table 6.5 summarizes the different kinds of memories from the point of view of the lifetime and scope of the data residing in them.

Table 6.5 Summary of the memory hierarchy characteristics

Type	Location	Access	Scope	Lifetime
Register	On-chip	R/W	Thread	Thread
Local	Off-chip	R/W	Thread	Thread
Shared	On-chip	R/W	Block	Block
Global	Off-chip	R/W	Grid	Controlled by host
Constant	Off-chip	R	Grid	Controlled by host
Texture	Off-chip	R	Grid	Controlled by host

Table 6.1 Access Time by Memory Type

Storage Type	Registers	Shared Memory	Texture Memory	Constant Memory	Global Memory
Bandwidth	~8 TB/s	~1.5 TB/s	~200 MB/s	~200 MB/s	~200 MB/s
Latency	1 cycle	1 to 32 cycles	~400 to 600	~400 to 600	~400 to 600

LOCAL MEMORY/REGISTERS

- Each multiprocessor gets a set of registers that are split among the resident executing threads.
- These are used to hold automatic variables declared in a kernel, speeding up operations that would otherwise require access to the global or shared memories.
- A device's compute capability determines the maximum number of registers that can be used per thread.
- If this number is exceeded, local variables are allocated in the run-time stack, which resides in off-chip memory and it is thus slow to work with.
- This off-chip memory is frequently called *local memory*, but it is actually the global memory that it is used for this purpose.
- The “local” specifier just conveys the fact that whatever resides there is only accessible to a particular thread. Local memory locations can be cached by the L1 cache, so performance may not suffer much, but the outcome is application-specific.

The number of registers used per thread influences the maximum number of threads that can be resident at an SM

- Kernel has 48 registers
- 1 block=256 threads
- 1 block= $48 \cdot 256 = 12,288$ registers
- $2 \cdot 12,288 = 24,576$ registers
- If the target GPU for running the kernel is a GTX 580, sporting 32k registers per SM, then each SM could have only two resident blocks (requiring $2 \cdot 12,288 = 24,576$ registers) as three would exceed the available register space ($3 \cdot 12,288 = 36,864 > 32,768$). This in turn means that each SM could have $2 \cdot 256 = 512$ resident threads running, which is well below the maximum limit of 1536 threads per SM. This undermines the GPU's capability to hide the latency of memory operations by running other ready warps.
- GTX 580 has 32k registers per SM

- Nvidia calls *occupancy* the ratio of resident warps over the maximum possible resident warps:
- $occupancy = resident_warps / maximum_warps$
- occupancy is equal to $2 * ((256 \text{ threads}) / 32 \text{ threads/warp}) / 48 \text{ warps} = 16 / 48 = 33.3\%$.

- The GPU, unlike its CPU cousin, has thousands of registers per SM (streaming multiprocessor).
- An SM can be thought of like a multithreaded CPU core.
- On a typical CPU we have two, four, six, or eight cores.
- On a GPU we have N SM cores.
- On a Fermi GF100 series, there are 16 SMs on the top-end device.
- The GT200 series has up to 32 SMs per device.
- The G80 series has up to 16 SMs per device.
- A typical CPU will support one or two hardware threads per core.
- A GPU by contrast has between 8 and 192 SPs per core, meaning each SM can at any time be executing this number of concurrent hardware threads.
- In practice on GPUs, application threads are pipelined, context switched, and dispatched to multiple SMs, meaning the number of active threads across all SMs in a GPU device is usually in the tens of thousands range.

- One major difference we see between CPU and GPU architectures is how CPUs and GPUs map registers.
- The CPU runs lots of threads by using register renaming and the stack
- To run a new task the CPU needs to do a context switch, which involves storing the state of all registers onto the stack (the system memory) and then restoring the state from the last run of the new thread.
- This can take several hundred CPU cycles. If you load too many threads onto a CPU it will spend all of the time simply swapping out and in registers as it context switches. The effective throughput of **useful** work rapidly drops off as soon as you load too many threads onto a CPU.

- The GPU by contrast is the exact opposite.
- It uses threads to hide memory fetch and instruction execution latency, so too few threads on the GPU means the GPU will become idle, usually waiting on memory transactions.
- The GPU also does not use register renaming, but instead dedicates real registers to each and every thread.
- Thus, when a context switch is required, it has near zero overhead. All that happens on a context switch is the selector (or pointer) to the current register set is updated to point to the register set of the next warp that will execute.

- A warp is simply a grouping of threads that are scheduled together. In the current hardware, this is 32 threads.
- Thus, we swap in or swap out, and schedule, groups of 32 threads within a single SM.
- Each SM can schedule a number of blocks.
- Blocks at the SM level are simply logical groups of independent warps.
- The number of registers per kernel thread is calculated at compile time.
- All blocks are of the same size and have a known number of threads, and the register usage per block is known and fixed.
- Consequently, the GPU can allocate a fixed set of registers for each block scheduled onto the hardware.

- At a thread level, this is transparent to the programmer.
- However, a kernel that requests too many registers per thread can limit the number of blocks the GPU can schedule on an SM, and thus the total number of threads that will be run.
- Too few threads and you start underutilizing the hardware and the performance starts to rapidly drop off.
- Too many threads can mean you run short of resources and whole blocks of threads are dropped from being scheduled to the SM.
- Be careful of this effect, as it can cause sudden performance drops in the application. If previously the application was using four blocks and now it uses more registers, causing only three blocks to be available, you may well see a one-quarter drop in GPU throughput.

CACHES

- A cache is a high-speed memory bank that is physically close to the processor core.
- Caches are expensive in terms of silicon real estate, which in turn translates into bigger chips, lower yields, and more expensive processors.
- Thus, the Intel Xeon chips with the huge L3 caches found in a lot of server machines are far more expensive to manufacture than the desktop version that has less cache on the processor die.
- The maximum speed of a cache is proportional to the size of the cache.
- The L1 cache is the fastest, but is limited in size to usually around 16 K, 32 K, or 64 K. It is usually allocated to a single CPU core.
- The L2 cache is slower, but much larger, typically 256 K to 512 K.
- The L3 cache may or may not be present and is often several megabytes in size.
- The L2 and/or L3 cache may be shared between
 - processor cores or maintained as separate caches linked directly to given processor cores.
- Generally, at least the L3 cache is a shared cache between processor cores on a conventional CPU. This allows for fast inter-core communication via this shared memory within the device.

- The G80 and GT200 series GPUs have no equivalent CPU-like cache to speak of
- They do, however, have a hardware-managed cache that behaves largely like a read-only CPU cache in terms of constant and texture memory.
- The GPU relies instead primarily on a programmer-managed cache, or shared memory section.
- The Fermi GPU implementation was the first to introduce the concept of a nonprogrammer- managed data cache.
- The architecture additionally has, per SM, an L1 cache that is both programmer managed and hardware managed. It also has a shared L2 cache across all SMs.

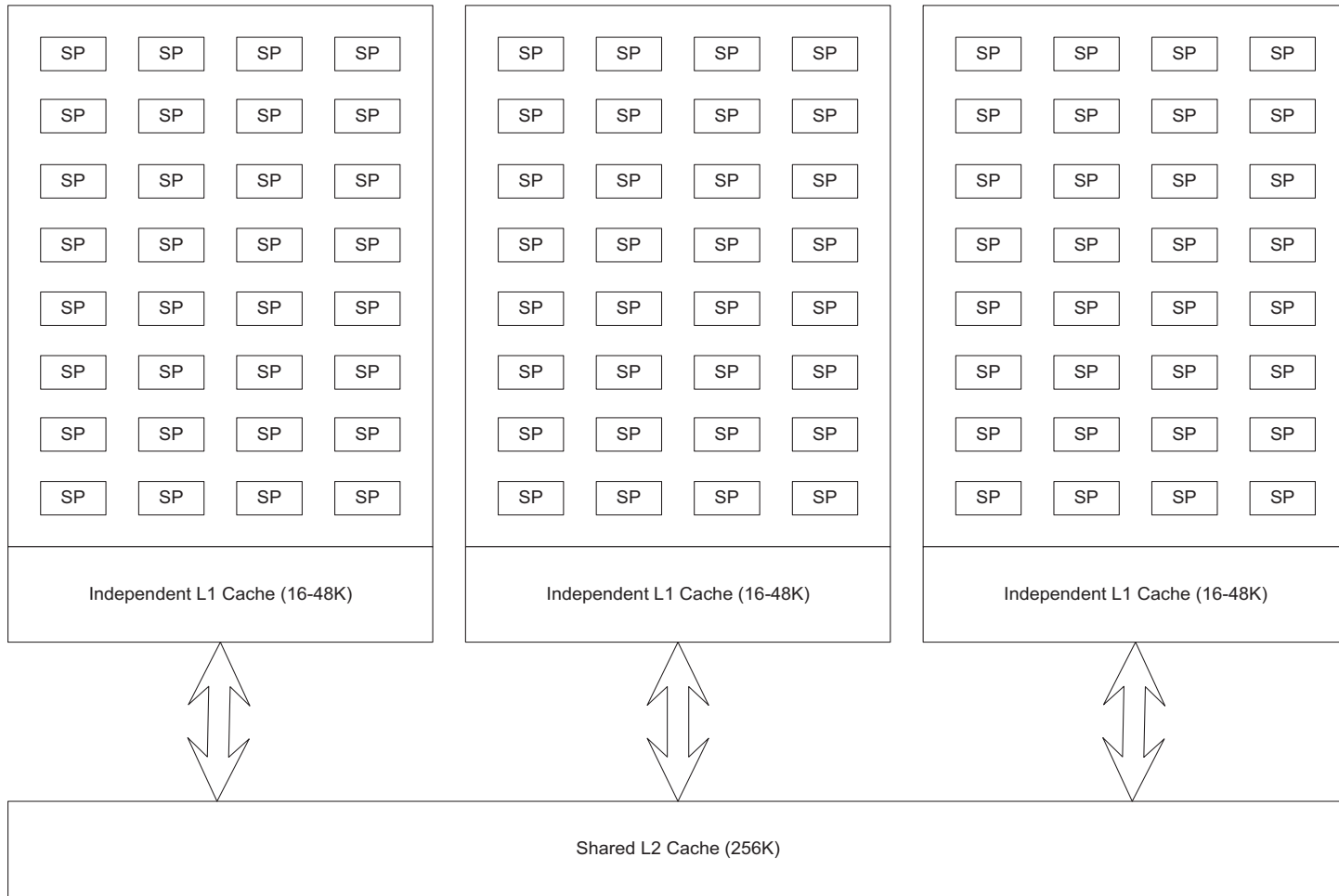


FIGURE 6.1

SM L1/L2 data path.

SHARED MEMORY

- *Shared memory* is a block of fast on-chip RAM that is shared among the cores of an SM.
- Each SM gets its own block of shared memory, which can be viewed as a user-managed L1 cache.
- In Fermi and Kepler architectures, shared memory and L1 cache are actually part of the same on-chip memory that can be programmatically partitioned in different ways.
- Currently, each SM in both architectures sports 64 KB RAM that can be partitioned as 16 KB/48 KB or 48 KB/16 KB to serve these two roles.
- Compute-Capability 3.x devices also have the option of a 32 KB/32 KB split.
- Compute-Capability 1.x devices come with only 16 KB shared memory and no L1 cache.

- A programmer can specify the *preferred* device-wide arrangement by calling the `cudaDeviceSetCacheConfig` function.
- A *preferred* kernel-specific arrangement can be also set with the `cudaFuncSetCacheConfig` function.
- Both of these functions set only a preference.

- Shared memory can be used in the following capacities:
 - As a holding place for very frequently used data that would otherwise require global memory access
 - As a fast *mirror* of data that reside in global memory, if they are to be accessed multiple times
 - As a fast way for cores within an SM, to share data
- Shared memory can be statically or dynamically allocated.
- *Static allocation* can take place if the size of the required arrays is known at compile time.
- **Dynamic allocation** is needed if shared memory requirements can be only calculated at run-time, i.e., upon kernel invocation.

Shared Memory Allocation

- Shared memory can be statically or dynamically allocated.
- Static allocation can take place if the size of the required arrays is known at compile time.
- Dynamic allocation is needed if shared memory requirements can be only calculated at run-time, i.e., upon kernel invocation.
- To facilitate this mode of operation, the execution configuration has an alternative syntax, with a third parameter holding the size in bytes of the shared memory to be reserved.

- If we were to calculate the histogram of a grayscale image consisting of N pixels, each taking up one byte in an array (`in`), the CPU code for performing this task could be in the form of the function shown in **Listing 6.6**.

```
1  // File: histogram/histogram.cu
2  . . .
3  void CPU_histogram (unsigned char *in, int N, int *h, int bins)
4  {
5      int i;
6      // initialize histogram counts
7      for (i = 0; i < bins; i++)
8          h[i] = 0;
9
10     // accumulate counts
11     for (i = 0; i < N; i++)
12         h[in[i]]++;
13 }
```

LISTING 6.6

CPU histogram calculation.

- The bins-sized array `h` holds the result of the calculation. Obviously, the one byte per pixel restricts the number of categories: $\text{bins} \leq 256$.
- A multithreaded solution to this problem would require the partitioning of the image data into disjoint sets, the calculation of partial histograms by each thread, and finally, the consolidation of the partial histograms into the complete one.
- The CUDA solution described here, follows the same guidelines, more or less, but with some key differences that go beyond the explicit data movement between the host and device memories:

- The data partitioning is implicit. All the spawned CUDA threads have access to all the image data but go through them using a different starting point and an appropriate stride that makes it possible to cover all the data while coalescing memory accesses (more on this topic in [Section 6.7](#)). This difference is highlighted in [Figure 6.9](#).
- To speed up the update of the local counts, a “local” array is set up in shared memory. Because multiple threads may access its locations at any time, atomic addition operations have to be used for modifying its contents.

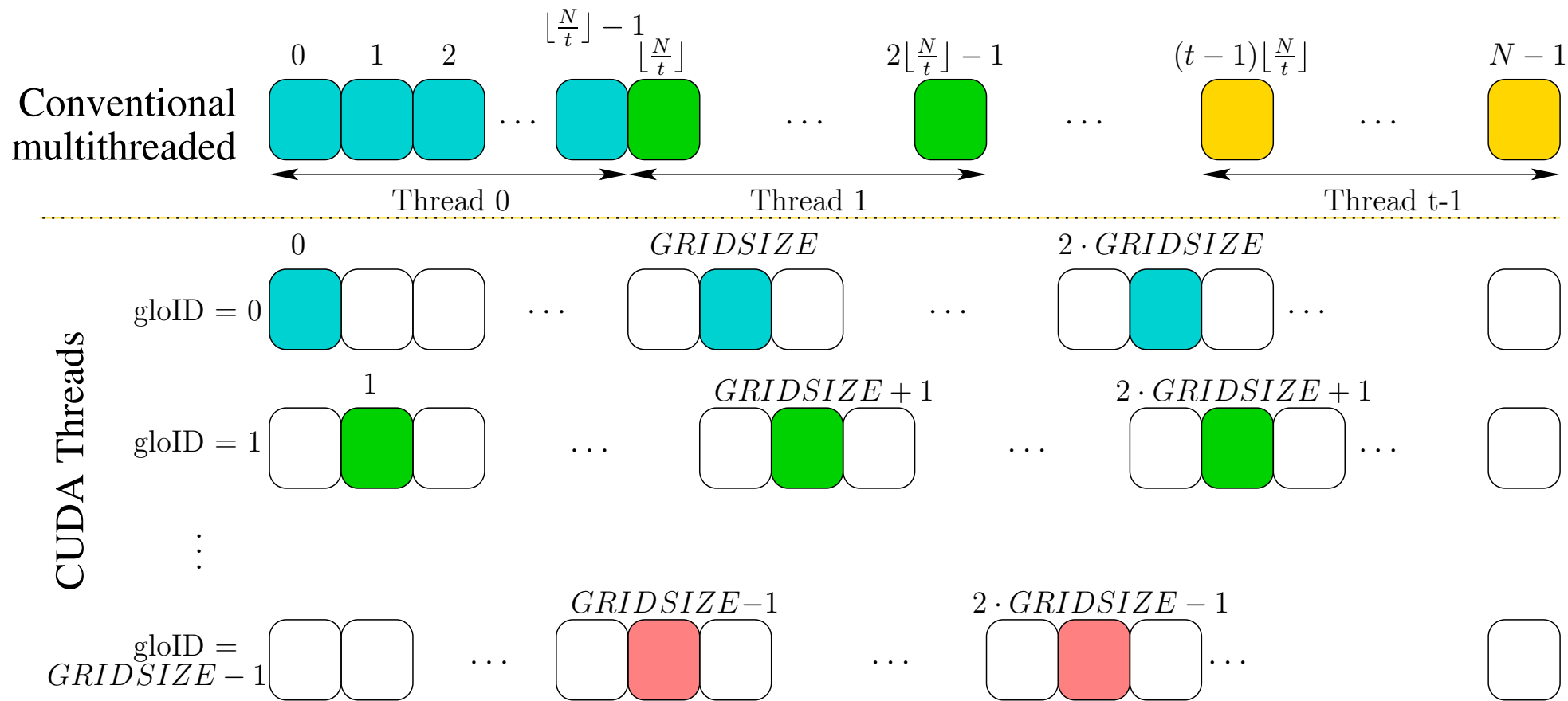


FIGURE 6.9

An illustrated comparison between the input data access patterns of a multithreaded CPU solution with t threads for histogram calculation and a CUDA solution. The symbols used here reference the variables of [Listing 6.7](#).

- Because threads execute in blocks and each block executes warp by warp, explicit synchronization of the threads must take place between the discrete phases of the kernel, e.g., between initializing the shared-memory histogram array and starting to calculate the histogram, and so on.
- The global histogram array, i.e., the one that holds the final result, is also updated concurrently by multiple threads. Atomic addition operations have to be employed so that the results stay consistent.
- If the number of bins are known a priori, we can allocate a fixed amount of local memory per block, as shown in [Listing 6.7](#). The `GPU_histogram_static` kernel assumes that its execution configuration calls for a 1D grid of 1D blocks.

```
14 // File: histogram/histogram.cu
15 . . .
16 static const int BINS = 256;
17
18 __global__ void GPU_histogram_static (int *in, int N, int *h)
19 {
20     int gloID = blockIdx.x * blockDim.x + threadIdx.x;
21     int locID = threadIdx.x;
22     int GRIDSIZE = gridDim.x * blockDim.x; // total number of threads
23     __shared__ int localH[BINS];          // shared allocation
24     int i;
25
26     // initialize the local, shared-memory bins
27     for (i = locID; i < BINS; i += blockDim.x)
28         localH[i] = 0;
29
30     // wait for all warps to complete the previous step
```

```

31  __syncthreads ();
32
33  // start processing the image data
34  for (i = gloID; i < N; i += GRIDSIZE)
35  {
36      int temp = in[i];
37      atomicAdd (localH + (temp & 0xFF), 1);
38      atomicAdd (localH + ((temp >> 8) & 0xFF), 1);
39      atomicAdd (localH + ((temp >> 16) & 0xFF), 1);
40      atomicAdd (localH + ((temp >> 24) & 0xFF), 1);
41  }
42
43  // wait for all warps to complete the local calculations , before ↵
44      updating the global counts
45  __syncthreads ();
46
47  // use atomic operations to add the local findings to the global ↵
48      memory bins
49  for (i = locID; i < BINS; i += blockDim.x)
50      atomicAdd (h + i, localH[i]);
51  }

```

LISTING 6.7

CUDA histogram calculation kernel using a static shared memory allocation.

Dynamic allocation

```
81   for (i = locID; i < bins; i += blockDim.x)
82       atomicAdd (h + i, localH[i]);
83 }
```

LISTING 6.8

CUDA histogram calculation kernel, using **dynamic** shared memory allocation.

```
50 // File : histogram/histogram.cu
51 . . .
52 __global__ void GPU_histogram_dynamic (int *in, int N, int *h, int ←
    bins)
53 {
54     int gloID = blockIdx.x * blockDim.x + threadIdx.x;
55     int locID = threadIdx.x;
56     extern __shared__ int localH[];
57     int GRIDSIZE = gridDim.x * blockDim.x;
58     int i;
59
60     // initialize the local bins
61     for (i = locID; i < bins; i += blockDim.x)
62         localH[i] = 0;
63
64     // wait for all warps to complete the previous step
65     __syncthreads ();
66
67     // start processing the image data
68     for (i = gloID; i < N; i += GRIDSIZE)
69     {
70         int temp = in[i];
71         atomicAdd (localH + (temp & 0xFF), 1);
72         atomicAdd (localH + ((temp >> 8) & 0xFF), 1);
73         atomicAdd (localH + ((temp >> 16) & 0xFF), 1);
74         atomicAdd (localH + ((temp >> 24) & 0xFF), 1);
75     }
76
77     // wait for all warps to complete the local calculations , before ←
        updating the global counts
78     __syncthreads ();
79
80     // use atomic operations to add the local findings to the global ←
        memory bins
```

- The only significant difference of kernel GPU_histogram_dynamic from GPU_histogram_static lies in the use of the extern keyword in line 56.
- The reservation of shared memory and the initialization of the localH pointer take place by the CUDA run-time, when the kernel is invoked in line 119.
- The change in the kernel signature was necessitated by the need to also pass the number of bins, which are in turn calculated (line 97) after an image is read (line 88):

```

84 // File: histogram/histogram.cu
85 . . .
86 int main (int argc, char **argv)
87 {
88     PGImage inImg (argv[1]);
89
90     int *d_in, *h_in;
91     int *d_hist, *h_hist, *cpu_hist;
92     int i, N, bins;
93
94     h_in = (int *) inImg.pixels;
95     N = ceil ((inImg.x_dim * inImg.y_dim) / 4.0);
96
97     bins = inImg.num_colors + 1;
98     h_hist = (int *) malloc (bins * sizeof (int));
99     cpu_hist = (int *) malloc (bins * sizeof (int));
100
101     // CPU calculation used for testing
102     CPU_histogram (inImg.pixels, inImg.x_dim * inImg.y_dim, cpu_hist, ←
        bins);
103
104     cudaMalloc ((void **) &d_in, sizeof (int) * N);
105     cudaMalloc ((void **) &d_hist, sizeof (int) * bins);
106     cudaMemcpy (d_in, h_in, sizeof (int) * N, cudaMemcpyHostToDevice);
107     cudaMemset (d_hist, 0, bins * sizeof (int));
108
109     GPU_histogram_static <<< 16, 256 >>> (d_in, N, d_hist);
110     cudaDeviceSynchronize (); // Wait for the GPU launched work to ←
        complete
111
112     cudaMemcpy (h_hist, d_hist, sizeof (int) * bins, ←
        cudaMemcpyDeviceToHost);
113
114     for (i = 0; i < BINS; i++)
115         if (cpu_hist[i] != h_hist[i])
116             printf ("Calculation mismatch (static) at: %i\n", i);
117
118     cudaMemset (d_hist, 0, bins * sizeof (int));

```



```

119 GPU_histogram_dynamic <<< 16, 256, bins * sizeof (int) >>> (d_in, N, ↵
    d_hist, bins);
120 cudaDeviceSynchronize ();      // Wait for the GPU launched work to ↵
    complete
121
122 cudaMemcpy (h_hist, d_hist, sizeof (int) * bins, ↵
    cudaMemcpyDeviceToHost);
123
124 for (i = 0; i < BINS; i++)
125     if (cpu_hist[i] != h_hist[i])
126         printf ("Calculation mismatch (dynamic) at: %i\n", i);
127
128 cudaFree ((void *) d_in);
129 cudaFree ((void *) d_hist);
130 free (h_hist);
131 free (cpu_hist);
132 cudaDeviceReset ();
133
134 return 0;
135 }

```

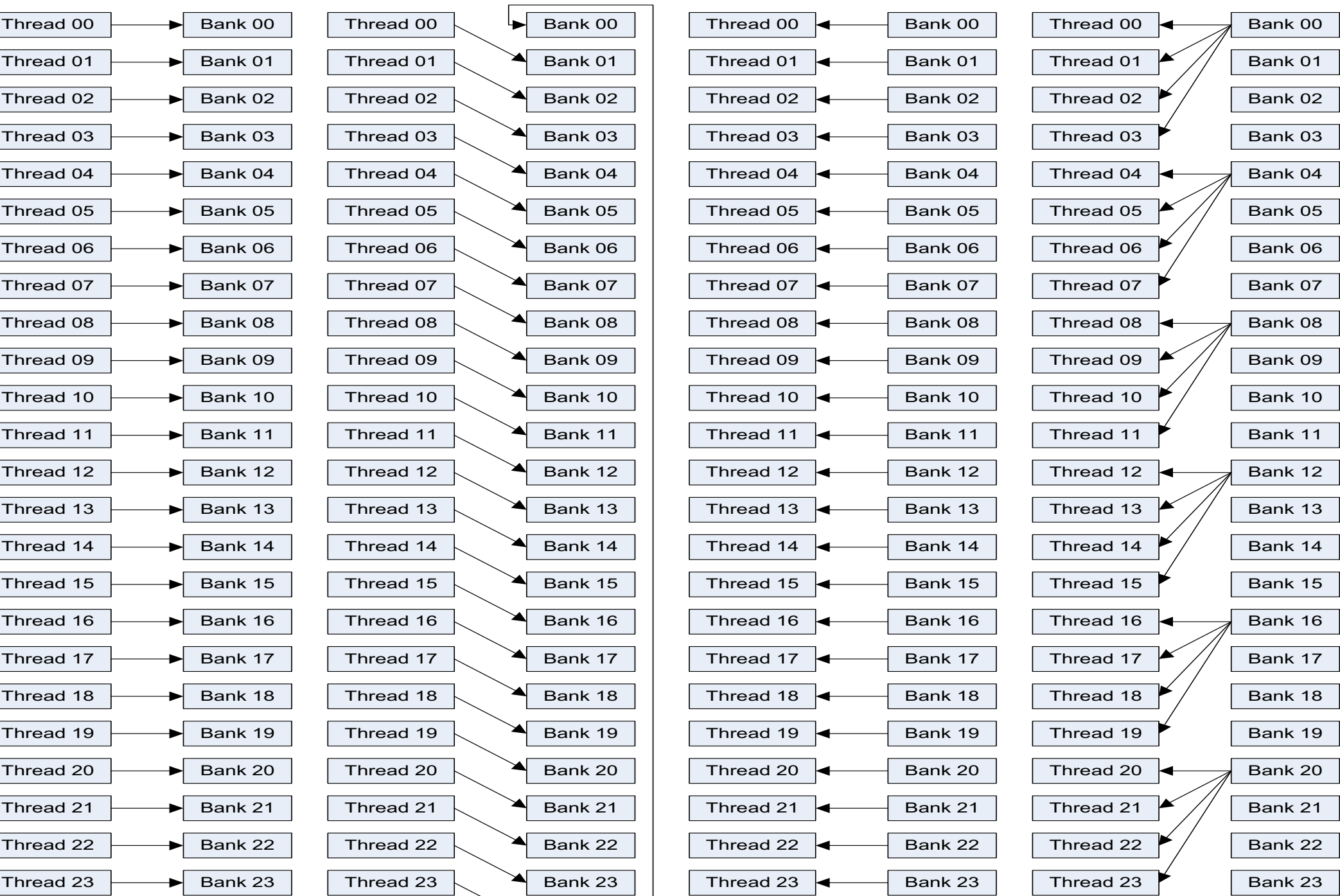
LISTING 6.9

main() function for the memory management and launch of the kernels in [Listings 6.7](#) and [6.8](#).

- Shared memory is effectively a user-controlled L1 cache.
- The L1 cache and shared memory share a 64 K memory segment per SM.
- In Kepler this can be configured in 16 K blocks in favor of the L1 or shared memory as you prefer for your application.
- In Fermi the choice is 16 K or 48K in favor of the L1 or shared memory.
- Pre-Fermi hardware (compute 1.) has a fixed 16 K of shared memory and no L1 cache.
- The shared memory has in the order of 1.5 TB/s bandwidth with extremely low latency.
- Clearly, this is hugely superior to the up to 190 GB/s available from global memory, but around one-fifth of the speed of registers.

- bandwidth figures 1.5 TB/s for shared memory and 190 GB/s for the best global memory access you can see that there is a 7:1 ratio.
- To put it another way, there is potential for a 7 X speedup if you can make effective use of shared memory
- However, the GPU operates a **load-store model of memory**, in that any operand must be loaded into a register prior to any operation.
- Thus, the loading of a value into shared memory, as opposed to just loading it into a register, must be justified by data reuse, coalescing global memory, or data sharing between threads.
- Otherwise, better performance is achieved by directly loading the global memory values into registers.

- Shared memory is a **bank-switched architecture**.
- On Fermi it is 32 banks wide, and on G200 and G80 hardware it is 16 banks wide.
- Each bank of data is 4 bytes in size, enough for a single-precision floating-point data item or a standard 32-bit integer value.
- Kepler also introduces a special 64 bit wide mode so larger double precision values no longer span two banks.
- Each bank can service only a single operation per cycle, regardless of how many threads initiate this action.
- Thus, if every thread in a warp accesses a separate bank address, every thread's operation is processed in that single cycle.



- Note there is no need for a one-to-one sequential access, just that every thread accesses a separate bank in the shared memory.
- There is, effectively, a crossbar switch connecting any single bank to any single thread.
- This is very useful when you need to swap the words, for example, in a sorting algorithm, an example of which we'll look at later.

- There is also one other very useful case with shared memory and that is where every thread in a warp reads the **same** bank address.
- As with constant memory, this triggers a broadcast mechanism to all threads within the warp.
- Usually thread zero writes the value to communicate a common value with the other threads in the warp.

- However, if we have any other pattern, we end up with bank conflicts of varying degrees.
- This means you stall the other threads in the warp that idle while the threads accessing the shared memory address queue up one after another.
- One important aspect of this is that it is not hidden by a switch to another warp, so we do in fact stall the SM.
- Thus, bank conflicts are to be avoided if at all possible as the SM will idle until all the bank requests have been fulfilled.

- The worst case is where every thread writes to the same bank, in which case we get 32 serial accesses to the same bank.
- We see this typically where the thread accesses a bank by a stride other than 32.
- Where the stride decreases by a power of two (e.g., in a parallel reduction), we can also see this, with each successive round causing more and more bank conflicts

Sorting using shared memory

- Let's introduce a practical example here, using sorting. A sorting algorithm works by taking a random dataset and generating a sorted dataset.
- We thus need N input data items and N output data items. The key aspect with sorting is to ensure you minimize the number of reads and writes to memory.
- Many sorting algorithms are actually **multipass**, meaning we read every element of N , M times, which is clearly not good.
- The quicksort algorithm is the preferred algorithm for sorting in the serial world. Being a divide-and-conquer algorithm, it would appear to be a good choice for a parallel approach.
- However, by default it uses recursion, which is only supported in CUDA compute 2.x devices.
- Typical parallel implementations spawn a new thread for every split of the data. The current CUDA model requires a specification of the total number of threads at kernel launch, or a series of kernel launches per level.
- The data causes significant branch divergence, which again is not good for GPUs.
- There are ways to address some of these issues. However, these issues make quicksort not the best algorithm to use on a pre-Kepler GK110/ Tesla K20 GPU. **In fact, you often find the best serial algorithm is not the best parallel algorithm and it is better to start off with an open mind about what will work best.**

- One common algorithm found in the parallel world is the **merge sort** (Figure 6.3).
- It works by recursively partitioning the data into small and smaller packets, until eventually you have only two values to sort. Each sorted list is then merged together to produce an entire sorted list.
- Recursion is not supported in CUDA prior to compute 2., so how can such an algorithm be performed? Any recursive algorithm will at some point have a dataset of size N .
- On GPUs the thread block size or the warp size is the ideal size for N . Thus, to implement a recursive algorithm all you have to do is break the data into blocks of 32 or larger elements as the smallest case of N .
- With merge sort, if you take a set of elements such as **$\{1,5,2,8,9,3,2,1\}$** we can split the data at element four and obtain two datasets, $\{1,5,2,8\}$ and $\{9,3,2,1\}$. You can now use two threads to apply a sorting algorithm to the two datasets. Instantly you have gone from $p = 1$ to $p = 2$, where p is the number of parallel execution paths.

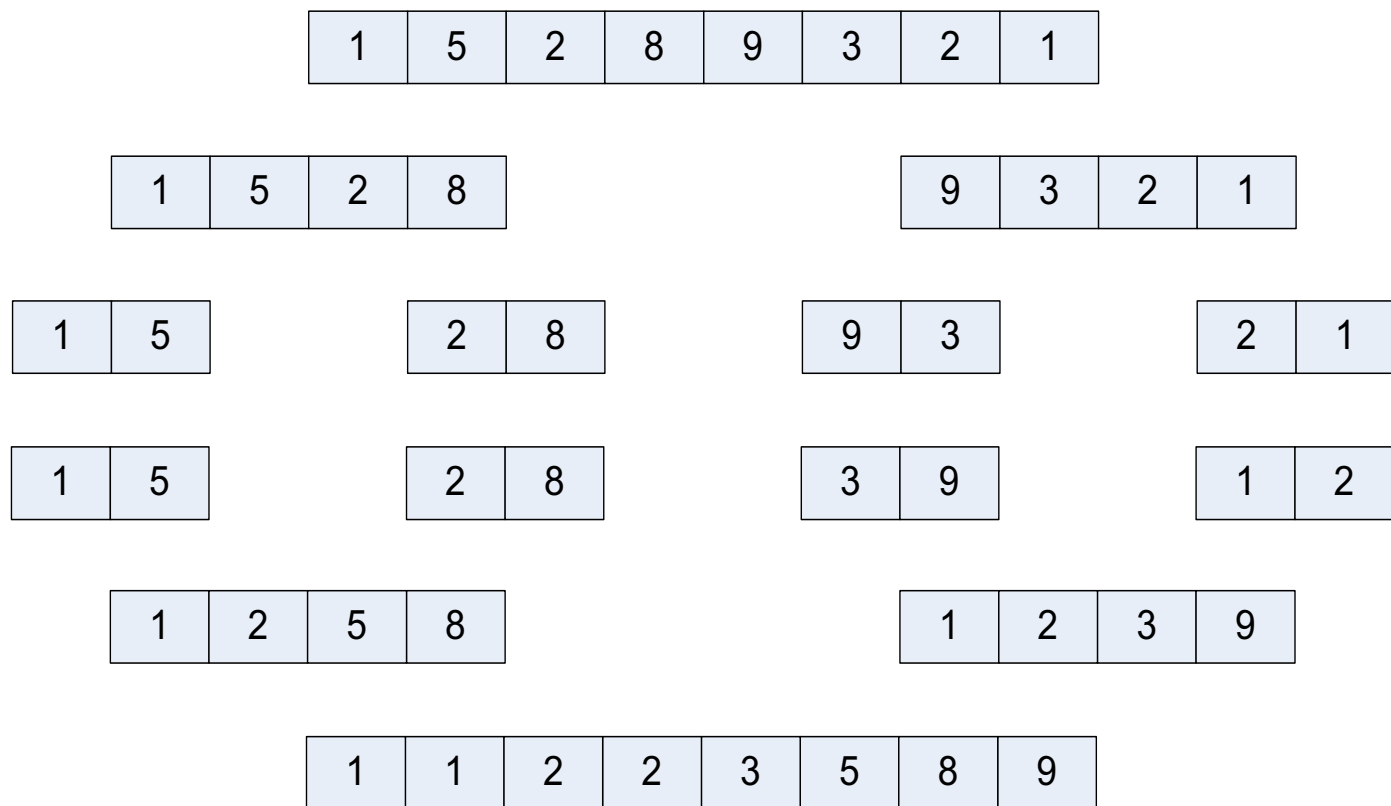


FIGURE 6.3

Simple merge sort example.

- Splitting the data from two sets into four sets gives you $\{1,5\}$, $\{2,8\}$, $\{9,3\}$, and $\{2,1\}$. It's now trivial to execute four threads, each of which compares the two numbers and swaps them if necessary. Thus, you end up with four sorted datasets: $\{1,5\}$, $\{2,8\}$, $\{3,9\}$, and $\{1,2\}$. The sorting phase is now complete.
- The maximum parallelism that can be expressed in this phase is **$N/2$ independent threads**.
- Thus, with a 512 MB dataset, you have 128K 32-bit elements, for which we can use a maximum of 64K threads ($N = 128K$, $N/2 = 64K$).
- Since a GTX580 GPU has 16 SMs, each of which can support up to 1536 threads, we get up to 24K threads supported per GPU. With around **two and a half passes**, you can therefore iterate through the 64K data pairs that need to be sorted with such a decomposition.

- However, you now run into the classic problem with merge sort, the **merge phase**. Here the lists are combined by moving the smallest element of each list into the output list.
- This is then repeated until all members of the input lists are consumed. With the previous example, the sorted lists are $\{1,5\}$, $\{2,8\}$, $\{3,9\}$, and $\{1,2\}$.
- In a traditional merge sort, these get combined into $\{1,2,5,8\}$ and $\{1,2,3,9\}$. These two lists are then further combined in the same manner to produce one final sorted list, $\{1,1,2,2,3,5,8,9\}$.

- Thus, **as each merge stage is completed, the amount of available parallelism halves**. As an alternative approach where N is small, you can simply scan N sets of lists and immediately place the value in the correct output list, skipping any intermediate merge stages as shown in [Figure 6.4](#).
- The issue is that the sort performed at the stage highlighted for elimination in [Figure 6.4](#) is typically done with two threads.
- **As anything below 32 threads means we're using less than one warp, this is inefficient on a GPU.**

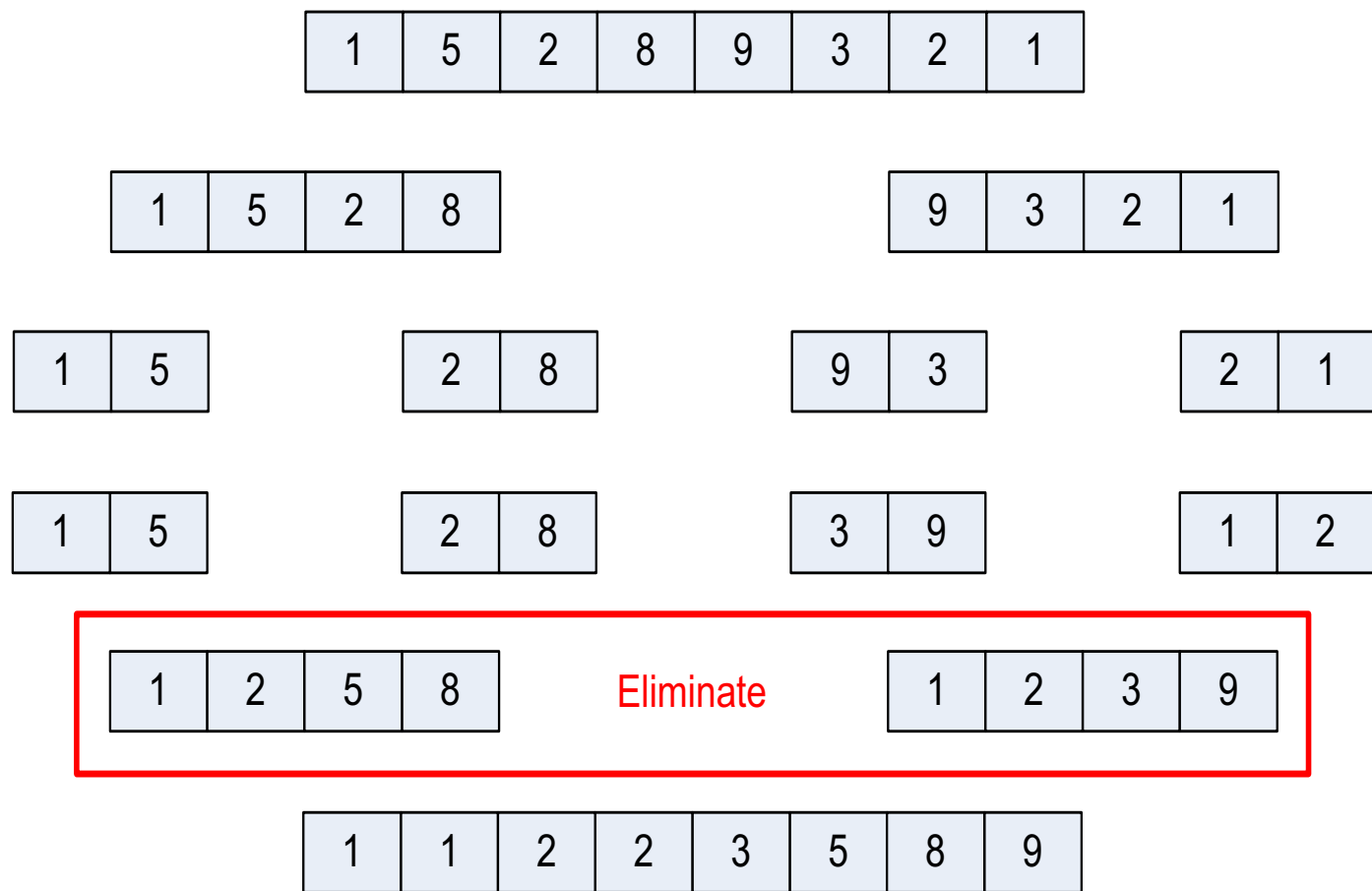


FIGURE 6.4

Merging N lists simultaneously.

- The downside of this approach is that it means you would need to read the first element of the sorted list set from every set.
- With 64 K sets, this is 64 K reads, or 256 MB of data that has to be fetched from memory.
- Clearly, this is not a good solution when the number of lists is very large.

- Thus, our approach is to try to achieve a much better solution to the merge problem by limiting the amount of recursion applied to the original problem and **stopping at the number of threads in a warp, 32**, instead of two elements per sorted set, as with a traditional merge sort.
- This reduces the number of sets in the previous example from 64 K sorted sets to just 4 K sets.
- It also increases the maximum amount of parallelism available from $N/2$ to $N/32$.
- In the 128 K element example we looked at previously, this would mean we would need 4 K processing elements. This would distribute 256 processing elements (warps) to every SM on a GTX580.
- As each Fermi SM can execute a maximum of 48 warps, multiple blocks will need to be iterated through, which allows for smaller problem sizes and speedups on future hardware. See [Figure 6.5](#).

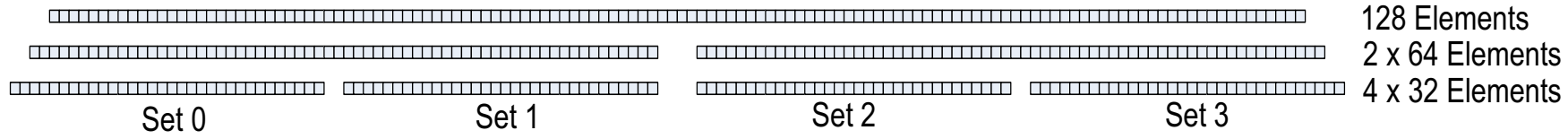


FIGURE 6.5

Shared memory–based decomposition.

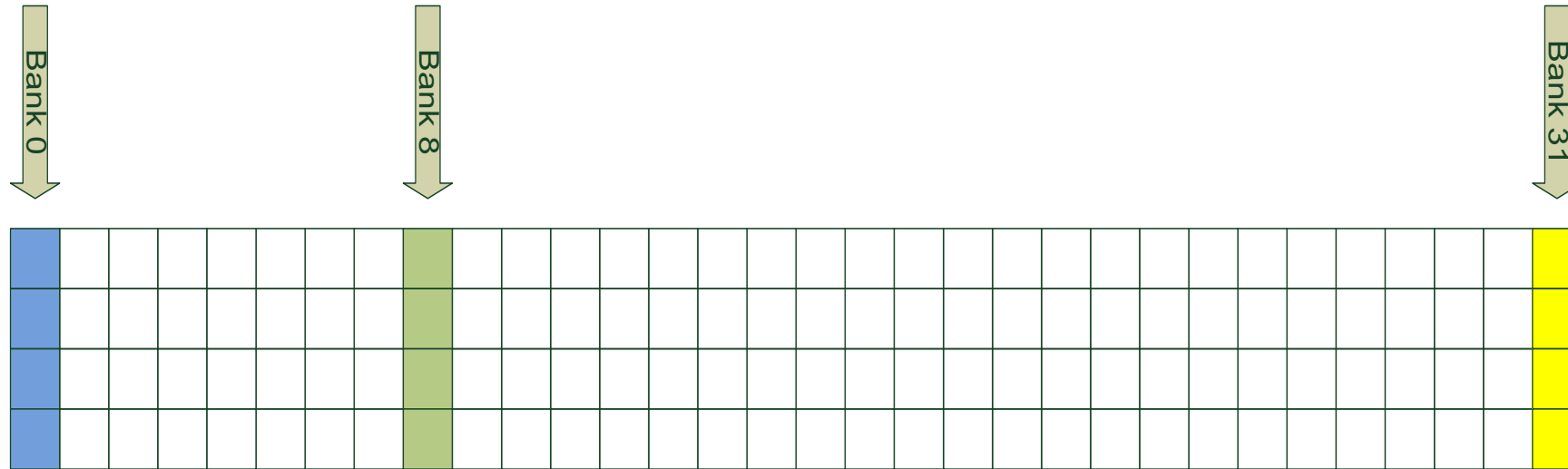


FIGURE 6.6

Shared memory bank access.

- **Shared memory is bank switched.** We have 32 threads within a single warp. However, if any of those threads access the same bank, there will be a **bank conflict**.
- If any of the **threads diverge in execution flow**, you could be running at up to $1/32$ of the speed in the worst case.
- Threads can use registers that are private to a thread. They can only communicate with one another using shared memory.
- By arranging a dataset in rows of 32 elements in the shared memory, and accessing it in columns by thread, you can achieve bank conflict-free access to the memory (Figure 6.6).

CONSTANT MEMORY

- Constant memory is a form of virtual addressing of global memory.
- There is no special reserved constant memory block.
- **Constant memory has two special properties** you might be interested in.
 - First, it is cached,
 - and second, it supports broadcasting a single value to all the elements within a warp.
- Constant memory, as its name suggests, is for **read-only memory**.
- This is memory that is either declared at compile time as read only or defined at runtime as read only by the host.
- It is, therefore, constant only in respect of the GPU's view onto memory.
- The size of constant memory is restricted to **64 K**.

- To declare a section of memory as constant at compile time, you simply use the `__constant__` keyword. For example:

```
__constant__ float my_array[1024] = { 0.0F, 1.0F, 1.34F, . . .};
```

- To change the contents of the constant memory section at runtime, you simply use the `cudaCopyToSymbol` function call prior to invoking the GPU kernel.
- If you do not define the constant memory at either compile time or host runtime then the contents of the memory section are undefined.

Constant memory caching

Compute 1.x devices

- On compute 1.x devices (pre-Fermi), constant memory has the property of being cached in a small 8K L1 cache, so **subsequent** accesses can be very fast.
- This is providing that there is some potential for **data reuse** in the memory pattern the application is using. It is also highly optimized for **broadcast access** such that threads accessing the same memory address can be serviced in a single cycle.
- With a 64 K segment size and an 8 K cache size, you have an 8:1 ratio of memory size to cache, which is really very good.
- If you can contain or localize accesses to 8 K chunks within this constant section you'll achieve very good program performance. On certain devices you will find localizing the data to even smaller chunks will provide higher performance.

- With a nonuniform access to constant memory a cache miss results in N fetches from global memory in addition to the fetch from the constant cache.
- Thus, a memory pattern that **exhibits poor locality and/or poor data reuse** should not be accessed as constant memory.
- Also, each **divergence in the memory fetch** pattern causes serialization in terms of having to wait for the constant memory.
- Thus, a warp with 32 separate fetches to the constant cache would take at least 32 times longer than an access to a single data item. This would grow significantly if it also included cache misses.

- Single-cycle access is a huge improvement on the several hundred cycles required for a fetch from global memory.
- However, the several hundred-cycle access to global memory will likely be hidden by task switches to other warps, if there are enough available warps for the SM to execute.
- **Thus, the benefit of using constant memory for its cache properties relies on the time taken to fetch data from global memory and the amount of data reuse the algorithm has.**
- As with shared memory, the low-end devices have much less global memory bandwidth, so they benefit proportionally more from such techniques than the high-end devices.

- Most algorithms can have their data broken down into “tiles” (i.e., smaller datasets) from a much larger problem.
- In fact, as soon as you have a problem that can’t physically fit on one machine, you have to do tiling of the data.
- The same tiling can be done on a multicore CPU with each one of the N cores taking $1/N$ of the data.
- You can think of each SM on the GPU as being a core on a CPU that is able to support hundreds of threads.

- Imagine overlaying a grid onto the data you are processing where the total number of cells, or blocks, in the grid equals the number of cores (SMs) you wish to split the data into.
- Take these SM- based blocks and further divide them into at least eight additional blocks. You've now decomposed your data area into N SMs, each of which is allocated M blocks.
- In practice, this split is usually too large and would not allow for future generations of GPUs to increase the number of SMs or the number of available blocks and see any benefit.
- It also does not work well where the number of SMs is unknown, for example, when writing a commercial program that will be run on consumer hardware. The largest number of SMs per device to date has been 32 (GT200 series).
- The Kepler and Fermi range aimed at compute have a maximum of 15 and 16 SMs respectively. The range designed primarily for gaming have up to 8 SMs.

- One other important consideration is what **interthread communication** you need, if any. This can only reasonably be done using threads and these are limited to 1024 per block on Fermi and Kepler, less on earlier devices. You can, of course, process multiple items of data per thread, so this is not such a hard limit as it might first appear.
- Finally, you need to consider **load balancing**. Many of the early card releases of GPU families had non power of two numbers of SMs (GTX460 1/4 7, GTX260 1/4 30, etc.). Therefore, using too few blocks leads to too little granularity and thus unoccupied SMs in the final stages of computation.

- **Tiling**, in terms of constant memory, means splitting the data into blocks of no more than 64 K each. Ideally, the tiles should be 8 K or less.
- Sometimes tiling involves having to deal with halo or ghost cells that occupy the boundaries, so values have to be propagated between tiles.
- Where halos are required, larger block sizes work better than smaller cells because the area that needs to be communicated between blocks is much smaller.
- When using tiling there is actually quite a lot to think about. Often the best solution is simply to run through all combinations of number of threads, elements processed per thread, number of blocks, and tile widths, and search for the optimal solution for the given problem.

Compute 2.x devices

- On Fermi (compute 2.x) hardware and later, there is a level two (L2) cache. Fermi uses an L2 cache shared between each SM.
- All memory accesses are cached automatically by the L2 cache.
- Additionally, the L1 cache size can be increased from 16 K to 48 K by sacrificing 32 K of the shared memory per SM.
- Because all memory is cached on Fermi, how constant memory is used needs some consideration.

- Fermi, unlike compute 1.x devices, allows **any** constant section of data to be treated as constant memory, even if it is not explicitly declared as such.
- Constant memory on 1.x devices has to be explicitly managed with special-purpose calls like `cudaMemcpyToSymbol` or declared at compile time.
- With Fermi, any nonthread-based access to an area of memory declared as constant (simply with the standard `const` keyword) goes through the constant cache.
- By nonthread-based access, this is an access that does not include `threadIdx.x` in the array indexing calculation.

- If you need access to constant data on a per-thread-based access, then you need to use the compile time (`__constant__`) or runtime function (`cudaMemcpyToSymbol`) as with compute 1.x devices.
- However, be aware that the L2 cache will still be there and is much larger than the constant cache. If you are implementing a tiling algorithm that needs halo or ghost cells between blocks, the solution will often involve copying the halo cells into constant or shared memory.
- Due to Fermi's L2 cache, this strategy will usually be slower than simply copying the tiled cells to shared or constant memory and then accessing the halo cells from global memory.
- The L2 cache will have collected the halo cells from the prior block's access of the memory. Therefore, the halo cells are quickly available from the L2 cache and come into the device much quicker than you would on compute 1.x hardware where a global memory fetch would have to go all the way out to the global memory.

Constant memory broadcast

- Constant memory has one very useful feature. It can be used for the purpose of distributing, or broadcasting, data to every thread in a warp.
- This broadcast takes place in just a single cycle, making this ability very useful.
- In comparison, a coalesced access to global memory on compute 1.x hardware would require a memory fetch taking hundreds of cycles of latency to complete. Once it has arrived from the memory subsystem, it would be distributed in the same manner to all threads, but only after a significant wait for the memory subsystem to provide the data. Unfortunately, this is an all too common problem, in that memory speeds have failed to keep pace with processor clock speeds.

- By using the broadcast mechanism, which is also present on Fermi for L2 cache–based accesses, you can distribute data very quickly to multiple threads within a warp.
- This is particularly useful where you have some common transformation being performed by all threads.
- Each thread reads element N from constant memory, which triggers a broadcast to all threads in the warp.
- Some processing is performed on the value fetched from constant memory, perhaps in combination with a read/write to global memory. You then fetch element $N + 1$ from constant memory, again via a broadcast, and so on. As the constant memory area is providing almost L1 cache speeds, this type of algorithm works well.

- However, be aware that if a constant is really a literal value, it is better to define it as a literal value using a `#define` statement, as this frees up constant memory.
- So don't place literals like PI into constant memory, rather define them as literal `#define` instead. In practice, it makes little difference in speed, only memory usage, as to which method is chosen.

Global Memory

- GPU global memory is global because it's writable from both the GPU and the CPU.
- It can actually be accessed from any device on the PCI-E bus.
- GPU cards can transfer data to and from one another, directly, without needing the CPU.
- The memory from the GPU is accessible to the CPU host processor in one of three ways:
 - Explicitly with a blocking transfer.
 - Explicitly with a nonblocking transfer.
 - Implicitly using zero memory copy.
- The memory on the GPU device sits on the other side of the PCI-E bus.
- This is a bidirectional bus that, in theory, supports transfers of up to 8 GB/s (PCI-E 2.0) in each direction.
- In practice, the PCI-E bandwidth is typically 4–5 GB/s in each direction.

- The usual model of execution involves the CPU transferring a block of data to the GPU, the GPU kernel processing it, and then the CPU initiating a transfer of the data back to the host memory.
- A slightly more advanced model of this is where we use streams (covered later) to overlap transfers and kernels to ensure the GPU is always kept busy, as shown in [Figure 6.16](#).
- [Figure 6.16](#), the memory accesses are pipelined.
- By creating a ratio of typically 10:1 of threads to number of memory accesses, you can hide memory latency, but only if you access global memory in a pattern that is coalesced.

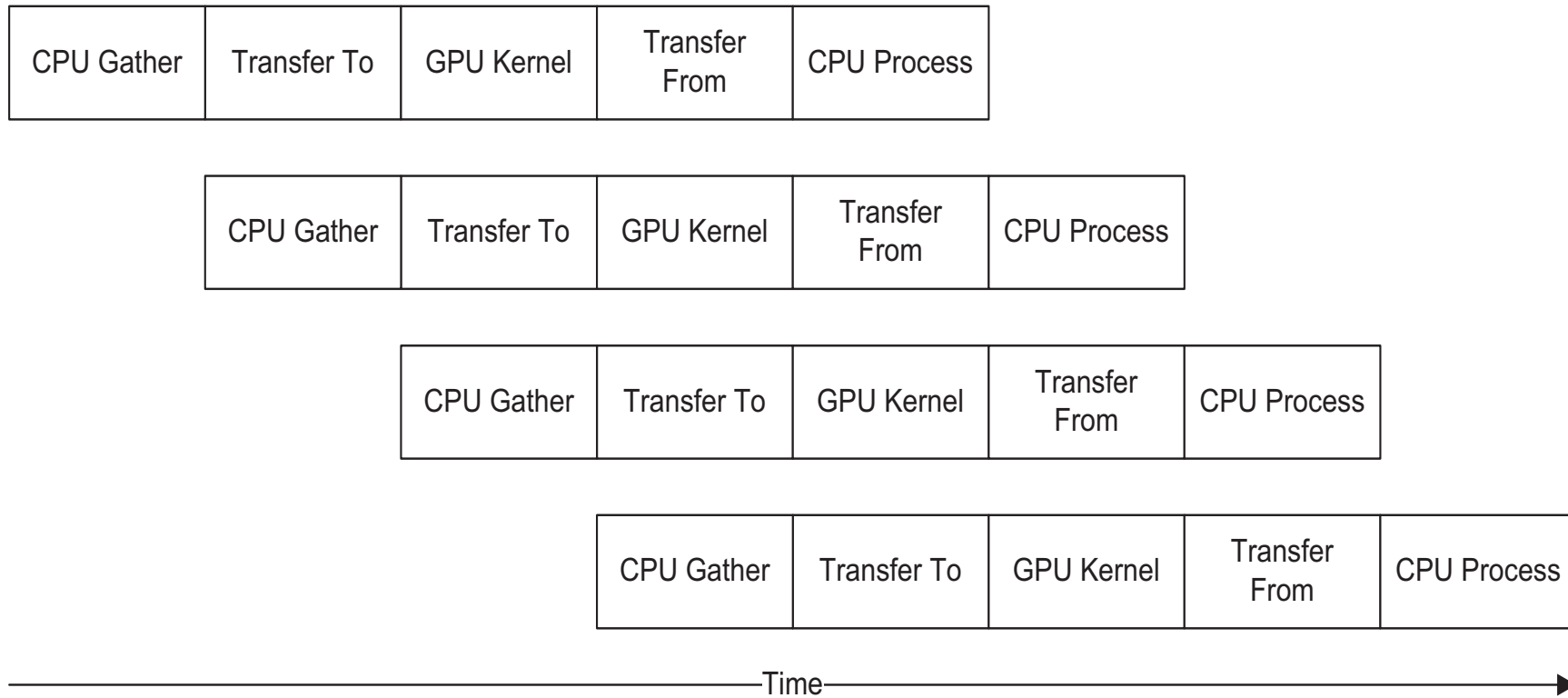


FIGURE 6.16

Overlapping kernel and memory transfers.

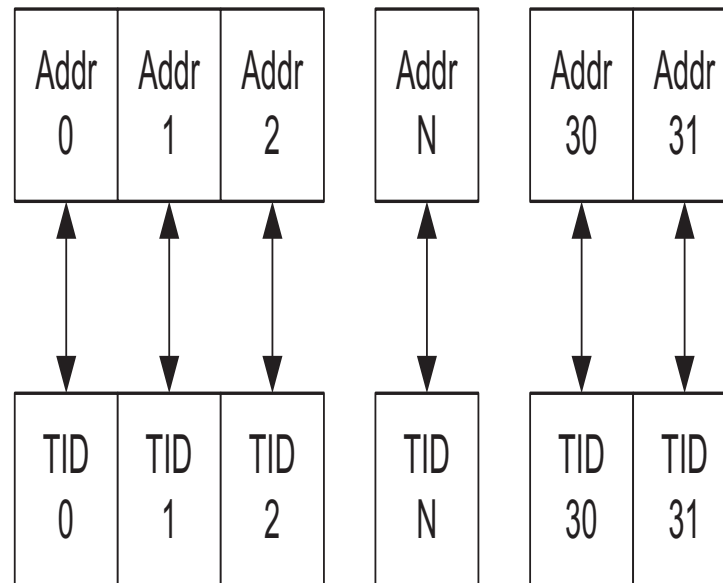
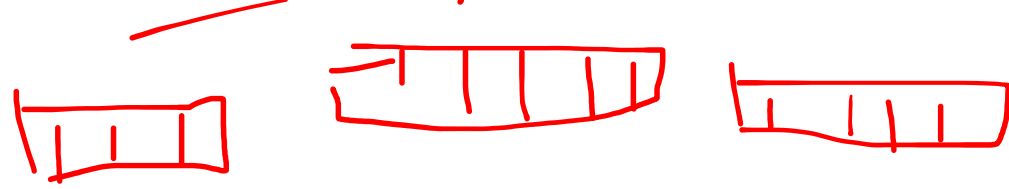


FIGURE 6.17

Addresses accessed by thread ID.



- Here we have shown Addr as the logical address offset from the base location, assuming we are accessing byte-based data.
- TID represents the thread number.
- If we have a one-to-one sequential and aligned access to memory, the address accesses of each thread are combined together and a single memory transaction is issued.
- Assuming we're accessing a single precision float or integer value, each thread will be accessing 4 bytes of memory.
- Memory is coalesced on a warp basis (the older G80 hardware uses half warps), meaning we get $32 * 4 = 128$ byte access to memory.

- Coalescing sizes supported are 32, 64, and 128 bytes, meaning warp accesses to byte, 16- and 32- bit data will always be coalesced if the access is a sequential pattern and aligned to a 32-byte boundary.
- The alignment is achieved by using a special malloc instruction, replacing the standard cudaMalloc
- with cudaMallocPitch, which has the following syntax:

```
extern __host__ cudaError_t CUDARTAPI cudaMallocPitch(void **devPtr, size_t *pitch,  
size_t width, size_t height);
```

This translates to cudaMallocPitch (pointer to device memory pointer, pointer to pitch, desired width of the row in bytes, height of the array in bytes).

- Thus, if you have an array of 100 rows of 60 float elements, using the conventional `cudaMalloc`, you would allocate $100 * 60 * \text{sizeof(float)}$ bytes, or $100 * 60 * 4 = 24,000$ bytes.
- Accessing array index `[1][0]` (i.e., row one, element zero) would result in noncoalesced access. This is because the length of a single row of 60 elements would be 240 bytes, which is of course not a power of two.
- The first address in the series of addresses from each thread would not meet the alignment requirements for coalescing.
- Using the `cudaMallocPitch` function the size of each row is padded by an amount necessary for the alignment requirements of the given device ([Figure 6.18](#)).
- In our example, it would in most cases be extended to 64 elements per row, or 256 bytes. The pitch the device actually uses is returned in the pitch parameters passed to `cudaMallocPitch`.

- Nonaligned accesses result in multiple memory fetches being issued. While waiting for a memory fetch, all threads in a warp are stalled until all memory fetches are returned from the hardware.
- Thus, to achieve the best throughput you need to issue a small number of large memory fetch requests, as a result of aligned and sequential coalesced accesses.

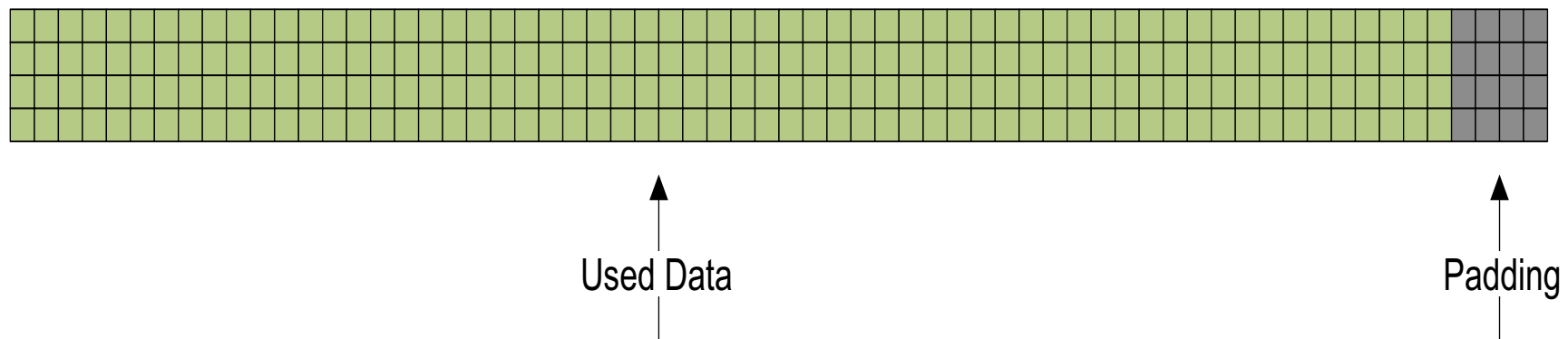
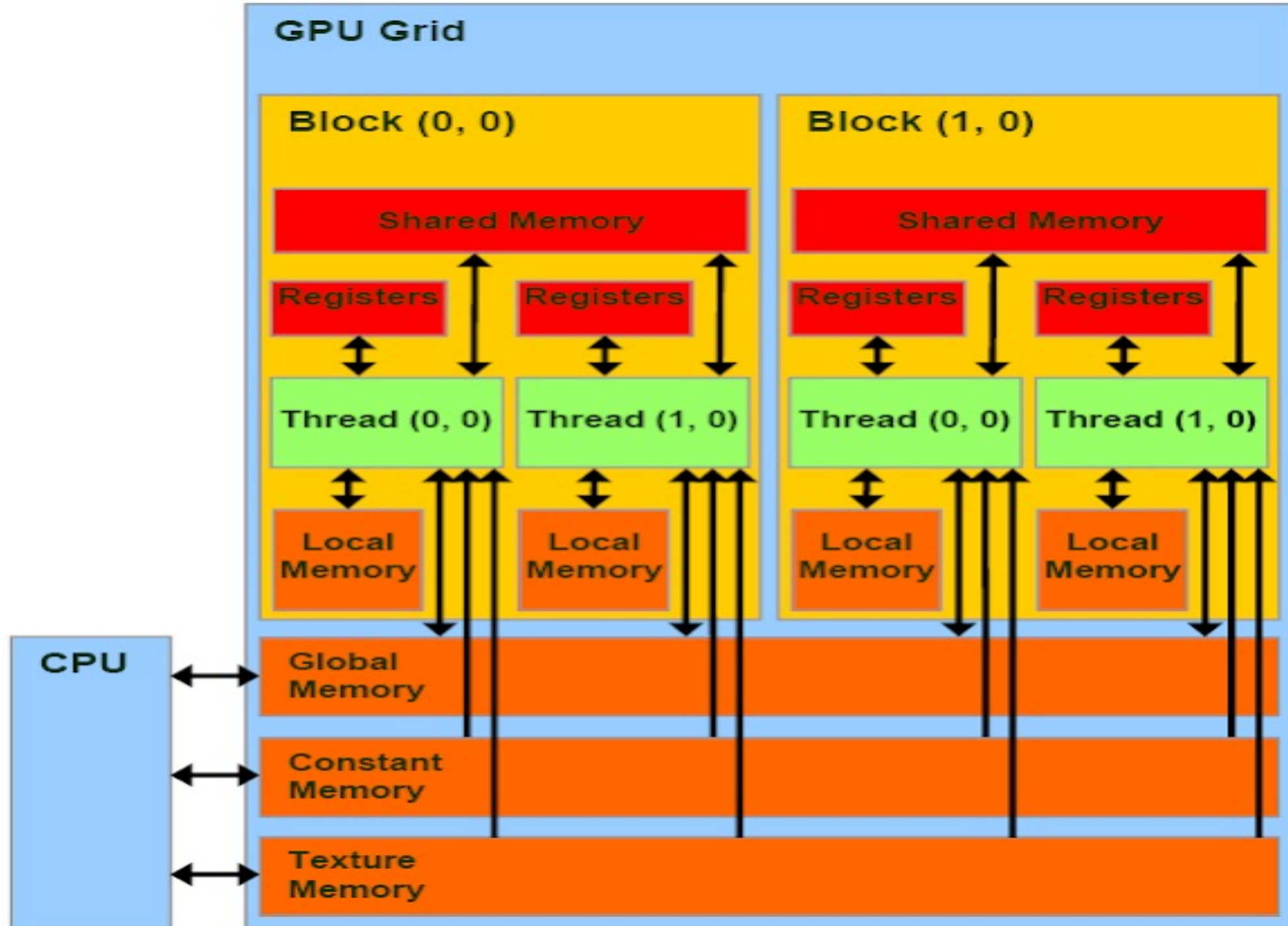


FIGURE 6.18

Padding achieved with `cudaMallocPitch`.



Registers

- Registers are fast on-chip memories that are used to store operands for the operations executed by the computing cores.
- Registers are local to a thread, and each thread has exclusive access to its own registers: values in registers cannot be accessed by other threads, even from the same block, and are not available for the host.
- Registers are also not permanent, therefore data stored in registers is only available during the execution of a thread.

- REGISTERS ARE THE FASTEST MEMORY ON THE GPU, SO USING THEM TO INCREASE DATA REUSE IS AN IMPORTANT PERFORMANCE OPTIMIZATION

Global Memory

- Global memory can be considered the main memory space of the GPU in CUDA. It is allocated, and managed, by the host, and it is accessible to both the host and the GPU, and for this reason the global memory space can be used to exchange data between the two.
- It is the largest memory space available, and therefore it can contain much more data than registers, but it is also slower to access. This memory space does not require any special memory space identifier.
- Memory allocated on the host, and passed as a parameter to a kernel, is by default allocated in global memory.
- Global memory is accessible by all threads, from all thread blocks. This means that a thread can read and write any value in global memory.

- WHILE GLOBAL MEMORY IS VISIBLE TO ALL THREADS, REMEMBER THAT GLOBAL MEMORY IS NOT COHERENT, AND CHANGES MADE BY ONE THREAD BLOCK MAY NOT BE AVAILABLE TO OTHER THREAD BLOCKS DURING THE KERNEL EXECUTION. HOWEVER, ALL MEMORY OPERATIONS ARE FINALIZED WHEN THE KERNEL TERMINATES.

Local Memory

- Memory can also be statically allocated from within a kernel, and according to the CUDA programming model such memory will not be global but *local* memory.
- Local memory is only visible, and therefore accessible, by the thread allocating it. So all threads executing a kernel will have their own privately allocated local memory.
- Local memory is not not a particularly fast memory, and in fact it has similar throughput and latency of global memory, but it is much larger than registers.
- As an example, local memory is automatically used by the CUDA compiler to store spilled registers, i.e. to temporarily store variables that cannot be kept in registers anymore because there is not enough space in the register file, but that will be used again in the future and so cannot be erased.

KEY POINTS

- “Registers can be used to locally store data and avoid repeated memory operations”
- “Global memory is the main memory space and it is used to share data between host and GPU”
- “Local memory is a particular type of memory that can be used to store data that does not fit in registers and is private to a thread”

Texture Memory

- Texture memory can be used for two primary purposes:
 - Caching on compute 1.x and 3.x hardware.
 - Hardware-based manipulation of memory reads.
- *Texture memory* may also be Like constant memory, **texture memory is cached on chip**, so in some situations it will provide higher effective bandwidth by reducing memory requests to off-chip DRAM.
- Specifically, texture caches are designed for graphics applications where memory access patterns exhibit a great deal of ***spatial locality***. **In a computing application, this roughly implies that a thread is likely to read from an address “near” the address that nearby threads read,**

Texture Memory

- As compute 1.x hardware has no cache to speak of, the 6–8K of texture memory per SM provides the only method to truly cache data on such devices.
- However, with the advent of Fermi and its up to 48 K L1 cache and up to 768 K shared L2 cache, this made the usage of texture memory for its cache properties largely obsolete.
- The texture cache is still present on Fermi to ensure backward compatibility with previous generations of code.

- The texture cache is optimized for locality, that is, it expects data to be provided to adjacent threads.
- This is largely the same cache policy as the L1 cache on Fermi.
- Unless you are using the other aspects of texture memory, texture memory brings you little benefit for the considerable programming effort required to use it on Fermi.
- However, on Kepler, the texture cache gets a special compute path, removing the complexity associated with programming it.

- On compute 1.x hardware, however, the texture cache can be of considerable use.
- If you consider a memory read that exhibits some locality, you can save a considerable number of memory fetches.
- Suppose we needed to perform a gather operation from memory, that is, to read an **out-of-sequence set of memory** addresses into N threads.
- Unless the thread pattern creates an aligned and sequential memory pattern, the **coalescing hardware will issue multiple reads**.
- **If we instead load the data via the texture memory, most of the reads will hit the texture cache, resulting in a considerable performance benefit.**

- You can, of course, equally use shared memory for this purpose, reading in a coalesced way from memory and then performing a read from the shared memory.
- As the shared memory of a compute 1.x device is limited to 16 K, you may decide to allocate shared memory to a specific purpose and use texture memory where the **memory pattern is not so deterministic**.

Hardware manipulation of memory fetches

- The second and perhaps more useful aspect of texture-based memory is that it allows some of the hardware aspects of GPUs to be automatically applied when accessing memory cells.
- **One useful feature is a low-resolution linear interpolation in hardware.**
- Typically, linear interpolation is used to represent a function where the output is not easy or is computationally expensive to express mathematically.
- Thus, the input from a sensor might have a correction applied to its value at the low or high end of its range.
- Rather than model this you simply place a number of points in a table that represent discrete values across the range. For the points falling between the real points you use linear interpolation to work out the approximate value.

- Consider an interpolation table of
- $P = 10, 20, 40, 50, 20$
- $X = 0, 2, 4, 6, 8$
- If we have a new value, 5 for X , what is its interpolated value of P ? The value 5 falls exactly halfway between the two points we have defined, 2 and 4. As the value for 2 is 20 and the value for 4 is 40 we can easily calculate the value for 5 as 30. See [Figure 6.28](#).
- With texture memory, you can set it up such that P is defined as an array normalized from the value 0 to 1 or 1 to $p1$. Fetches are then automatically interpolated in hardware. Combined with the cache properties, this can be a quick method of handling data that is not easily represented as a pure calculation. Bilinear and trilinear interpolation in hardware is also supported for two-dimensional and three-dimensional arrays, respectively.

- One other nice feature of textures is the automatic handling of boundary conditions on array indexes.
- You can configure the handling of texture arrays to either wrap around or clamp at the array boundary.
- This can be useful, as it allows the normal case to be handled for all elements without having to embed **special edge handling code**.
- Special case code typically causes thread divergence and may not be necessary at all with the caching features of Fermi

Restrictions using textures

- Textures come from the graphics world of the GPU and therefore are less flexible than the standard CUDA types. Textures must be declared as a fixed type, i.e. one of the various aligned vector types(u8, u16, u32, s8, s16, s32) at compile time.
- How the values are interpreted is specified at runtime. Texture memory is read only to the GPU kernel and must be explicitly accessed via a special texture API (e.g., `tex1Dfetch()`, etc.) and arrays bound to textures.
- Textures have their uses, especially on compute 1.x hardware.

OPTIMIZATION TECHNIQUES

1.BLOCK AND GRID DESIGN

- The optimization of the grid and block design depends on both the data layout of the problem and the hardware capabilities of the target device.
- In that regard, it is advisable to design an algorithm that adapts to the available CUDA hardware and dynamically deploys the threads as required to minimize the total execution time.
- One of the aspects of the execution minimization problem is that the available computing hardware should be “occupied/busy” as much as possible.

- Two things must happen:
 - 1. Enough work should be *assigned* to the CUDA cores (deployment/execution configuration phase).**
 - 2. The assigned work should allow *execution* with the minimum amount of stalling due to resource contention or slow memory access (execution phase).**
- The reason is that the number of dimensions has no effect on the execution efficiency. The total number of threads per block and the total number of blocks in the grid do.

- The first step in designing a host front-end function that adjusts the execution configuration to the capabilities of the target device is the discovery of said capabilities and primarily the number of available SMs.
- Central to querying the capabilities of a device is the **cudaDeviceProp** structure, which is populated by a call to the **cudaGetDeviceProperties()** function¹⁷:

```

cudaError_t cudaGetDeviceProperties(
    struct cudaDeviceProp *prop, // Pointer to structure for ↵
    storing info
    int device);                // Integer identifying the ↵
                                device to be queried. For systems with one GPU, 0 can ↵
                                be used.

```

17. <http://docs.nvidia.com/cuda/cuda-driver-api/index.html>

Here is a small subset of the fields included in the `cudaDeviceProp` structure (the field names are self-explanatory):

```
struct cudaDeviceProp{
    char name[256]; // A string identifying the device
    int major;      // Compute capability major number
    int minor;      // Compute capability minor number
    int maxGridSize [3];
    int maxThreadsDim [3];
    int maxThreadsPerBlock;
    int maxThreadsPerMultiProcessor;
    int multiProcessorCount;
    int regsPerBlock; // Number of registers per block
    size_t sharedMemPerBlock;
    size_t totalGlobalMem;
    int warpSize;
    . . .
};
```


The following is a generic approach that could be used for listing the names of the GPUs in a system equipped with multiple devices:

```
int deviceCount = 0;
cudaGetDeviceCount(&deviceCount);
if(deviceCount == 0)
    printf("No CUDA compatible GPU exists.\n");
else
{
    cudaDeviceProp pr;
    for(int i=0;i<deviceCount;i++)
    {
        cudaGetDeviceProperties(&pr, i);
        printf("Dev #%i is %s\n", i, pr.name);
    }
}
```

No. of Blocks

- we need at least as many blocks as the number of SMs.
- However, limiting this number to exactly the number of SMs may not be possible, depending on how many resources (e.g., registers and shared memory) each block will need to consume.
- A sound idea would be to have the **block number be a multiple of the SMs**, possibly giving a better balance of computational work assignments.
- If the number of blocks cannot reach the number of SMs, another kernel could possibly run on the remaining SMs by scheduling it on a separate stream

- One of the myths surrounding kernel deployment optimization is that the best approach is the one maximizing occupancy

In fact, Nvidia calls *occupancy* the ratio of resident warps over the maximum possible resident warps:

$$occupancy = \frac{resident_warps}{maximum_warps} \quad (6.1)$$

- A key point here is that there are cases in which **lower occupancy and a smaller number of threads per block can provide better performance** if the kernel is properly designed to take advantage of the more available resources per thread.

Summarizing the general guidelines that one should follow:

1. Do more parallel work per thread. Ideally, this work should be composed of items that can be executed concurrently by the warp schedulers.
2. Use more registers per thread to avoid access to shared memory. This may mean that the number of threads has to be smaller than suggested here (e.g., 64).
3. Threads per block should be a multiple of warp size to avoid wasting computing resources on underpopulated warps. An initial choice of between 128 and 256 threads is a good one for experimentation.
4. The grid should be big enough to provide multiple (e.g., three or four) blocks per SM.

5. Use big enough blocks to take advantage of multiple warp schedulers. Devices of Compute Capability 2.0 or newer, can run multiple warps at the same time, courtesy of having multiple warp schedulers per SM. In Compute Capability 3.0, 3.5 and 5.0 devices, we have four schedulers, which means we should have at least $4 \cdot \text{warpSize}$ -sized blocks.

- The number of threads per block can be derived from the warpSize, the maxThreadsPerBlock, and the register and shared memory demands per thread of the kernel, given the sharedMemPerBlock and regsPerBlock fields of the cudaDeviceProp structure.

The following formula that incorporates this list of guidelines can be used to calculate an initial estimate for the the number of threads per block:

$$threadsPerBlock = \min \left(\begin{array}{l} numWarpSchedulers \cdot warpSize, \\ \frac{regsPerBlock}{registersPerThread}, \\ \frac{sharedMem}{sharedPerThread}, \\ maxThreadsPerSM \end{array} \right) \quad (6.3)$$

where the first line corresponds to our above suggestion of $numWarpSchedulers \cdot warpSize$ per block, the second line considers register restrictions, and the third line incorporates the limit imposed by shared memory consumption. The fourth line is there for completeness, as the first line should never exceed the target device's hardware limitation on the block size.

We are assuming that these are given or supplied:

- `numberOfThreads`: The total number of threads that need to be executed.
- `sharedPerThread`: The total amount of shared memory needed by each thread
(in bytes).
- `registersPerThread`: The total number of registers per thread needed.

The threadsPerBlock can be turned into a multiple of the warpSize via this simple calculation:

$$\text{threadsPerBlock} = \text{warpSize} \cdot \left\lceil \frac{\text{threadsPerBlock}}{\text{warpSize}} \right\rceil \quad (6.4)$$

Given the threadsPerBlock, we can then calculate the size of the grid:

$$\text{totalBlocks} = \left\lceil \frac{\text{numberOfThreads}}{\text{threadsPerBlock}} \right\rceil \quad (6.5)$$

KERNEL STRUCTURE

- The structure of a kernel influences how efficiently an SM is utilized.
- A branching operation leads to the stalling of the threads that do not follow the particular branch,
- the work done by each thread depends explicitly on its ID:

```
__global__ void foo()
{
    int ID = threadIdx.y * blockDim.x + threadIdx.x;
    if( ID % 2 == 0)
    {
        doSmt( ID );
    }

    else
    {
        doSmtElse( ID );
    }
    doFinal( ID );
}
```

LISTING 6.16

An example of a kernel that causes half the threads in a warp to stall.

- A way around the stalling problem would be to modify the condition so that all the threads in a warp follow the same execution path, but they diversify across warps or blocks.
- An alternative ID can be calculated for each thread (called IDprime in the code samples that follow) that alternates between being universally even or universally odd for each warp.
- The new ID can be calculated from the formula:

$$ID' = (ID - warpSize \cdot \lceil \frac{warpID}{2} \rceil) \cdot 2 + (warpID \% 2) \quad (6.7)$$

where ID is the original thread ID and warpID enumerates the warps using this formula: $\text{warpID} = \lfloor \text{ID} / \text{warpSize} \rfloor$. Equation 6.7 can be broken down and explained as

follows:

- The last term, $(\text{warpID} \% 2)$, forces all the threads in a warp to alternate between even or odd ID's, matching the even or odd property of the warpID.
- The multiplier of the first term $(\cdot 2)$ makes the threads form groups whose ID's are different by 2 (effectively all the odd or all the even ones).
- The offset in the first term $(-\text{warpSize} \cdot \lfloor \text{warpID} \rfloor)$ of the equation adjusts the 2

beginning of each new even (odd) warp, so that it is warpSize-distant from the previous even (odd) warp.

```

1 // File: warpFix.cu
2 . . .
3 __global__ void foo ()
4 {
5     int ID = threadIdx.y * blockDim.x + threadIdx.x;
6     int warpID = ID >> offPow;
7     int IDprime = ((ID - (((warpID + 1) >> 1) << offPow )) << 1) + (↵
        warpID & 1);
8
9     if( (warpID & 1) ==0 ) // modulo-2 condition
10    {
11        doSmt(IDprime);
12    }
13    else
14    {
15        doSmtElse(IDprime);
16    }
17    doFinal(IDprime);
18 }

```

LISTING 6.17

An example of a kernel that would not cause intra-warp branching while doing the same

- The code in [Listing 6.17](#) shows the complete modifications that result in a non- stalling kernel despite the use of the if-else blocks

- What if the kernel involves a multiway decision? It is still possible to arrange threads in warps that follow the same path. Similar restrictions apply, i.e., if we have an N-way decision control structure (e.g., a switch statement with N outcomes), we can use the following ID calculation for blocks that are size-multiples of $N \cdot \text{warpSize}$:
- ...
- `__global__ void foo ()`
- $ID' = \text{grpOff} + (ID - \text{grpOff} - (\text{warpID} \% N) \cdot \text{warpSize}) \cdot N + (\text{warpID} \% N)$ where:
- $\text{grpOff} = \lfloor \text{warpID} \rfloor \cdot \text{warpSize} \cdot N$

PAGE-LOCKED /Pinned Memory

- The term page-locked or pinned memory refers to host memory that cannot be swapped out as part of the regular virtual memory operations employed by most contemporary operating systems.
- Pinned memory is **used to hold critical code and data** that cannot be moved out of the main memory, such as the OS kernel.
- It is also needed for performing Direct Memory Access (DMA) transfers across the PCIe bus.
- When one uses regular memory for holding the host data, upon a request to transfer the data to the device, the Nvidia driver allocates paged-locked memory, copies the data to it, and, upon the completion of the transfer, frees up the pinned memory.
- This buffering overhead can be eliminated by using pinned memory for all data that are to be moved between the host and the device.

Pinned memory can be allocated with:

- malloc(), followed by a call to mlock().

Deallocation is done in the reverse order, i.e., calling munlock(), then free().

- Or by calling the **cudaMallocHost()** function. Memory allocated in this fashion has to be deallocated with a call to **cudaFreeHost()**. Otherwise, the program may behave in an unpredictable manner:

```
cudaError_t cudaMallocHost(  
void ** ptr, // Addr . of pointer to pinned  
// memory (IN/OUT)  
size_t size) ; // Size in bytes of request (IN)  
cudaError_t cudaFreeHost ( void * ptr ) ;
```


- The only **potential problem** with pinned memory use is that if it is used in excess, it could lead to performance degradation due to the inability of the host to use virtual memory effectively.
- The performance gain that page-locked memory can yield depends on the **hardware platform** and most importantly, on **the size of the data** involved. There is virtually no gain for transfers below 32 KB, and the bus becomes saturated (i.e., reaches maximum throughput) when transfers go beyond 2 MB.
- For the plateaus Cook got beyond the 2 MB mark, the performance gain ranged between 10% and a massive 2.5x.

ZERO-COPY MEMORY

- Zero-copy memory is a term used to convey that **no explicit memory transfer between the host and the device** needs to be initiated.
- Another, less fashionable term used for the same concept is **mapped memory**.
- Mapped memory is page-locked memory that can be mapped to the address space of the device.
- So, we have a memory region with two addresses: one for access from the host and one for access from the device.
- A transfer across the PCIe bus will be initiated by the CUDA run-time upon the first attempt to access a region of memory that is designated as mapped memory, stalling the active kernel while it is taking place.

This process may sound inefficient, but there is still justification for using mapped memory:

- It makes the program logic simpler because there is no need to separately allocate device memory and transfer the data from the host. This can be a viable option for early development phases that involve porting CPU code to CUDA. Devoting lengthy parts of the code for memory transfers may be a distraction that can be reserved for the later stages, when the core logic of the program behaves as expected.
- The CUDA run-time can automatically overlap kernel-originating memory transfers with another kernel execution. This can boost performance without the need for using streams.
- For low-end systems where the CPU and the GPU share the same physical RAM, no transfer ever takes place, making the use of mapped memory in such cases a no-brainer.

To set up a mapped memory region, we must call the **cudaHostAlloc()** function:

```
cudaError_t cudaHostAlloc( void ** pHost,
```

```
size_t size,
```

```
unsigned int flags) ;
```

```
// Addr . of pointer to mapped
```

```
// memory (IN/OUT)
```

```
// Size in bytes of request (IN) // Options for function (IN)
```

using the symbolic constant `cudaHostAllocMapped` for the `flags` parameter. Because mapped memory is also page-locked memory, its release is done with the `cudaFreeHost()` function.

- Mapped memory is referenced by two pointers: one on the host and one on the device.
- The device pointer, which is the one to be passed to a kernel, can be retrieved from the host pointer with the `cudaHostGetDevicePointer` function:

```

cudaError_t cudaHostGetDevicePointer(
    void ** pDevice,    // Address where the returned device
                        // pointer is stored (IN/OUT)
    void * pHost,       // Address of host pointer (IN)
    unsigned int flags) // Currently should be set to 0

```

So, we can have:

```

int *h_data, *d_data;
cudaHostAlloc((void **)&h_data, sizeof(int)* DATASIZE, ←
    cudaHostAllocMapped);
cudaHostGetDevicePointer((void **)&d_data, (void *)h_data, 0);
doSmt<<< gridDim, blkDim >>>(d_data);

```

UNIFIED MEMORY



- The separate memory spaces of host and device necessitate the explicit (or implicit via zero-copy memory) transfer of data between them so that a GPU can process the designated input and return the outcome of a computation.
- This results in a sequence of `cudaMemcpy()` operations, as shown in the examples of the previous sections.
- **Unified Memory is a facility introduced in CUDA 6.0 that allows implicit transfers to take place both to and from the device, without the need for lengthy and error-prone `cudaMemcpy()` sequences.**

- Unified Memory introduces the concept of [managed memory](#), which is essentially memory allocated on both host and device under the control of the device driver.
- The device driver ensures that the two memory ranges stay coherent, i.e., contain the same data, when they are being accessed by either CPU or GPU.
- The Unified Memory term is justified by the fact that a program needs to maintain just a single pointer to the data, which is similar to the zero-copy memory described in [Section 6.7.5](#).
- The difference is that for zero-copy memory, the transfer is triggered by access, i.e., during kernel execution, whereas in the Unified Memory case the transfer is initiated immediately before the launch and promptly after the termination of a kernel.

- Unified Memory does not reduce the execution time of a program, since the transfers take place as in a typical CUDA program, albeit implicitly. In that regard, we can consider it an optimization technique only as far as the program structure is concerned.
- Managed memory can be allocated in two ways:
 - Dynamically, via a call to the `cudaMallocManaged()` function, which is just a variant of `cudaMalloc()`.
 - Statically, by declaring a global variable as being `__managed__`.
- In both cases, the resulting pointer/variable can be also accessed from the host, indicating the second host-side allocation. The key, however, is when this access is possible. The managed memory is “handed over” to the GPU for the duration of a kernel’s execution. The host side of the memory is inaccessible while the kernel is being executed, actually generating a protection fault if access is attempted.[19](#)

The syntax of `cudaMallocManaged()` is as follows:

```
template < class T > cudaError_t cudaMallocManaged (
    T **devPtr,    // Address for storing the memory pointer
                  // (IN/OUT)
    size_t size,   // Size in bytes of the required memory (IN)
    unsigned flags)// Creation flag, defaults to
                  // cudaMemAttachGlobal (IN)
```

- Improve what you can measure, need to identify:
-  hotspots: which function takes most of the run item.
-  bottlenecks: what limits the performance of the hotspots. Manual timing/debugging is difficult
- CUDA tools: cuda-memcheck to detect invalid memory access cuda-gdb, nsight eclipse to debug a CUDA program. nvprof, visual profiler to profile CUDA programs occupancy calculator

DEBUGGING CUDA PROGRAMS

Nvidia provides two tools for debugging CUDA applications:

- The Parallel NSight Eclipse-based IDE (nsight) provides integrated GUI-based debugging for Linux and Mac OS X platforms. Under Windows, NSight integrates with Visual Studio.
- CUDA-GDB (cuda-gdb) is a command-line debugger based on GNU's debugger (gdb) and it is available for Linux and Mac OS X platforms.

- Debugging CUDA programs suffers from a major drawback that stems from the peculiarity of using a display device for computations: It requires two GPUs, one for running the application under development and one for regular display.
- The former GPU may be hosted in the same machine or a remote one. Most users would find themselves under the second scenario, i.e., using a remote machine (e.g., a shared server) for debugging purposes.
- Those unfortunate enough to not have access to a second GPU will have to rely on `printf()` for probing the state of CUDA threads during execution.

- A remote debugging session has to be initially configured by selecting the Run → Debug Remote Application... menu option. In response, the dialog box in [Figure 6.20](#) will appear.
- The remote debugging is conducted by having the application run under the supervision of a cuda-gdbserver process. Unless the remote system is already running cuda-gdbserver, the first option should be selected in [Figure 6.20](#).
- The next dialog box, which is shown in [Figure 6.21](#), controls how the remote system will get the application.
- Unless a common filesystem, e.g., an NFS volume, is being used, this has to be done by uploading the new binary.
- If the remote system is being used for the first time, pressing “Manage” allows the user to specify the connection details.

- The connection details of all the remote systems used for debugging, are collectively managed by the dialog shown in [Figure 6.22](#).
- A new debugging session can commence by selecting the just created configuration from the “Run → Debug Configurations...” dialog window (see [Figure 6.22](#)).
- Caution: It is a good idea to use the same CUDA SDK in both the local and remote hosts in order to avoid errors caused by missing libraries.

PROFILING CUDA PROGRAMS

- In the case of a sequential program, profiling can help us pinpoint the parts of the program that contribute the most to the execution time.
- In the case of a GPU program, profiling can help us understand the factors limiting or preventing the GPU from achieving its peak performance.

Nvidia provides profiling functionality via two tools:

- **nvprof** is a **command-line-based** utility for profiling CUDA programs. A number of switches enable control over the profiling process (e.g., what kind of metrics to collect). It is the tool of choice for profiling remote systems. In contrast with the debugging process, there is no mandate for using two GPUs, so profiling a remote system is just a choice.
- **Nvidia Visual Profiler (nvvp)** is a GUI-based tool for visualizing the execution timeline of a CUDA program. nvvp can provide a plethora of analyses, guiding the development of a CUDA program. nvvp can be also used for the visualization of data collected from nvprof.

- The Visual Profiler can be used as a standalone tool or embedded within Parallel NSight. As a part of NSight, it can be used from within the IDE by switching to the Profile view.
- The starting point in profiling a program is to decide which are the parts to be examined.
- Profiling is a process that is characterized by the following:
 - 1. The potential generation of huge amounts of visual data that increase monotonically as a function of the execution time. Extracting useful information from long runs can be very challenging. It is a better idea to focus on specific parts of the program.

- 2. The profiler can influence the execution of the program by altering the timing of operations and the amount of I/O generated.
- 3. The profiler's results can be influenced by irregular operations timing. Because the profiler may need to run the program several times in order to collect reliable statistics, it is desirable, especially in the case of multi-threaded host programs that use different contexts/streams, to have a single CPU thread generate all the operations in the same sequence every time.

- These problems can be minimized or eliminated by performing focused instead of whole-program profiling.
- In focused profiling, only parts of the program that are executed between calls to the `cudaProfilerStart()` and `cudaProfilerStop()` functions will be profiled, i.e., profiling is controlled programmatically from within the profiled program. The skeleton of a focused profiled program would be similar to:

- `#include <cuda_profiler_api.h> /* necessary include file */ ...`
- `cudaProfilerStart () ;`
- `// calls to CPU and GPU routines`
- `cudaProfilerStop () ; ...`
- Firing up a program in NSight and getting the profiler's report is pretty simple `Run → Profile As... → Local C/C++ Application` is all you need to select from the menu (this option is visible in the C/C++ view of the IDE). Only a handful of options need to be specified, as shown in the Profile Configurations dialog window of [Figure 6.23](#). These are:

- **Start execution with profiling enabled:** If this option is selected, the profiler will collect data from the start till the end of execution. Otherwise, focused profiling is in effect.
- **Enable concurrent kernel profiling:** Should be selected only if multiple streams are used in the application.
- **Enable power, clock, and thermal profiling:** Enables the collection of these additional metrics. The collection also depends on whether the target GPU can supply this information.
- **Run guided analysis:** If selected, the profiler will run an analysis phase where a set of problem areas are identified in the program, and solutions are proposed. This option is not available from within NSight, but it is shown in Visual Profiler whenever a new session is created.