# Exercise Solutions

All performance results included below are only examples of possible observations. Depending on your platform, your results may and likely will vary. These results are only included as examples of the type of measurements you should perform when answering these questions, and not as the actual values you should expect to see.

The example commands provided are for binaries included with CUDA 6. The commands for other versions of CUDA will likely vary, though the underlying concepts are the same.

Additionally, many of these questions can be answered in multiple ways. If your solution does not perfectly match with the provided example, but does implement the same functionality and demonstrate the same performance characteristics that would be considered an equivalent solution.

---

# Chapter 1

1 *Refer to Figure 1-5 and finish the following patterns of data partition:*

- *Block partition along the* `x` *dimension for 2D data*
- *Cyclic partition along the* `y` *dimension for 2D data*
- *Cyclic partition along the* `z` *dimension for 3D data*


2 *Remove the* `cudaDeviceReset` *function from the file, then compile and run it to see what would happen.*

When `cudaDeviceReset` is removed, none of the prints from the GPU are displayed:

```
$ ./hello
Hello World from CPU!
$
```

While `printf` is still called on the GPU, `cudaDeviceReset` forces those prints to be flushed from the GPU, to the host, and then output in the user-visible console. Without calling `cudaDeviceReset` (or other functions that force flushing of GPU output), there are no guarantees that these prints will be displayed. Chapter 10 will provide more details on the use and behavior of CUDA's `printf`.

3 *Replace the function `cudaDeviceReset` with `cudaDeviceSynchronize`, then compile and run it to see what happens.*

When `cudaDeviceReset` is replaced with `cudaDeviceSynchronize`, the prints from the GPU are displayed on the console:

```
$ ./hello
Hello World from CPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
$
```

`cudaDeviceSynchronize` is another function (in addition to `cudaDeviceReset`) that can be used to force GPU prints to be flushed to the user-visible console.

4 *Refer to the section "Hello World from GPU." Remove the switch of device architecture from the compiler command line and compile it as follows to see what happens.*

```
$ nvcc hello.cu -o hello
```

Removing the device architecture switch (`-arch=sm_20`) produces the following output, and fails to produce an executable:

```
$ nvcc hello.cu -o hello
nvcc warning : The 'compute_10' and 'sm_10' architectures are deprecated, and may be
removed in a future release.
hello.cu(11): error: calling a __host__ function("printf") from a __global__
function("helloFromGPU") is not allowed

1 error detected in the compilation of "tmpxft_00005521_00000000-6_hello.cpp1.ii".
```

When no architecture flag is specified, `nvcc` will default to using compute capability 1.0. However, calling `printf` from the GPU is not supported by compute capability 1.0. The compiler therefore prints an error and exits before the

executable is generated. This exercise illustrates the different capabilities that progressively higher compute capabilities can add to a CUDA application.

5 *Refer to the CUDA online document (`http://docs.nvidia.com/cuda/index.html`). Based on the section "CUDA Compiler Driver NVCC," what file suffixes does nvcc support compilation on?*

`nvcc` supports the following input file suffixes: `.cu`, `.cup`, `.c`, `.cc`, `.cxx`, `.cpp`, `.gpu`, `.ptx`, `.o`, `.obj`, `.a`, `.lib`, `.res`, and `.so`. It can be used as a general-purpose compiler to build more than just CUDA source code.

6 *Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through the built-in* `threadIdx.x` *variable. Modify the kernel function using the thread index to let the output be:*

```
$ ./hello
Hello World from CPU!
Hello World from GPU thread 5!
```

For this solution, use a conditional to ensure that only thread 5 executes the `printf`, and add the thread ID to the print statement:

```
#include <stdio.h>

__global__ void helloFromGPU(void)
{
    if (threadIdx.x == 5)
    {
        printf("Hello World from GPU thread %d!\n", threadIdx.x);
    }
}

int main(void)
{
       printf("Hello World from CPU!\n");

       helloFromGPU<<<1, 10>>>();
       cudaDeviceReset();
       return 0;
}
```

# Chapter 2

1 *Using the program of `sumArraysOnGPU-timer.cu`, set the `block.x = 1023`. Recompile and run it. Compare the result with the execution configuration of `block.x = 1024`. Try to explain and reason about the difference.*

With `block.x` set to 1023, `sumArraysOnGPU-timer` produces the following output on an M2050:

```
$ ./sumArraysOnGPU-timer
./sumArraysOnGPU-timer Starting...
Using Device 0: Tesla M2050
Vector size 16777216
initialData Time elapsed 0.356134 sec
sumArraysOnHost Time elapsed 0.023177 sec
sumArraysOnGPU <<<  16401, 1023  >>>  Time elapsed 0.003164 sec
Arrays match.
```

With `block.x` set to 1024, the following output is produced:

```
$ ./sumArraysOnGPU-timer
./sumArraysOnGPU-timer Starting...
Using Device 0: Tesla M2050
Vector size 16777216
initialData Time elapsed 0.356447 sec
sumArraysOnHost Time elapsed 0.023294 sec
sumArraysOnGPU <<<  16384, 1024  >>>  Time elapsed 0.002416 sec
Arrays match.
```

Note the performance difference between the kernels with different block configurations. With `block.x` set to 1024 the kernel runs 1.31 times faster. Because threads are scheduled in groups of 32, running blocks with 1023 threads leads to two suboptimal application characteristics. First, the last warp in every thread block will have a disabled thread that performs no work because 1024 is not evenly divisible into warps. Second, because there are fewer threads per block the application will require more blocks to fully process the input. Running more thread blocks causes longer execution times as only a fixed number of blocks can run concurrently.

2 *Refer to `sumArraysOnGPU-timer.cu`, and let `block.x = 256`. Make a new kernel to let each thread handle two elements. Compare the results with other execution configurations.*

One way to implement the `sumArraysOnGPU` kernel such that each thread handles two elements is below. Note that this is not the only way to correctly implement this behavior.

```
__global__ void sumArraysOnGPU(float *A, float *B, float *C, const int N)
{
    int nthreads = gridDim.x * blockDim.x;
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = i + nthreads;

    if (i < N) C[i] = A[i] + B[i];
    if (j < N) C[j] = A[j] + B[j];
}
```

Changing the behavior of the kernel requires changing the execution configuration of the kernel as well. Namely, half as many threads are required to process the full input. Therefore, setting the grid size should be changed from:

```
dim3 grid  ((nElem + block.x - 1) / block.x);
```

to:

```
dim3 grid  (((nElem / 2) + block.x - 1) / block.x);
```

With these modifications, executing `sumArraysOnGPU` with a variety of block configurations produces the performance results in Table 1. Note that these measurements are slightly faster than the performance measured in Exercise 2. This is a result of improved global memory bandwidth utilization, a topic that will be discussed further in Chapter 4.

**Table 1: Performance Results of Modified sumArraysOnGPU**

| Thread Block Size | Kernel Time |
|---|---|
| 256 | 0.002034 s |
| 512 | 0.002037 s |
| 768 | 0.002064 s |
| 1024 | 0.002196 s |

3 *Refer to `sumMatrixOnGPU-2D-grid-2D-block.cu`. Adapt it to integer matrix addition. Find the best execution configuration.*

An implementation of `sumMatrixOnGPU-2D-grid-2D-block` for integer matrix addition can be found in `sumMatrixOnGPU-2D-grid-2D-block-integer.cu`. Both the host and CUDA implementations of matrix sum are modified to take integers as input, and the validation and input generation code on the host are also changed to work with integer data types.

An example of testing different execution configurations is shown in Table 2 with the best execution configuration highlighted.

**Table 2: Performance Results of Modified sumMatrixOnGPU-2D-grid-2D-block-integer**

| block.x | block.y | Kernel Time |
|---|---|---|
| 8 | 1 | 0.072383 s |
| 8 | 2 | 0.039848 s |
| 8 | 4 | 0.024574 s |
| 8 | 8 | 0.016544 s |
| 8 | 16 | 0.013650 s |
| 8 | 32 | 0.013071 s |
| 8 | 64 | 0.014778 s |
| 8 | 128 | 0.020552 s |
| 16 | 1 | 0.036748 s |
| 16 | 2 | 0.021407 s |
| 16 | 4 | 0.013488 s |
| 16 | 8 | 0.010118 s |

| | | |
|---|---|---|
| 16 | 16 | 0.010298 s |
| 16 | 32 | 0.010512 s |
| 16 | 64 | 0.014894 s |
| 32 | 1 | 0.018809 s |
| 32 | 2 | 0.011414 s |
| 32 | 4 | 0.008204 s |
| 32 | 8 | 0.008154 s |
| 32 | 16 | 0.008840 s |
| 32 | 32 | 0.011436 s |
| 64 | 1 | 0.011015 s |
| 64 | 2 | 0.007912 s |
| 64 | 4 | 0.007655 s |
| 64 | 8 | 0.007982 s |
| 64 | 16 | 0.011314 s |
| 128 | 1 | 0.007864 s |
| **128** | **2** | **0.007264 s** |
| 128 | 4 | 0.007537 s |
| 128 | 8 | 0.009351 s |
| 256 | 1 | 0.007262 s |
| 256 | 2 | 0.007471 s |
| 256 | 4 | 0.011192 s |
| 512 | 1 | 0.007357 s |
| 512 | 2 | 0.009597 s |

4 *Refer to* `sumMatrixOnGPU-2D-grid-1D-block.cu`*. Make a new kernel to let each thread handle two elements. Find the best execution configuration.*

An implementation of `sumMatrixOnGPU-2D-grid-1D-block` that supports processing two elements per thread can be found in `sumMatrixOnGPU-2D-grid-1D-block-two.cu`. The device kernel is modified to process two elements in each thread, and the execution configuration is modified to launch half as many threads.

An example of testing different execution configurations is shown in Table 3 with the best execution configuration highlighted.

**Table 3: Performance Results of Modified sumMatrixOnGPU-2D-grid-1D-block-two**

| block.x | Kernel Time |
|---|---|
| 1 | 0.432855 s |
| 2 | 0.216920 s |
| 4 | 0.119819 s |

| | |
|---|---|
| 8 | 0.060605 s |
| 16 | 0.031227 s |
| 32 | 0.016533 s |
| 64 | 0.010049 s |
| 128 | 0.007589 s |
| **256** | **0.007461 s** |
| 512 | 0.007469 s |
| 1024 | 0.008510 s |

5 *Using `checkDeviceInfor.cu`, find the maximum sizes of each dimension of grid and block of your system.*

`checkDeviceInfor` prints a variety of information on the current system, including the information below on grid and block dimension limits. Note that this is only an example output, and that the actual values will vary between systems.

```
Maximum sizes of each dimension of a block:    1024 x 1024 x 64
Maximum sizes of each dimension of a grid:     65535 x 65535 x 65535
```

# Chapter 3

1 *What are the two primary causes of performance improvement when unrolling loops, data blocks, or warps in CUDA? Explain how each type of unrolling improves instruction throughput.*

Unrolling loops, data blocks, or warps can lead to less frequent branching from fewer loop conditionals. Additionally, unrolling can lead to an increase in the number of independent memory operations discoverable by the compiler. As a result, more concurrent read and write operations can be issued and memory bandwidth utilization will increase. The topic of memory bandwidth utilization will be covered in greater detail in Chapter 4.

2 *Refer to the kernel `reduceUnrolling8` and implement the kernel `reduceUnrolling16`, in which each thread handles 16 data blocks. Compare kernel performance with `reduceUnrolling8` and use the proper metrics and events with `nvprof` to explain any difference.*

`reduceInteger-16.cu` adds a sample implementation of `reduceUnrolling16` as well as added host code to call it. Running `reduceInteger-16` produces the following results:

```
$ ./reduceInteger-16
./reduceInteger-16 starting reduction at device 0: Tesla M2050     with array size
16777216  grid 32768 block 512
cpu reduce      elapsed 0.022204 sec cpu_sum: 2139353471
```

```
gpu Neighbored  elapsed 0.011671 sec gpu_sum: 2139353471 <<<grid 32768 block 512>>>
gpu Neighbored2 elapsed 0.009792 sec gpu_sum: 2139353471 <<<grid 32768 block 512>>>
gpu Interleaved elapsed 0.006355 sec gpu_sum: 2139353471 <<<grid 32768 block 512>>>
gpu Unrolling2  elapsed 0.003418 sec gpu_sum: 2139353471 <<<grid 16384 block 512>>>
gpu Unrolling4  elapsed 0.001816 sec gpu_sum: 2139353471 <<<grid 8192 block 512>>>
gpu Unrolling8  elapsed 0.001378 sec gpu_sum: 2139353471 <<<grid 4096 block 512>>>
gpu Unrolling16 elapsed 0.000901 sec gpu_sum: 2139353471 <<<grid 2048 block 512>>>
gpu UnrollWarp8 elapsed 0.001343 sec gpu_sum: 2139353471 <<<grid 4096 block 512>>>
gpu Cmptnroll8  elapsed 0.001279 sec gpu_sum: 2139353471 <<<grid 4096 block 512>>>
gpu Cmptnroll   elapsed 0.001245 sec gpu_sum: 2139353471 <<<grid 4096 block 512>>>
```

`reduceUnrolling16` demonstrates a 1.53 times performance improvement relative to `reduceUnrolling8`. Using the `dram_read_throughput` metric shows that `reduceUnrolling16` nearly doubles global memory read throughput relative to `reduceUnrolling8` on an M2050 GPU (shown below). Results may vary with your hardware platform, but because unrolling leads to more concurrent read operations you should expect to see some improvement on other platforms as well.

```
      Kernel: reduceUnrolling8(int*, int*, unsigned int)
         1                    dram_read_throughput             Device Memory Read
Throughput  62.110GB/s  62.110GB/s  62.110GB/s
      Kernel: reduceUnrolling16(int*, int*, unsigned int)
         1                    dram_read_throughput             Device Memory Read
Throughput  102.61GB/s  102.61GB/s  102.61GB/s
```

3 *Refer to the kernel `reduceUnrolling8` and replace the following code segment:*

```
int a1 = g_idata[idx];
int a2 = g_idata[idx+blockDim.x];
int a3 = g_idata[idx+2*blockDim.x];
int a4 = g_idata[idx+3*blockDim.x];
int b1 = g_idata[idx+4*blockDim.x];
int b2 = g_idata[idx+5*blockDim.x];
int b3 = g_idata[idx+6*blockDim.x];
int b4 = g_idata[idx+7*blockDim.x];
g_idata[idx] = a1+a2+a3+a4+b1+b2+b3+b4;
```

*with the functionally equivalent code below:*

```
int *ptr = g_idata + idx;
int tmp = 0;

// Increment tmp 8 times with values strided by blockDim.x
for (int i = 0; i < 8; i++) {
    tmp += *ptr;
    ptr += blockDim.x
}
g_idata[idx] = tmp;
```

*Compare the performance of each and explain the difference using `nvprof` metrics and register usage.*

`reduceUnrolling8` implements data block unrolling by a factor of eight. This transformation retains that data block unrolling, but uses an eight-iteration loop to implement it rather than inlining it as eight separate statements.

An example implementation is provided in `reduceInteger-8-new.cu`. Depending on the version of the CUDA Toolkit you have installed, you may see the original or new version of the `reduceUnrollng8` kernel faster. However, no matter which version is faster the cause of that performance improvement will be the same.

To start, run each version of the kernel to measure overall performance. Testing with CUDA 4, the original kernel averages 1.089 ms execution time while the new kernel averages 1.419 ms. With CUDA 6, the original kernel averages 1.399 ms and the new kernel averages 1.083 ms. Therefore, the relative performance of these kernels on different CUDA versions may be inverted depending on which version you are using.

The first metric to consider is the number of registers used by each kernel with each CUDA version. Table 4 shows that for each version, the higher performing kernel uses fewer registers. The number of registers can be determined from the output of:

```
$ nvcc -O2 -arch=sm_20 -Xptxas="-v" reduceInteger-8-new.cu
```

**Table 4: Register Usage of `reduceUnrolling8` kernels**

| | CUDA 4 # registers | CUDA 6 # registers |
|---|---|---|
| **Original Kernel** | 20 | 22 |
| **New Kernel** | 21 | 20 |

Because the registers in an SM are a shared resource among all threads in that SM, the number of registers used by a kernel limits the number of concurrent warps that can be scheduled. Using the nvprof `achieved_occupancy` metric, you can verify that fewer threads are being concurrently scheduled for kernels with higher register usage. Example values of `achieved_occupancy` for both kernels using CUDA 4 and CUDA 6 are shown in Table 5.

**Table 5: `achieved_occupancy` of `reduceUnrolling8` kernels**

| | CUDA 4 `achieved_occupancy` | CUDA 6 `achieved_occupancy` |
|---|---|---|
| **Original Kernel** | 0.957 | 0.635 |
| **New Kernel** | 0.633 | 0.957 |

With more concurrent threads scheduled, a greater number of memory instructions will be concurrently scheduled and bus utilization will increase. This can be verified using the `gld_transactions`, `gst_transactions`, `gld_efficiency`, `gst_efficiency`, `gld_throughput`, and `gst_throughput` metrics. If `gld_transactions` and `gst_transactions` are similar for both kernels, that verifies that both are doing similar amounts of I/O. If `gld_efficiency` and `gst_efficiency` are also similar, then those transactions are being used equally well in both

kernels. Then, if `gld_throughput` and `gst_throughput` show significant improvement for the faster kernels then the bus is being better utilized and likely the cause for improved performance. Tables 6, 7, and 8 show example measurements for all metrics on both kernels and both CUDA 4 and 6. The measurements below show better bus utilization for the faster kernels.

**Table 6: `gld_transactions`, `gst_transactions` of `reduceUnrolling8` kernels**

| | CUDA 4 `gld_transactions` | CUDA 4 `gst_transactions` | CUDA 6 `gld_transactions` | CUDA 6 `gst_transactions` |
|---|---|---|---|---|
| **Original Kernel** | 694,420 | 152,002 | 693,868 | 151,976 |
| **New Kernel** | 696,412 | 152,313 | 694,124 | 151,955 |

**Table 7: `gld_efficiency`, `gst_efficiency` of `reduceUnrolling8` kernels**

| | CUDA 4 `gld_efficiency` | CUDA 4 `gst_efficiency` | CUDA 6 `gld_efficiency` | CUDA 6 `gst_efficiency` |
|---|---|---|---|---|
| **Original Kernel** | 94.38% | 97.71% | 94.38% | 97.71% |
| **New Kernel** | 94.30% | 97.71% | 94.40% | 97.71% |

**Table 8: `gld_throughput`, `gst_throughput` of `reduceUnrolling8` kernels**

| | CUDA 4 `gld_throughput` | CUDA 4 `gst_throughput` | CUDA 6 `gld_throughput` | CUDA 6 `gst_throughput` |
|---|---|---|---|---|
| **Original Kernel** | 75.686 GB/s | 14.637 GB/s | 57.067 GB/s | 11.022 GB/s |
| **New Kernel** | 57.344 GB/s | 11.068 GB/s | 76.909 GB/s | 14.896 GB/s |

4 *Refer to the kernel `reduceCompleteUnrollWarps8`. Instead of declaring `vmem` as `volatile`, use `__syncthreads`. Note that `__syncthreads` must be called by all threads in a block. Compare the performance of the two kernels. Use `nvprof` to explain any differences.*

The most important thing to recognize for this exercise is that declaring `vmem` as `volatile` affects both the loads and stores performed using that pointer, while `__syncthreads` simply ensures that all threads in a block are aware of any changes to global memory by other threads in the same block.

As a result of making `vmem` `volatile`, all loads performed with that pointer will skip the L1 cache on Fermi devices and directly access global memory. When `__syncthreads` is used instead, loads can still hit in the L1 cache on Fermi devices since it is shared by a thread block. On Kepler devices, the performance improvement from `__syncthreads` will be smaller, but still there as a result of loads still hitting in L2.

To verify this on a Fermi device, you can use two techniques. First, the `l1_cache_global_hit_rate` metric checks how well L1 cache is being

used for L1 loads. Measuring this and comparing the two kernels allows you to verify that cache behavior in the two is different.

Then, you can also replace all loads in the `__syncthreads` version to instead use a `volatile` pointer and verify that performance then becomes similar to the original version.

An example implementation is provided in `reduceInteger-sync.cu`. The kernel `reduceCompleteUnrollWarps8Sync` provides an implementation that uses `__syncthreads` instead of a `volatile` pointer. The kernel `reduceCompleteUnrollWarps8SyncVolatile` uses `__syncthreads` and a `volatile` pointer for loads to verify that cached loads are causing the performance discrepancy. Measurements of overall execution time and L1 cache hit rates on a Fermi GPU for the original kernel and the two kernels are included below in Table 9. These measurements demonstrate that using `__syncthreads` produces better performance as a result of using the L1 cache.

**Table 9: Execution time and L1 hit rates for `reduceInteger-sync`**

| | Execution Time | `l1_cache_global_hit_rate` |
|---|---|---|
| **Original** | 1.289 ms | 0.00% |
| **Using** `__syncthreads` | 1.170 ms | 5.65% |
| **Using** `__syncthreads` **and** `volatile` | 1.291 ms | 0.00% |

5 *Implement sum reduction of* `floats` *in C.*

An example implementation of floating-point sum reduction is available in `reduceFloat.c`. The types of parameters and variables were updated to be `float` instead of `int`, the input data initialization logic was changed to produce random floating-point values rather than random integers, and memory allocation was changed to allocate sufficient bytes for floating-point data rather than integer data (though the actual number of bytes may not have changed).

6 *Refer to the kernel* `reduceInterleaved` *and the kernel* `reduceCompleteUnrollWarps8` *and implement a version of each for* `floats`*. Compare their performance and choose proper metrics and/or events to explain any differences. Are there any differences compared to operating on integer data types?*

Example implementations of `reduceInterleaved` and `reduceCompleteUnrollWarps8` for floating-point numbers are available in `reduceFloatGpu.cu`.

Depending on the architecture, there may or may not be differences in performance. On older GPU architectures, contention for floating-point arithmetic logic units may lead to a loss in performance. However, on many GPUs there will be no difference in performance. As both `int` and `float` values have the same

number of bytes, the amount of I/O performed by this kernel will not change, only the hardware units used to perform arithmetic.

7 *When are the changes to global data made by a dynamically spawned child kernel guaranteed to be visible to its parent?*

When a parent has synchronized on the completion of its child, and continued past that synchronization.

8 *Refer to the file `nestedHelloWorld.cu` and implement a new kernel using the methods illustrated in Figure 3-30.*

An example solution that consolidates the spawning of all child threads is provided in `nestedHelloWorldNew.cu`.

9 *Refer to the file `nestedHelloWorld.cu` and implement a new kernel that can limit nesting levels to a given depth.*

An example solution is provided in `nestedHelloWorldLimited.cu`. `nestedHelloWorldLimited` uses a `maxDepth` parameter passed to the kernel to determine when no further nested kernels should be created.

# Chapter 4

1 *Refer to the file `globalVariable.cu`. Declare statically a global float array with a size of five elements. Initialize the global array with the same value of 3.14. Modify the kernel to let each thread change the value of the array element with the same index as the thread index. Let the value be multiplied with the thread index. Invoke the kernel with five threads.*

An example solution is available in `globalVariable1.cu`. globalVariable1 declares the global array as follows:

```
__device__ float devData[5];
```

and initializes it using `cudaMemcpyToSymbol`. The kernel is launched with five threads and the results of its modifications copied back using `cudaMemcpyFromSymbol`.

2 *Refer to the file `globalVariable.cu`. Replace the following symbol copy functions:*

```
cudaMemcpyToSymbol()
cudaMemcpyFromSymbol()
```

*with the data transfer function*

```
cudaMemcpy()
```

*You will need to acquire the address of the global variable using:*

`cudaGetSymbolAddress()`

An example solution is available in `globalVariable2.cu`. `globalVariable2` starts by acquiring the device memory address of `devData` using `cudaGetSymbolAddress` and storing it in a `float*` on the host. Then `cudaMemcpy` is used to transfer a value from the host to the device and back, just as if the device memory was allocated using `cudaMalloc`.

3 *Compare performance of the pinned and pageable memory copies in `memTransfer` and `pinMemTransfer` using `nvprof` and different sizes: 2M, 4M, 8M, 16M, 32M, 64M, 128M.*

The `memTransfer` and `pinMemTransfer` examples should be used to answer this exercise. A useful first step is to modify each application to take a command line argument that sets the transfer size, though results can also be measured by manually modifying and recompiling the code for each different size. Example performance results on an M2050 GPU are shown below in Table 10. Keep in mind that results on other hardware will vary. These results were obtained by running the commands below with modified versions of `memTransfer.cu` and `pinMemTransfer.cu` (available with the other solutions).

```
$ nvprof ./memTransfer
$ nvprof ./pinMemTransfer
```

**Table 10: Memory Transfer Performance of Pinned and Pageable Memory**

| Buffer Size | Pageable HtoD (µs/MB) | Pageable DtoH (µs/MB) | Pinned HtoD (µs/MB) | Pinned DtoH (µs/MB) |
|---|---|---|---|---|
| 2MB | 278.77 | 163.63 | 175.46 | 158.3165 |
| 4MB | 325.74 | 199.48 | 174.69 | 157.80 |
| 8MB | 338.03 | 213.3 | 174.31 | 157.54 |
| 16MB | 371.70 | 286.32 | 174.13 | 158.70 |
| 32MB | 389.18 | 292.16 | 174.04 | 160.40 |
| 64MB | 348.21 | 265.75 | 173.98 | 160.35 |
| 128MB | 344.93 | 266.81 | 173.95 | 160.33 |

4 *Using the same examples, compare the performance of pinned and pageable memory allocation and deallocation using CPU timers and different sizes: 2M, 4M, 8M, 16M, 32M, 64M, 128M.*

The `memTransfer` and `pinMemTransfer` examples should be used to answer this exercise as well. Just as in the previous question, a good first step is to modify each example to take a command line argument that sets the transfer size, though results can also be measured by manually modifying and recompiling the code for each different size. You will also need to add CPU timers around the host memory allocations and deallocations to measure their performance. This can be

done using the provided `seconds` function, or any time measurement function of your choosing. An example of allocation performance on an M2050 is shown below in Table 11, and deallocation performance in Table 12. Results on other hardware will vary. These results were obtained with the modified versions of `memTransfer.cu` and `pinMemTransfer.cu` included with the other solutions, using the output from the commands below.

```
$ ./memTransfer
$ ./pinMemTransfer
```

**Table 11: Memory Allocation Performance of Pinned and Pageable Memory**

| Buffer Size | Pageable Allocation (µs/MB) | Pinned Allocation (µs/MB) |
|---|---|---|
| 2MB | 1.323 | 22139.104 |
| 4MB | 0.874 | 11299.111 |
| 8MB | 1.674 | 5821.948 |
| 16MB | 0.208 | 3104.741 |
| 32MB | 0.105 | 1739.005 |
| 64MB | 0.052 | 1056.625 |
| 128MB | 0.027 | 714.851 |

**Table 12: Memory Deallocation Performance of Pinned and Pageable Memory**

| Buffer Size | Pageable Deallocation (µs/MB) | Pinned Deallocation (µs/MB) |
|---|---|---|
| 2MB | 90.40 | 233.24 |
| 4MB | 48.99 | 197.94 |
| 8MB | 25.93 | 183.78 |
| 16MB | 14.59 | 174.81 |
| 32MB | 8.21 | 169.58 |
| 64MB | 4.98 | 166.48 |
| 128MB | 3.51 | 163.42 |

5 *Modify `sumArrayZerocopy.cu` to access `A`, `B`, and `C` at an offset. Compare performance with and without L1 cache enabled. If you do not have a Fermi GPU, reason about the expected results on a Fermi.*

An example implementation is provided in `sumArrayZerocopy-offset.cu`. Measuring execution time on a Fermi GPU with offset 11 and L1 cache enabled produces an average of 5.9199 ms, with L1 cache disabled an average of 8.8577 ms. These results demonstrate that the L1 cache is not only used for caching global loads in the Fermi architecture, but also for loads from system memory. By enabling L1 cache, partial re-use of earlier reads leads to improved bus utilization.

6 *Modify `sumArrayZerocopyUVA.cu` to access `A`, `B`, and `C` at an*

*offset. Compare performance with and without L1 cache enabled. If you do not have a Fermi GPU, explain the results you would expect to see with and without L1 cache enabled.*

An example implementation is provided in `sumArrayZerocopyUVA-offset.cu`. Measuring execution time on a Fermi GPU with offset 11 and L1 cache enabled produces an average of 5.9255 ms, with L1 cache disabled an average of 8.8588 ms. These results demonstrate that the L1 cache is used for caching loads from system memory even when UVA is enabled. By enabling L1 cache, partial re-use of earlier reads leads to improved bus utilization.

7 *Compile the file `readSegment.cu` and run the following command on offsets 0, 4, 8, 16, 32, 64, 96, 128, 160, 192, 224, and 256:*

```
./iread $OFFSET
```

*Confirm what byte the aligned address must be a multiple of.*

Example outputs of `readSegment` for all offsets are shown below in Table 13. There is a clear separation between offsets that are multiples of 32 and those that are not. Because the offsets are provided in multiples of `float`, offsets that are multiples of 32 correspond to offsets aligned to 128 bytes. Therefore, the results below confirm that addresses aligned to 128 bytes demonstrate better performance.

**Table 13: readSegment Performance with Varying Memory Access Offsets**

| Float Offset | Kernel Execution Time (µs) |
|---|---|
| 0 | 128.033 |
| 4 | 136.067 |
| 8 | 136.100 |
| 16 | 135.867 |
| 32 | 128.300 |
| 64 | 127.933 |
| 96 | 128.167 |
| 128 | 127.733 |
| 160 | 128.367 |
| 192 | 128.067 |
| 224 | 128.133 |
| 256 | 127.900 |

8 *Refer to the file `Makefile` for `readSegment.cu`. Disable L1 cache in the `Makefile` and generate the executable `iread_l2`. Test it with the following command on offset 0, 11, and 128:*

```
./iread_l2 $OFFSET
```

*Compare the result with L1 cache enabled to see what is different.*

Compiling `readSegment.cu` with L1 cache disabled can be done using:

```
$ nvcc -O2 -arch=sm_20 -Xptxas -dlcm=cg -o readSegment readSegment.cu
```

Example results from running on the specified offsets with and without L1 cache are shown below in Table 14. Note that results may vary across platforms. In general you should observe worse performance with offset 11 relative to offsets 0 and 128 because of unaligned accesses. You should also observe that offset 11 runs slower with caching disabled. This is a result of not being able to re-use partially used cache lines that were fetched into the cache by an earlier unaligned read.

**Table 14: readSegment Performance with Varying Memory Access Offsets and Cache Modes**

| Float Offset | Kernel Execution Time With L1 Cache (µs) | Kernel Execution Time Without L1 Cache (µs) |
|---|---|---|
| 0 | 127.867 | 128.333 |
| 11 | 136.233 | 159.200 |
| 128 | 128.033 | 128.167 |

9 *Run the following command for offsets 0, 11, and 128:*

```
nvprof --metrics gld_efficiency \
       --metrics gld_throughput \
       ./iread_l2 $OFFSET;
```

*Compare the results with L1 enabled and explain the difference.*

Example measurements of `gld_efficiency` and `gld_throughput` with and without L1 cache enabled are shown below in Table 15. There are a few trends of note in these measurements. First, unaligned global load throughput is greater with the L1 cache compared to both without the L1 cache and compared to aligned accesses. This is likely a result of pre-fetching for unaligned accesses when caching is enabled. Also note that load efficiency is more precise without the L1 cache than with it. This is also a result of pre-fetching cache lines, which may lead to varying and unpredictable load efficiency, as well as lost efficiency with the cache.

**Table 15: readSegment Global Load Efficiency and Throughput with Varying Memory Access Offsets and Cache Modes**

| Float Offset | Global Load Efficiency With L1 Cache | Global Load Throughput With L1 Cache | Global Load Efficiency Without L1 Cache | Global Load Throughput Without L1 Cache |
|---|---|---|---|---|
| 0 | 99.66% | 72.74 GB/s | 100.0% | 72.31 GB/s |
| 11 | 49.65% | 134.01 GB/s | 80.0% | 68.93 GB/s |
| 128 | 99.61% | 72.85 GB/s | 100.0% | 72.30 GB/s |

10 *Refer to the file* `simpleMathAoS.cu`. *Define* `innerStruct` *as* `struct __align__(8) innerStruct`, *aligning it to eight bytes. Use* `nvprof` *to compare performance, and explain any difference using* `nvprof` *metrics.*

The `__align__` compiler directive in CUDA ensures that all instances of a type start at the specified byte alignment. For this problem, you specify that all instances of `innerStruct` should start at an eight-byte address alignment. Doing so allows `nvcc` to perform more advanced optimizations in how global memory accesses are performed.

An example implementation of the modified `innerStruct` is provided in `simpleMathAoS-align.cu`. Measuring execution time, the original implementation averaged 0.951 ms and the 8-byte aligned version averaged 0.625 ms. To explain this significant difference in performance, you can use the nvprof metrics `gld_transactions`, `gst_transactions`, `gld_efficiency`, and `gst_efficiency` to demonstrate that without the `__align__` directive a large amount of processor time and bus bandwidth is being wasted. Example values for these metrics for both kernels are shown in Table 16 below. The original implementation is using approximately twice as many memory transactions to load and store values and getting half the bus utilization.

**Table 16: simpleMathAoS Memory Utilization**

| Version | gld_transactions | gst_transactions | gld_efficiency | gst_efficiency |
|---|---|---|---|---|
| simpleMathAoS | 527,520 | 527,352 | 49.69% | 50.00% |
| simpleMathAoS-align | 266,168 | 265,860 | 98.49% | 100.00% |

Because this book does not cover the PTX instruction set in detail, it is not a required part of this question that you look at PTX to explain these performance discrepancies. However, the real reason for the significant performance gains from `__align__` can be seen much clearer in the PTX than in the original source code. If you do so, you will notice that the load and store instructions in the original:

```
ld.global.f32   %f1, [%rd6+4];
ld.global.f32   %f2, [%rd6];
...
st.global.f32   [%rd7+4], %f4;
st.global.f32   [%rd7], %f3;
```

have been replaced with:

```
ld.global.v2.f32    {%f1, %f2}, [%rd6];
st.global.v2.f32    [%rd7], {%f6, %f4};
```

As a result, the `floats` in each `innerStruct` are loaded simultaneously in the same memory instruction and transaction, rather than loading each individually and wasting half of the loaded bytes in each transaction.

11 *Based on the revision in Exercise 10, modify the kernel to read/write variable* `x` *only. Compare the result with the file* `simpleMathSoA.cu`*. Use proper* `nvprof` *metrics to explain the difference.*

By only loading, modifying, and storing the field `x` in `innerStruct`, the new kernel is referencing every other four bytes. Compared to `simpleMathSoA`, you should expect the number of memory transactions to be approximately the same but bus efficiency to drop by half. This can be verified using the `nvprof` metrics `gld_transactions`, `gst_transactions`, `gld_efficiency`, and `gst_efficiency`. Example measurements for these metrics are shown in Table 17 below.

**Table 17: simpleMathAoS-x Memory Utilization**

| Version | gld_transactions | gst_transactions | gld_efficiency | gst_efficiency |
|---|---|---|---|---|
| simpleMathSoA | 265,552 | 265,552 | 98.72% | 100.00% |
| simpleMathAoS-x | 265,720 | 266,028 | 49.33% | 50.00% |

12 *Refer to the file* `writeSegment.cu`*. Make a new kernel,* `readWriteOffset`*, that offsets both reads and writes and run with offsets 32, 33, 64, 65, 128, and 129. Compare performance with* `readOffset` *and* `writeOffset`*, and explain any difference.*

An example implementation of `readWriteOffset` is provided in `readWriteSegment.cu`. The only change made to the `writeOffset` kernel to create the `readWriteOffset` is to index the reads using `k`. Example performance results with the provided offsets are shown below in Table 18. `readWriteOffset` demonstrates the worst performance when accesses are unaligned (that is, when `offset` is not a multiple of 32) because it exhibits both unaligned reads and writes. `readOffset` and `writeOffset` each only exhibit one or the other.

The `nvprof` metrics `gld_efficiency`, `gld_throughput`, `gst_efficiency`, and `gst_throughput` are measured to explain changes in performance from different kernel versions. Example measurements for these metrics are shown in Tables 19 and 20. In general, you should note that load efficiency is worse with unaligned accesses, that load throughput is greater with unaligned accesses due to more bytes being loaded, that store efficiency is worse with unaligned accesses, but that store throughput is unaffected by alignment because stores are not cached in the L1 cache.

**Table 18: readWriteOffset Performance With Varying Offsets**

| Float Offset | readOffset Execution Time (µs) | writeOffset Execution Time (µs) | readWriteOffset Execution Time (µs) |
|---|---|---|---|
| 32 | 128.400 | 126.666 | 126.433 |
| 33 | 136.233 | 165.300 | 170.533 |
| 64 | 128.066 | 126.933 | 126.133 |
| 65 | 136.333 | 165.100 | 170.066 |
| 128 | 127.866 | 126.566 | 125.933 |
| 129 | 136.266 | 164.666 | 170.200 |

**Table 19: readWriteOffset Global Load Performance With Varying Offsets**

| Float Offset | readOffset Global Load Efficiency | writeOffset Global Load Efficiency | readWriteOffset Global Load Efficiency | readOffset Global Load Throughput | writeOffset Global Load Throughput | readWriteOffset Global Load Throughput |
|---|---|---|---|---|---|---|
| 32 | 99.66% | 99.59% | 99.66% | 72.52 GB/s | 72.81 GB/s | 72.51 GB/s |
| 33 | 49.55% | 99.68% | 49.85% | 134.26 GB/s | 54.46 GB/s | 104.90 GB/s |
| 64 | 99.60% | 99.42% | 99.66% | 72.80 GB/s | 72.90 GB/s | 72.77 GB/s |
| 65 | 49.54% | 99.85% | 49.87% | 134.20 GB/s | 54.52 GB/s | 105.08 GB/s |
| 128 | 99.66% | 99.63% | 99.61% | 72.78 GB/s | 72.71 GB/s | 72.78 GB/s |
| 129 | 49.62% | 99.72% | 49.82% | 134.09 GB/s | 54.60 GB/s | 105.10 GB/s |

**Table 20: readWriteOffset Global Store Performance With Varying Offsets**

| Float Offset | readOffset Global Store Efficiency | writeOffset Global Store Efficiency | readWriteOffset Global Store Efficiency | readOffset Global Store Throughput | writeOffset Global Store Throughput | readWriteOffset Global Store Throughput |
|---|---|---|---|---|---|---|
| 32 | 100.0% | 100.0% | 100.0% | 36.19 GB/s | 36.28 GB/s | 36.16 GB/s |
| 33 | 100.0% | 80.0% | 80.0% | 33.26 GB/s | 33.94 GB/s | 32.74 GB/s |
| 64 | 100.0% | 100.0% | 100.0% | 36.30 GB/s | 36.27 GB/s | 36.29 GB/s |
| 65 | 100.0% | 80.0% | 80.0% | 33.28 GB/s | 34.04 GB/s | 32.77 GB/s |
| 128 | 100.0% | 100.0% | 100.0% | 36.31 GB/s | 36.29 GB/s | 36.31 GB/s |
| 129 | 100.0% | 80.0% | 80.0% | 33.27 GB/s | 34.05 GB/s | 32.78 GB/s |

*13 Apply an unrolling factor of four to readWriteOffset and compare performance with the original, using proper metrics with nvprof to explain the difference.*

An example implementation of `readWriteOffsetUnroll4` is available in `readWriteOffsetUnroll.cu`. Example kernel execution times are shown below in Table 21. Interestingly, for `readWriteOffset` unrolling actually causes lost performance. You can use `gld_efficiency`, `gld_throughput`, `gst_efficiency`, `gst_throughput`, and `ipc_instance` to understand why. `readWriteOffsetUnroll4`

demonstrates a significant loss in both load and store throughput. This may indicate that not all loads and stores are being concurrently scheduled.

`ipc_instance` indicates the number of instructions being executed per cycle on the GPU. If each load instruction were being issued concurrently, followed by all store instructions, then you would expect these to be approximately equal to without unrolling. However, you can see another significant loss in performance in this metric. This would indicate that warps are stalling on memory operations, leading to fewer instructions per cycle. There could be a variety of factors causing this. For instance, the compiler might reduce the number of concurrent memory operations to reduce the number of registers required by each thread. It might also might be unable to determine that the reads and writes are independent and can be scheduled concurrently.

**Table 21: readWriteOffsetUnroll4 Performance**

| Kernel | Kernel Execution Time | Global Load Efficiency | Global Load Throughput | Global Store Efficiency | Global Store Throughput | Instructions Per Cycle |
|---|---|---|---|---|---|---|
| readWriteOffset | 127.900 µs | 99.58% | 72.89 GB/s | 100.0% | 36.30 GB/s | 0.683 |
| readWriteOffsetUnroll4 | 132.433 µs | 99.83% | 68.96 GB/s | 100.0% | 34.45 GB/s | 0.473 |

14 *Adjust the execution configuration for kernels `readWriteOffset` and `readWriteOffsetUnroll4` and find the best one, using proper metrics to explain why one is better.*

Table 22 below shows example kernel performance results for `readWriteOffset` and `readWriteOffsetUnroll4` with varying execution configurations. On this particular platform, `readWriteOffset` achieves optimal performance with 256 threads per block and `readWriteOffsetUnroll4` with 128 threads per block.

**Table 22: readWriteOffsetUnroll4 Performance With Varying Execution Configurations**

| Threads Per Block | readWriteOffset Kernel Performance | readWriteoffsetUnroll4 Kernel Performance |
|---|---|---|
| 32 | 300.833 µs | 233.333 µs |
| 64 | 183.667 µs | 157.200 µs |
| 128 | 140.367 µs | **131.567 µs** |
| 256 | **126.533 µs** | 132.667 µs |
| 512 | 128.033 µs | 132.433 µs |
| 1024 | 157.333 µs | 131.633 µs |

15 *Refer to the kernel `tranposeUnroll4Row`. Implement a new kernel, `transposeRow`, to let each thread handle all elements in a row.*

*Compare the performance with existing kernels and use proper metrics to explain the difference.*

An example implementation of `transposeRow` is shown in the `transpose.cu` file provided with the other solutions. Other implementations are also possible. A performance comparison to some of the other transpose kernels is provided in Table 23 below. `transposeRow` demonstrates a significant loss in performance due to poor strided memory access patterns.

To support this explanation of performance loss, global memory throughput and efficiency can be measured. Example measurements using `nvprof` are shown in Table 24. Table 24 demonstrates that the main problem is load performance. Load efficiency is extremely low, due to strided reads. Load throughput is mediocre. Because reads are strided, many unnecessary bytes are being requested and keeping the bus busy. As a result, load throughput is artificially inflated. However, stores are still coalesced across a warp, so store efficiency is good. Store throughput is still low because so much kernel time is spent blocked waiting on loads, that stores never have a chance to consume bus capacity.

**Table 23: transposeRow Performance Compared to Other Transpose Kernels**

| Kernel | Execution Time (ms) |
|---|---|
| transposeRow | 2.839 |
| transposeNaiveRow | 0.700 |
| transposeNaiveCol | 0.626 |
| transposeUnroll4Row | 1.132 |
| transposeUnroll4Col | 0.666 |
| transposeDiagonalRow | 0.818 |
| transposeDiagonalCol | 0.671 |

**Table 24: transposeRow Memory Metrics Compared to Other Transpose Kernels**

| Kernel | Global Memory Load Efficiency | Global Memory Store Efficiency | Global Memory Load Throughput | Global Memory Store Throughput |
|---|---|---|---|---|
| transposeRow | 3.36% | 100.0% | 180.37 GB/s | 6.06 GB/s |
| transposeNaiveRow | 49.95% | 25.0% | 48.33 GB/s | 96.70 GB/s |
| transposeNaiveCol | 6.21% | 100.0% | 471.93 GB/s | 29.31 GB/s |
| transposeUnroll4Row | 49.68% | 25.0% | 31.94 GB/s | 63.59 GB/s |
| transposeUnroll4Col | 6.15% | 100.0% | 446.30 GB/s | 27.42 GB/s |
| transposeDiagonalRow | 49.96% | 25.0% | 43.82 GB/s | 87.66 GB/s |
| transposeDiagonalCol | 6.25% | 100.0% | 434.83 GB/s | 27.15 GB/s |

16 *Refer to the kernel `transposeUnroll4Row`. Implement a new kernel, `transposeUnroll8Row`, to let each thread handle eight*

*elements. Compare the performance with existing kernels and use proper metrics to explain the difference.*

An example implementation of `transposeUnroll8Row` is shown in the `transpose.cu` file provided with the other solutions. A performance comparison with the most similar transpose kernel, `transposeUnroll4Row`, is shown in Table 25. The only change from `transposeUnroll4Row` to `transposeUnroll8Row` is more running threads and a smaller unrolling factor. `transposeUnroll8Row` demonstrates a performance improvement relative to `transposeUnroll4Row`.

Below, Table 26 shows detailed performance metrics on `transposeUnroll8Row` compared to `transposeUnroll4Row`. Note that while efficiencies remain approximately the same, both load and store throughput increase for `transposeUnroll8Row` as a result of more concurrently scheduled I/O operations within each thread.

**Table 25: transposeUnroll8Row Performance Compared to Other Transpose Kernels**

| Kernel | Execution Time (µs) |
|---|---|
| transposeUnroll8Row | 220.367 |
| transposeUnroll4Row | 331.000 |

**Table 26: transposeUnroll8Row Memory Metrics Compared to Other Transpose Kernels**

| Kernel | Global Memory Load Efficiency | Global Memory Store Efficiency | Global Memory Load Throughput | Global Memory Store Throughput |
|---|---|---|---|---|
| transposeUnroll8Row | 46.53% | 25.00% | 27.35 GB/s | 50.89 GB/s |
| transposeUnroll4Row | 45.98% | 25.00% | 32.54 GB/s | 59.85 GB/s |

17 *Refer to the kernels `transposeDiagonalCol` and `transposeUnroll4Row`. Implement a new kernel, `transposDiagonalColUnroll4`, to let each thread handle four elements. Compare the performance with existing kernels and use proper metrics to explain the difference.*

`transpose.cu` contains an example implementation of `transposeDiagonalColUnroll4`. Table 27 shows example performance results of `transposeUnroll4Row`, `transposeDiagonalCol`, and `transposeDiagonalColUnroll4`. The measurements in Table 27 demonstrate that unrolling and avoiding partition camping results in significant performance improvement.

Table 28 explains this improvement based on memory throughput and memory transactions. Unrolling leads to better coalescing of memory coalesces in `transposeDiagonalColUnroll4`, causing fewer memory transactions while still avoiding partition camping.

**Table 27: transposeDiagonalColUnroll4 Performance Compared to Other Transpose Kernels**

| Kernel | Execution Time (µs) |
|---|---|
| transposeDiagonalColUnroll4 | 201.800 |
| transposeUnroll4Row | 333.400 |
| transposeDiagonalCol | 671.800 |

**Table 28: transposeDiagonalColUnroll4 Memory Metrics Compared to Other Transpose Kernels**

| Kernel | Global Memory Load Throughput | Global Memory Store Throughput | Load Transactions | Store Transactions |
|---|---|---|---|---|
| transposeDiagonalColUnroll4 | 455.105 GB/s | 28.397 GB/s | 787,622.07 | 65,908.27 |
| transposeUnroll4Row | 30.069 GB/s | 59.400 GB/s | 328,801.20 | 532,044.80 |
| transposeDiagonalCol | 434.767 GB/s | 27.120 GB/s | 4,208,182.43 | 262,210.67 |

*18 Refer to the program sumArrayZerocpy.cu and implement array addition using unified memory. Compare performance with `sumArrays` and `sumArraysZeroCopy` using `nvprof`.*

An example solution is provided in `sumArrayZerocpyUVA.cu`. The kernel `sumArrayZeroCopyWithUVA` is identical to `sumArrayZeroCopy`, but is added so that `nvprof` distinguishes its performance. Table 29 shows performance measurements for `sumArrays`, `sumArraysZeroCopy`, and `sumArraysZeroCopyWithUVA`. Zero-copy performance with and without UVA is identical, as the address translation for host memory addresses has a very low overhead. Execution time is dominated by I/O overheads caused by reading zero-copy memory over the PCIe bus.

Table 30 below shows the number of read transactions issued to system memory by each of the kernels in this exercise. Because `sumArrays` does not use zero-copy memory no system memory reads are issued. However, `sumArraysZeroCopy` and `sumArraysZeroCopyWithUVA` both perform the same number of system reads. Using UVA does not affect the actual I/O behavior of a program, but rather the way it references addresses on the GPU. This explains the lack of change in performance when using UVA.

**Table 29: sumArraysZeroCopyWithUVA Performance**

| Kernel | Execution Time (ms) |
|---|---|
| sumArrays | 1.975 |
| sumArraysZeroCopy | 22.377 |
| sumArraysZeroCopyWithUVA | 22.368 |

**Table 30: sumArraysZeroCopyWithUVA Metrics**

| Kernel | System Memory Read Transactions |
|---|---|
| sumArrays | 0 |
| sumArraysZeroCopy | 4194304 |
| sumArraysZeroCopyWithUVA | 4194304 |

19 *Refer to the program* `sumMatrixGPUManaged.cu`*. If the warm-up kernel were removed, how would performance change? If you can, measure performance with* `nvprof` *and* `nvvp`*.*

Removing the warm-up kernel in `sumMatrixGPUManaged.cu` causes the transfer of data to the device to be associated with the main kernel whose performance is being reported, rather than the warm-up kernel. As a result, overall kernel performance will decrease dramatically. `nvvp` will show transfers being performed prior the main kernel launch, rather than before the warm-up kernel. `nvprof` can also be used in trace mode to demonstrate this.

20 *Refer to the program* `sumMatrixGPUManaged.cu`*. Would removing the* `memsets` *below affect performance? If you can, check performance with* `nvprof` *or* `nvvp`*.*

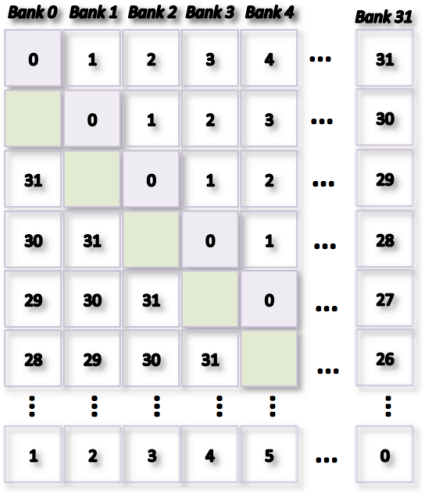
```
memset(hostRef, 0, nBytes);
memset(gpuRef, 0, nBytes);
```

Removing these statements will have no effect on performance. The managed memory will still be resident on the CPU initially, and transferred to the GPU when the first kernel executes.

# Chapter 5

1 *Suppose you have a shared memory tile with dimensions [32][32]. Pad a column to it and then draw an illustration showing the mapping between data elements and banks for a Kepler device in 4-byte access mode.*

An example illustration is shown below.




2 *Refer to the kernel* `setRowReadCol` *in the file* `checkSmemSquare.cu`*. Make a new kernel named* `setColReadRow` *to perform the operations of writing to a row and reading from a column. Check the memory transactions with* `nvprof` *and observe the output.*

An example implementation of `setColReadRow` is provided with the other solutions files in `checkSmemSquare.cu`. Example transaction metrics for `setColReadRow` are shown in Table 31, compared to `setRowReadCol`. Swapping the access patterns for shared memory reads and writes compared to `setRowReadCol` resulted in a swapping of the number of read and write transactions as well. These metrics were generated with the `nvprof` command:

```
$ nvprof --metrics gld_transactions_per_request,gst_transactions_per_request
         --metrics shared_load_transactions_per_request
         --metrics shared_store_transactions_per_request ./checkSmemSquare
```

**Table 31: setColReadRow Transaction Metrics**

| Kernel | Global Load Transactions Per Request | Global Store Transactions Per Request | Shared Memory Load Transactions Per Request | Shared Memory Store Transactions Per Request |
|---|---|---|---|---|
| setRowReadCol | 0.00 | 3.50 | 112.00 | 3.50 |
| setColReadRow | 0.00 | 3.50 | 3.50 | 112.00 |

3 *Refer to the kernel* `setRowReadColDyn` *in the file* `checkSmemSquare.cu`*. Make a new kernel named* `setColReadRowDyn` *that declares shared memory dynamically, and then perform the operations of writing to columns and reading from rows. Check the memory transactions with* `nvprof` *and observe the output.*

An example implementation of `setColReadRowDyn` is provided with the other solutions files in `checkSmemSquare.cu`. Example transaction metrics for `setColReadRowDyn` are shown in Table 32, compared to `setRowReadColDyn`. The measured metrics are identical to the previous example as static and dynamic allocations do not affect load or store transactions. These metrics were generated with the `nvprof` command:

```
$ nvprof --metrics gld_transactions_per_request,gst_transactions_per_request
         --metrics shared_load_transactions_per_request
         --metrics shared_store_transactions_per_request ./checkSmemSquare
```

**Table 32: setColReadRowDyn Transaction Metrics**

| Kernel | Global Load Transactions Per Request | Global Store Transactions Per Request | Shared Memory Load Transactions Per Request | Shared Memory Store Transactions Per Request |
|---|---|---|---|---|
| setColReadRowDyn | 0.00 | 3.50 | 3.50 | 112.00 |
| setRowReadColDyn | 0.00 | 3.50 | 112.00 | 3.50 |

4 *Refer to the kernel `setRowReadColPad` in the file `checkSmemSquare.cu`. Make a new kernel named `setColReadRowPad` that pads by one column. Then, implement the operation of writing by columns and reading from rows. Check the memory transactions with `nvprof` and observe the output.*

An example implementation of `setColReadRowPad` is provided with the other solutions files in `checkSmemSquare.cu`. Example transaction metrics for `setColReadRowPad` are shown in Table 33, compared to `setRowReadColPad`. The reported metrics are identical in both kernels because shared memory padding results in no bank conflicts, which would cause the number of required transactions to increase. These metrics were generated with the `nvprof` command:

```
$ nvprof --metrics gld_transactions_per_request,gst_transactions_per_request
         --metrics shared_load_transactions_per_request
         --metrics shared_store_transactions_per_request ./checkSmemSquare
```

**Table 33: setColReadRowPad Transaction Metrics**

| Kernel | Global Load Transactions Per Request | Global Store Transactions Per Request | Shared Memory Load Transactions Per Request | Shared Memory Store Transactions Per Request |
|---|---|---|---|---|
| setColReadRowPad | 0.00 | 3.50 | 3.50 | 3.50 |
| setRowReadColPad | 0.00 | 3.50 | 3.50 | 3.50 |

5 *Suppose the size of the square shared memory array in `checkSmemSquare.cu` were 16×16 instead of 32×32. How would the number of shared memory transactions change on both Fermi and*

*Kepler devices? Try to draw a picture of the shared memory arrangement in each case.*

Using 16×16 and 32×32 square shared memory does not impact the layout of data in shared memory or the assignment of threads to warps. As a result, there is no change to the shared memory transactions on Fermi or Kepler devices.

6 *Refer to kernel `setRowReadCol` in the file `checkSmemRectangle.cu`. Make a new kernel named `setColReadRow` that writes by columns and reads by rows. Check the memory transactions with `nvprof` and observe the output.*

An example implementation of `setColReadRow` is provided with the other solutions files in `checkSmemRectangle.cu`. Example transaction metrics for `setColReadRow` are shown in Table 34, compared to `setRowReadCol`. Swapping the access patterns for shared memory reads and writes compared to `setRowReadCol` resulted in a swapping of the number of read and write transactions as well. These metrics were generated with the `nvprof` command:

```
$ nvprof --metrics gld_transactions_per_request,gst_transactions_per_request
         --metrics shared_load_transactions_per_request
         --metrics shared_store_transactions_per_request ./checkSmemRectangle
```

**Table 34: setColReadRow Transaction Metrics**

| Kernel | Global Load Transactions Per Request | Global Store Transactions Per Request | Shared Memory Load Transactions Per Request | Shared Memory Store Transactions Per Request |
|---|---|---|---|---|
| setColReadRow | 0.00 | 3.50 | 3.50 | 56.00 |
| setRowReadCol | 0.00 | 3.50 | 56.00 | 3.50 |

7 *Refer to the kernel `setRowReadColPad` in the file `checkSmemRectangle.cu`. Make a new kernel named `setColReadRowPad` that writes by columns and reads by rows. Check the memory transactions with `nvprof` and observe the output.*

An example implementation of `setColReadRowPad` is provided with the other solutions files in `checkSmemRectangle.cu`. Example transaction metrics for `setColReadRowPad` are shown in Table 35, compared to `setRowReadColPad`. These metrics were generated with the `nvprof` command:

```
$ nvprof --metrics gld_transactions_per_request,gst_transactions_per_request
         --metrics shared_load_transactions_per_request
         --metrics shared_store_transactions_per_request ./checkSmemRectangle
```

**Table 35: setColReadRow Transaction Metrics**

| Kernel | Global Load Transactions Per Request | Global Store Transactions Per Request | Shared Memory Load Transactions Per Request | Shared Memory Store Transactions Per Request |
|---|---|---|---|---|
| setColReadRowPad | 0.00 | 3.50 | 7.00 | 7.00 |
| setRowReadColPad | 0.00 | 3.50 | 3.50 | 3.50 |

8 *Refer to the file* `reduceInteger.cu`*. Test block sizes of 64, 128, 512, and 1024. Measure elapsed times for the kernels with* `nvprof`*. Determine the best execution configuration.*

Example performance results from testing `reduceInteger.cu` on varying block sizes are shown below in Table 36. A copy of `reduceInteger.cu` that allows configuration of block sizes from the command line is provided with the other solutions. In general, the best execution configuration is 512 threads per block, though `reduceNeighboredGmem` and `reduceNeighboredSmem` achieve better performance with 128 threads per block.

**Table 36: reduceInteger Performance on Varying Block Sizes**

| Kernel | Execution Time in µs (64 threads) | Execution Time in µs (128 threads) | Execution Time in µs (512 threads) | Execution Time in µs (1024 threads) |
|---|---|---|---|---|
| reduceGmem | 1185.85 | 670.26 | **578.49** | 951.07 |
| reduceSmem | 695.264 | 465.77 | **430.18** | 644.46 |
| reduceSmemDyn | 673.95 | 441.55 | **408.81** | 617.39 |
| reduceGmemUnroll | 353.79 | 213.55 | **198.41** | 321.02 |
| reduceSmemUnroll | 191.17 | 137.422 | **129.65** | 228.18 |
| reduceSmemUnrollDyn | 190.44 | 133.92 | **128.88** | 223.52 |
| reduceNeighboredGmem | 1790.43 | **1334.47** | 1582.42 | 2275.20 |
| reduceNeighboredSmem | 1288.79 | **1115.73** | 1349.62 | 1761.73 |

9 *Refer to the kernel* `stencil_1d_read_only`*. Write a kernel that uses global memory to store the finite difference coefficients. Compare three kernels with* `nvprof`*: the kernel using constant cache, the one using read-only cache, and the new one using global memory with L1 cache enabled.*

An example implementation of all three kernels is included with the other solutions in `constantReadOnlyGlobal.cu`. Running the executable with `nvprof` is done using:

```
$ nvprof ./constantReadOnlyGlobal
```

and produces the performance results shown in Table 37. While using the constant cache produces a minor performance improvement, the read-only cache and global

memory accesses perform similarly. This indicates that the coefficients are already being stored in L1 cache when directly accessing global memory, and that explicitly using the read-only cache does not have a significant benefit on access latency as a result.

**Table 37: constantReadOnlyGlobal Performance**

| Kernel | Execution Time in ms |
|---|---|
| stencil_1d | 2.699 |
| stencil_1d_read_only | 2.806 |
| stencil_1d_global | 2.806 |

10 *Refer to kernel* `test_shfl_up` *in the file* `simpleShfl.cu`*, invoke it with a negative delta as follows:*

```
test_shfl_up<<<1, BDIMX>>>(d_outData, d_inData, -2);
```

*Check the results and reason about the output.*

A sample output from `test_shfl_up` with a negative delta is shown below:

```
shfl up         :  0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15
```

Passing a negative value to `__shfl_up` leads to that negative value being cast to a very large unsigned integer, as `delta` is defined in the function declaration as an unsigned integer:

```
int __shfl_up(int var, unsigned int delta, int width=warpSize);
```

As a result, passing a negative value means that none of the deltas correspond to existing threads. The calling thread's value is then returned from `__shfl_up`. Hence, the output for `test_shfl_up` with a negative delta is identical to the initial data.

11 *Refer to kernel* `test_shfl_wrap` *in the file* `simpleShfl.cu`*, make a new kernel that can generate the following result:*

```
Initial: 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
Result : 2  4  6  8 10 12 14 16 18 20 22 24 26 28 14 16
```

An example implementation of the kernel for this exercise is shown in `simpleShflAltered.cu` with the name `test_shfl_wrap_plus`. `test_shfl_wrap_plus` increments the current thread's value by th value of the thread which is two threads above, and outputs the result. Because the width of the shuffle operation is not modified and the `__shfl` function is used, the top two threads' references wrap around to the bottom two threads.

12 *Refer to the kernel* `test_shfl_xor` *in the file* `simpleShfl.cu`*, make a new kernel that can generate the following result:*

```
Initial: 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
Result : 1  1  5  5  9  9 13 13 17 17 21 21 25 25 29 29
```

An example implementation of the kernel for this exercise is shown in `simpleShflAltered.cu` with the name `test_shfl_xor_plus`. By passing 1 as the mask, the output above is produced. `test_shfl_xor_plus` simple performs a `__shfl_xor` with the provided mask and increments this thread's value with the received value. Passing 1 as the mask results in every thread even thread receiving the value of the odd thread above it, and every odd thread receiving the value of the even thread below it.

13 *Refer to the kernel `test_shfl_xor_array` in the file `simpleShfl.cu`, make a new kernel that just performs one operation as follows:*

```
value[3] = __shfl_xor (value[0], mask, BDIMX);
```

*Check the result and reason about its cause.*

A modified kernel is shown in `simpleShflAltered.cu` with the name `test_shfl_xor_array_03swap`. The output from the specified operation is:

```
shfl array 0 3         :  0  1  2  4  4  5  6  0  8  9  10  12  12  13  14  8
```

Essentially, each thread is responsible for a four-element chunk. Each thread `tid` performs its shuffle operation with either `tid+1` if `tid` is even, or `tid-1` if `tid` is odd. This shuffle operation replaces the fourth element of the other thread's chunk with the first element of the current thread's chunk. Therefore, the chunk that thread 1 is responsible for (indices 4-7 inclusive) has the last element (index 7) replaced with the first element in thread 0's chunk (index 0). This pattern is then repeated for all other even-odd pairs of threads (2 and 3, 4 and 5, etc).

14 *Refer to the kernel `test_shfl_wrap` in the file `simpleShfl.cu`. Make a new kernel that can shift double-precision variables in a wrap-around warp approach.*

An example solution for this exercise is included in `simpleShflAltered.cu` with the name `test_shfl_wrap_double`. This function splits the shuffle of the double-precision floating-point in half to support shuffling values that are twice the length of the data type supported by the shuffle functions. The added kernel is included inline below.

```
__global__ void test_shfl_wrap_double (double *d_out, double *d_in,
        int const offset)
{
    // Assumes that a double is twice the size of an int
    int *iptr = (int *)d_in;
    int *optr = (int *)d_out;

    // Shuffle first half of the double
    optr[2 * threadIdx.x] = __shfl(iptr[2 * threadIdx.x], threadIdx.x + offset,
```

```
            BDIMX);
    // Shuffle second half of the double
    optr[2 * threadIdx.x + 1] = __shfl(iptr[2 * threadIdx.x + 1],
            threadIdx.x + offset, BDIMX);
}
```

15 *Refer to the inline function* `warpReduce` *in the file* `reduceIntegerShfl.cu`*. Write an equivalent function that uses the* `__shfl_down` *instruction instead.*

An example implementation of `warpReduce` using the `__shfl_down` function is shown below:

```
__inline__ __device__ int warpReduce(int localSum)
{
    localSum += __shfl_down(localSum, 1);
    localSum += __shfl_down(localSum, 2);
    localSum += __shfl_down(localSum, 4);
    localSum += __shfl_down(localSum, 8);
    localSum += __shfl_down(localSum, 16);

    return localSum;
}
```

# Chapter 6

1 *Define the term "CUDA stream". What kind of operations can be placed in a CUDA stream? What are the main benefits of using streams in an application?*

A CUDA stream refers to a sequence of asynchronous CUDA operations that execute on a device in the order issued by the host code. A stream encapsulates these operations, maintains their ordering, permits operations to be queued in the stream to be executed after all preceding operations, and allows for querying the status of queued operations.

Operations that can be placed in a CUDA stream include kernels and memory transfers.

2 *How do events relate to streams? Give an example where a CUDA event would be useful and allow you to implement logic that you could not efficiently implement with streams alone.*

Events can be enqueued in a CUDA stream as markers. They allow you to query the progress of a CUDA stream at a specific point in execution, rather than for the completion of all previous operations in the stream.

For example, say you had 4 operations to enqueue in a CUDA stream: an asynchronous copy to the device, an asynchronous kernel, an asynchronous copy back to the host, and another asynchronous kernel that consumes the output of the

first kernel and modifies it. Say you also had host computation that depended on the output transferred back to the host. Without events, you would have to either wait for all four operations to complete, or not be able to asynchronously issue the second kernel without blocking on the first three operations in the stream. With CUDA events, an event can be enqueued after the transfer back to the host that the host can block on, rather than blocking on the full stream.

3 *What can cause false dependencies on GPUs? How do these causes differ between the Fermi and Kepler architectures? How does Hyper-Q help to limit false dependencies?*

False dependencies between operations in different CUDA streams can be caused by the hardware work queue. If the operation at the head of the work queue depends on another operation that has not completed yet, it will prevent all other operations behind it in the work queue from executing regardless of whether their dependencies have been satisfied or not.

This is a particularly large problem on Fermi devices because there is only a single hardware work queue. Any operation blocking at the head of that work queue implies that all operations behind it must also block. This problem is somewhat relieved by a feature of the Kepler architecture called Hyper-Q, which adds 32 hardware work queues to the GPU (though by default only 8 are enabled). While false dependencies are still possible on Kepler, distributing streams among multiple work queues makes them far less likely because a single blocked operation in a single work queue does not force all other operations to wait.

4 *Describe the difference between explicit and implicit synchronization. What are examples of specific CUDA API functions that create implicit host-device synchronization points?*

Explicit synchronization is performed by a CUDA programmer to manually ensure that certain work is not started until all previous work that it depends on completes. Explicit synchronization can be done using `cudaDeviceSynchronize`, `cudaEventSynchronize`, `cudaStreamSynchronize`, or `cudaStreamWaitEvent`.

Implicit synchronization is performed as a side effect of other function calls whose main task is some other operation. For example, calling `cudaMemcpy` on the host implicitly synchronizes device execution to ensure that all ongoing work completes before the `cudaMemcpy` is performed. However, `cudaMemcpy`'s main purpose is the transfer of data.

5 *How do depth-first and breadth-first ordering differ when executing work from CUDA streams? In particular, how does the Fermi architecture benefit from breadth-first ordering of work?*

Depth-first and breadth-first ordering describe different total orderings of operations enqueued to all of the CUDA streams in an application. In depth-first order, all operations associated with the same CUDA stream are enqueued in chunks to that stream, generally with no operations enqueued to other streams during that period. In breadth-first order, the enqueuing of operations to different streams are interleaved with each other.

Because Fermi only has a single hardware work queue and events in different streams are more likely to be independent (unless `cudaStreamWaitEvent` is used), using breadth-first order reduces the chance for false dependencies in that work queue because earlier operations in program execution are placed earlier in the work queue.

6 *List the different types of CUDA overlapping. Describe the techniques that would be required to implement each.*

There are four different types of work overlap in CUDA: overlapped host computation and device computation, overlapped host computation and host-device data transfer, overlapped host-device data transfer and device computation, and concurrent device computation.

Overlapped host computation and device computation is implemented by launching an asynchronous kernel and performing useful work on the host while it executes.

Overlapped host computation and host-device data transfer is implemented by launching one or more asynchronous data transfers in CUDA streams and performing useful work on the host while that copy completes. In general, an asynchronous transfer is associated with an asynchronous kernel so you will often see host and device computation overlap associated with host computation and transfer overlap.

Overlapped data transfer and device computation is implemented by launching both an asynchronous kernel and an asynchronous transfer, each in different CUDA streams and with no other dependencies between them. Because these operations do not rely on any shared hardware resources (that is, CUDA cores vs. the PCIe bus) and there are no software dependencies expressed, they are free to execute concurrently.

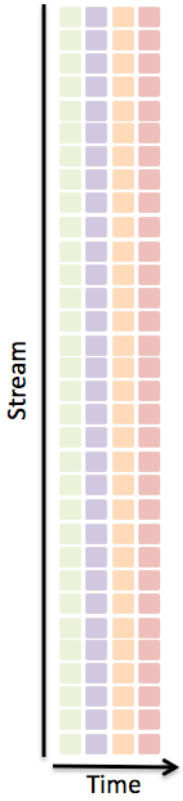
Concurrent device computation is implemented by launching multiple asynchronous kernels in different CUDA streams. As long as sufficient computational resources are available to execute both kernel launches and there are no false dependencies between kernels, they will run concurrently.

7 *Draw the timeline you would expect to be produced from running* `simpleHyperqBreadth` *on a Fermi device with* `nvvp`*, as follows:*

```
$ nvvp ./simpleHyperqBreadth
```

*Assume that 32 streams are used. Explain the reasoning behind the timeline you drew.*

An example illustration is shown below.



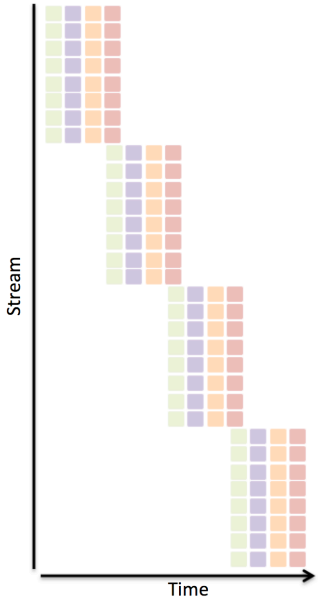

Assuming that there are sufficient computational resources to run all thread blocks concurrently (which may not be the case), running `simpleHyperqBreadth` on a Fermi device will produce the schedule of kernels above because there are no inter-stream true dependencies and no work queue-induced false dependencies. Hence, all streams can run concurrently while all kernels within the same stream will be serialized.

*8 Draw the timeline produced by the command below on a Kepler device:*

```
$ nvvp ./simpleHyperqDepth
```

*Assume that 32 streams are used. Explain the reasoning behind the timeline you drew.*

An example illustration is shown below.




By default, Kepler devices only have 8 work queues enabled. Therefore, running 32 streams in depth-first order will induce false-dependencies between every eighth stream, preventing it from executing until the last element in the previous stream in the same work queue has started.

9 *Refer to `simpleCallback.cu`, and put the callback point after the second kernel launch. Run it with `nvvp` and observe the difference.*

If `simpleCallback` is run on a Fermi device (that is, with a single hardware work queue) then the depth first launching of all kernels leads to serialization of nearly all kernel launches. With the modified `simpleCallback`, each callback will be shown following the completion of the second kernel launch in each stream and so they will be spaced out due to false dependencies.

If `simpleCallback` is run on a Kepler device (that is, with multiple hardware work queues) it is likely that all streams will be able to be scheduled concurrently and the callbacks may or may not be ordered in the final output or in the `nvvp` view depending on variations in device performance and runtime scheduling. However, all callbacks will still appear following the completion of the second kernel launch in each stream.

# Chapter 7

1 *Translate the following sequence of arithmetic operations into calls to the double-precision intrinsic function `__fma_rn`. Then, try the same with calls to `__dmul_rn` and `__dadd_rn`:*
`a * b + c * d + e * f * g`

To express the above operation with the double-precision floating-point multiple-add operation (`__fma_rn`) requires the following statement:

```
__fma_rn(a, b, __fma_rn(c, d, __fma_rn(__fma_rn(e, f, 0.0), g, 0.0)))
```

To instead use separate `__fmul_rn` and `__fadd_rn` calls requires:

```
__fadd_rn(__fadd_rn(__fmul_rn(a, b), __fmul_rn(c, d)), __fmul_rn(__fmul_rn(e, f), g))
```

Note that using `__fma_rn` requires four operations while using `__dadd_rn` and `__dmul_rn` requires six.

2 *Write a program that, given a floating-point value, calculates the next largest and smallest floating-point value that can be correctly represented using the same storage type. Do this for both single-precision and double-precision types and compare the results. Are you able to find any values for which single-precision has a more accurate representation? If not, do you still think they exist?*

An example solution is provided in `closestFP.cu`. Because of the IEEE floating-point standards and the definition of the C programming language, the set of single-precision floating-point values must be a subset of the set of double-precision floating-point values. The C99 standard states:

```
The set of values of the type float is a subset of the set of values of the type
double
```

Therefore, single-precision is never more accurate than double precision.

3 *You have seen that unsafe accesses from multiple threads cause unpredictable results. But you have also seen that some guarantees can be extracted from unsafe accesses (that is, if every thread writes a 1 then the final value cannot magically be something else). Consider the following code snippet:*

```
int n = 0;
int main(int argc, char **argv) {
    for (i = 0; i < 5; i++) {
        int tmp = n;
        tmp = tmp + 1;
        n = tmp;
    }
    return 0;
}
```

*If a single thread ran this application, you would expect the final output to be 5. What if 5 threads ran the same loop in parallel? What are the largest and smallest values `n` could have? The largest should be self-evident: 25, with 5 increments from 5 threads. However, reasoning about the smallest possible value is more difficult. Hint: `n` can be less than 5, but it is up to you to figure out why. As a follow-up, how would you use atomic instructions and local reduction to improve the performance and correctness of this code snippet when run in parallel?*

With five threads running this five-iteration loop and with no protection from concurrent accesses, the lowest value that `n` can reach is two. Understanding how to reach this result is easiest when working backwards from the final result.

For the final output to be two, a thread must have read a value of one from `n`, incremented it, and then written two. That means that another thread wrote one, implying that it also initially read zero (which is also the starting value for `n`). This accounts for the behavior of two of the five threads. However, for this behavior to occur the results of the other three threads must have been overwritten. Two valid executions could accomplish this. Either 1) all three threads began and completed execution between the first thread reading zero and writing one, or 2) all three threads began and completed execution between the final thread reading one and writing two. Both execution orderings are valid.

To improve the performance of this code snippet, a thread-local variable should be used to calculate the local result of performing the five increments in each thread. Then, for correctness, atomic instructions should be used to add each local sum to the global output variable `n`.

4 *Based on the `myAtomicAdd` example, implement a custom `myAtomicMin` device function based on `atomicCAS`.*

`myAtomicMin` could be implemented as shown below. The main change to `myAtomicAdd` is the operation performed as the final argument to `atomicCAS`, which takes the minimum of two values rather than summing them.

```
__device__ int myAtomicMin(int *address, int other)
{
    // Create an initial guess for the value stored at *address.
    int guess = *address;
    int oldValue = atomicCAS(address, guess, guess < other ? guess : other);

    // Loop while the guess is incorrect.
    while (oldValue != guess)
    {
        guess = oldValue;
        oldValue = atomicCAS(address, guess, guess < other ? guess : other);
    }

    return oldValue;
}
```

5 *Based on the floating-point `myAtomicAdd` example, implement atomic double-precision floating-point addition using `atomicCAS`.*

An example implementation of `myAtomicDoubleAdd` is below, based on the atomic floating-point add example demonstrated in Chapter 7, plus the use of `__double_as_longlong` and `__longlong_as_double` to convert values between types.

```
__device__ double myAtomicDoubleAdd(double *address, double incr) {
    // Convert address to point to a supported type of the same size
```

```
    unsigned long long int *typedAddress = (unsigned long long int *)address;

    // Store the expected and desired double values as an unsigned long long int
    double currentVal = *address;
    unsigned long long int expected = __double_as_longlong (currentVal);
    unsigned long long int desired = __double_as_longlong (currentVal + incr);

    unsigned long long int oldIntValue = atomicCAS(typedAddress, expected, desired);
    while (oldIntValue != expected) {
        expected = oldIntValue;
        /*
         * Convert the value read from typedAddress to a double, increment,
         * and then convert back to an unsigned long long int
         */
        desired = __double_as_longlong (__longlong_as_double(oldIntValue) + incr);
        oldIntValue = atomicCAS(typedAddress, expected, desired);
    }
    return __longlong_as_double(oldIntValue);
}
```

6 *The examples in this chapter used atomic operations on global memory locations. CUDA 32-bit atomics are also supported on shared memory for GPUs above compute capability 1.2. Based on what you learned about conflicts in shared memory in Chapter 5, what additional issues do you see arising when using atomic instructions in shared memory?*

Shared memory bank conflicts occur when multiple threads in the same warp access addresses in shared memory that fall in the same shared memory bank. Using atomic instructions to access shared memory implies that multiple threads may access the same location, and therefore the same shared memory bank. Otherwise, atomic instructions would not be necessary. Therefore, the use of atomic instructions implies that bank conflicts are more likely to occur which may lead to increased serialization of atomic instructions, beyond the serialization you would expect from atomic instructions simply interfering with each other.

7 *Try compiling the* `nbody.cu` *example down to PTX with and without the* `-use_fast_math` *compiler flag. How does the number of instructions generated change?*

Using `-use_fast_math` drastically reduces the number of PTX instructions generated by the CUDA compiler. Recall that you can generated the PTX instructions for a `.cu` file using:

```
$ nvcc -ptx -o foo.ptx foo.cu
```

Without `-use_fast_math`, approximately 800 PTX instructions are generated. With it, approximately 150 are generated instead. These values may vary from one version of the compiler to another, but in general you should see many more PTX instructions without fast math enabled.

8 *In* `nbody.cu`*, replace the use of* `atomicAdd` *with an optimized, parallel reduction function based on* `reduceSmemShfl` *from Chapter 5. Note that you will still need to perform global aggregation of results*

*using an* `atomicAdd`*. Does this affect performance? Why or why not?*

An example solution is available in `nbody_reduction.cu`. With `nbody_reduction`, performance should improve on larger data sets. The threshold at which `nbody_reduction` outperforms the original implementation will vary across hardware platforms. This is a result of far fewer atomic operations being performed, in favor of local reduction within a thread block.

9 *Try toggling individual compiler flags when building* `nbody.cu`*. Are you able to find a combination that performs better than the example provided? Why might performance improve? Can you find a median point with some but not all optimization flags enabled that balances performance and numerical accuracy?*

The set of instruction flags that can be toggled include: `--ftz`, `--prec-div`, `--prec-sqrt`, `--fmad`, and `-use_fast_math` (though `-use_fast_math` implies `--ftz=true`, `--prec-div=false`, and `--prec-sqrt=false`). All of these flags are relevant to this exercise as `nbody` contains operations that are affected by each. There are a number of ways to approach this question, and performance results will obviously vary from platform to platform. Below an example approach in exploring the full spectrum of numerical accuracy and performance results is presented.

First, you can start by analyzing the performance and correctness of pairs of compiler flags. The performance and correctness results are shown in Tables 38 and 39.

**Table 38: Performance of Paired Compiler Flags in Seconds**

| | --fmad | --prec-sqrt | --prec-div | --ftz |
|---|---|---|---|---|
| -use_fast_math | 4.001769 | 5.754815 | 7.162551 | 3.971537 |
| --fmad | | 5.387539 | 5.922986 | 3.972937 |
| --prec-sqrt | | | 8.466046 | 5.139863 |
| --prec-div | | | | 3.618273 |

**Table 39: Correctness of Paired Compiler Flags**

| | --fmad | --prec-sqrt | --prec-div | --ftz |
|---|---|---|---|---|
| -use_fast_math | 1.288646 | 1.158045 | 1.137092 | 1.171180 |
| --fmad | | 1.189561 | 1.229414 | 1.288646 |
| --prec-sqrt | | | 0.000000 | 1.158045 |
| --prec-div | | | | 1.137092 |

Based on these results, you can conclude a number of different things. First, `--ftz` seems to generally cause a large improvement in performance but loss in precision. Enabling `--prec-div` causes the most loss in performance, though -

`-prec-sqrt` also has a negative effect. However, only when both of these flags are enabled is `nbody` numerically equivalent to the host implementation. `--fmad` seems less significant to the performance or numerical accuracy of this example, though you can still observe some impact on both.

Based on this information, you could produce three sets of flags: one optimized for performance, one optimized for accuracy, and a middle ground that would offer a balance of both. The first two are the simplest and were already covered in the chapter. Achieving the best performance is possible with:

```
--ftz=true --prec-div=false --prec-sqrt=false --fmad=true -use_fast_math
```

Achieving the best numerical accuracy is possible with:

```
--ftz=false --prec-div=true --prec-sqrt=true --fmad=false
```

However, finding a mix of the two is more complicated. During real-world application development, it would be up to you to experiment further to find a set of flags that meet application requirements. In this case, one option would be to use:

```
--ftz=true --prec-div=true --prec-sqrt=true --fmad=true -use_fast_math
```

which provides some balance of performance and accuracy, without going to the extreme in either direction. Table 40 below is an extension of Table 7-8 that includes results for the intermediate options above.

**Table 40: Changes to NBody Performance and Accuracy with Compiler Flags**

| Description | `--ftz` | `--prec-div` | `--prec-sqrt` | `--fmad` | `-use_fast_math` | Time (ms) | Error |
|---|---|---|---|---|---|---|---|
| All flags set to maximize performance | True | False | False | True | True | 5336 | 4.6946 |
| All flags set to maximize numerical accuracy | False | True | True | False | False | 12042 | 0.0000 |
| Flags set to balance accuracy and performance | True | True | True | True | True | 6662 | 1.1803 |

*10 Rewrite `nbody.cu` so that it directly calls intrinsic functions. Is there any difference between the PTX generated from `nbody.cu` when comparing your version that explicitly calls intrinsic functions to the provided code with `-use_fast_math` set? Keep in mind other*

*optimizations that are implicitly enabled by `-use_fast_math`. Can you manually write an `nbody.cu` that is equivalent to the version generated by `-use_fast_math`? If not, compare the number of PTX instructions to see how close you can come.*

An example implementation of `nbody` that explicitly calls intrinsic functions is available in `nbody_intrinsic.cu`. Compiling it with:

```
$ nvcc -O2 -arch=sm_20 -ptx nbody_intrinsic.cu -o nbody_intrinsic.ptx
```

and compiling `nbody.cu` with:

```
$ nvcc -O2 -arch=sm_20 -use_fast_math nbody.cu -o nbody.ptx
```

allows you to compare the PTX instructions generated when intrinsic functions are used and when `-use_fast_math` is used. In general, you should note that using intrinsic functions is a subset of the optimizations `-use_fast_math` performs, and that compiling with `-use_fast_math` produces a significantly shorter sequence of instructions than simply using intrinsic functions does. For example, on a sample platform with CUDA 6, compiling `nbody.cu` without `-use_fast_math` produced ~800 instructions, compiling `nbody.cu` with `-use_fast_math` produced ~180 instructions, and compiling `nbody_intrinsic.cu` produced ~500 instructions.

# Chapter 8

1 *What step is missing from the following cuBLAS workflow:*
   1. *Read input data from data file in column-major order.*
   2. *Allocate device memory with `cudaMalloc`.*
   3. *Configure a cuBLAS handle with `cublasCreate`.*
   4. *Execute matrix-matrix multiplication with `cublasSgemm`.*
   5. *Fetch results with `cudaGetMatrix`.*

This cuBLAS workflow is missing a step that transfers the input (which is already in column-major order) to the GPU. Otherwise, step 4 will execute matrix-matrix multiplication on uninitialized data.

2 *Write a function that takes as input a single dense matrix `A`, the number of rows `M`, and the number of columns `N` and uses a pipeline of cuSPARSE format conversion functions to convert it to the COO format. Assume that this is not a submatrix of a larger matrix and is in row-major order. Your function signature should be:*

```
void dense2coo(float *A, int M, int N,
    float **values, int **row_indices, int **col_indices, );
```

An example implementation of the solution to this exercise is provided below. It pipelines `cusparseSdense2csr` and `cusparseScsr2coo` to implement a dense-to-COO conversion.

```
void dense2coo(float *A, int M, int N, float **values, int **row_indices,
        int **col_indices) {
    cusparseHandle_t handle = 0;
    cusparseMatDescr_t descr = 0;
    float *dA, *dCsrValA;
    int *dNnzPerRow, *dCsrRowPtrA, *dCsrColIndA, *dCooRowIndA;
    int totalNnz;

    CHECK_CUSPARSE(cusparseCreate(&handle));
    CHECK_CUSPARSE(cusparseCreateMatDescr(&descr));
    CHECK_CUSPARSE(cusparseSetMatType(descr, CUSPARSE_MATRIX_TYPE_GENERAL));
    CHECK_CUSPARSE(cusparseSetMatIndexBase(descr, CUSPARSE_INDEX_BASE_ZERO));

    CHECK(cudaMalloc((void **)&dA, M * N * sizeof(float)));
    CHECK(cudaMalloc((void **)&dNnzPerRow, sizeof(int) * M));

    CHECK(cudaMemcpy(dA, A, sizeof(float) * M * N, cudaMemcpyHostToDevice));

    CHECK_CUSPARSE(cusparseSnnz(handle, CUSPARSE_DIRECTION_ROW, M, N, descr, dA,
                                M, dNnzPerRow, &totalNnz));

    CHECK(cudaMalloc((void **)&dCsrValA, sizeof(float) * totalNnz));
    CHECK(cudaMalloc((void **)&dCsrRowPtrA, sizeof(int) * (M + 1)));
    CHECK(cudaMalloc((void **)&dCsrColIndA, sizeof(int) * totalNnz));
    CHECK(cudaMalloc((void **)&dCooRowIndA, sizeof(int) * totalNnz));

    CHECK_CUSPARSE(cusparseSdense2csr(handle, M, N, descr, dA, M, dNnzPerRow,
                                      dCsrValA, dCsrRowPtrA, dCsrColIndA));

    CHECK_CUSPARSE(cusparseXcsr2coo(handle, dCsrRowPtrA, totalNnz, M,
                                    dCooRowIndA, CUSPARSE_INDEX_BASE_ZERO));

    *values = (float *)malloc(sizeof(float) * totalNnz);
    *row_indices = (int *)malloc(sizeof(int) * totalNnz);
    *col_indices = (int *)malloc(sizeof(int) * totalNnz);
    CHECK(cudaMemcpy(*values, dCsrValA, sizeof(float) * totalNnz,
                     cudaMemcpyDeviceToHost));
    CHECK(cudaMemcpy(*row_indices, dCooRowIndA, sizeof(int) * totalNnz,
                     cudaMemcpyDeviceToHost));
    CHECK(cudaMemcpy(*col_indices, dCsrColIndA, sizeof(int) * totalNnz,
                     cudaMemcpyDeviceToHost));

    CHECK(cudaFree(dA));
    CHECK(cudaFree(dNnzPerRow));
    CHECK(cudaFree(dCsrValA));
    CHECK(cudaFree(dCsrRowPtrA));
    CHECK(cudaFree(dCsrColIndA));
    CHECK(cudaFree(dCooRowIndA));
    CHECK_CUSPARSE(cusparseDestroy(handle));
}
```

3 *Use the random matrix generation function `generate_random_dense_matrix` in `cusparse.cu` to generate two random dense matrices and perform matrix-matrix multiplication using cuSPARSE.*

An example solution for this exercise is provided in `cusparse-matrix-matrix.cu`.

4 *Modify the code you wrote for question 3 to operate on double-precision floating-point values. Note that this will require changes to data initialization, storage, and the cuSPARSE functions used. Use* `nvprof` *to measure and explain any performance differences.*

An example implementation is provided in `cusparse-matrix-matrix-double.cu`. Using `nvprof` to run both the original, single-precision implementation and the new, double-precision version will likely reveal timing information on a number of kernels, copies to the GPU, copies from the GPU, and a `memset` on the GPU. Table 41 below includes sample timing information generated from `nvprof`. On the five kernels that amount for the least amount of execution time, no significant difference in performance is observed between single- and double-precision values. However, the more time-consuming kernels and all memory operations demonstrate significant increases in execution time, in some cases more than doubling.

**Table 41: cusparse-matrix-matrix nvprof output**

| CUDA Operation | Single-Precision | Double-Precision |
|---|---|---|
| [CUDA memcpy HtoD] | 2.2713 ms | 4.8145 ms |
| [CUDA memcpy DtoH] | 1.3985 ms | 3.1606 ms |
| [CUDA memset] | 42.208 µs | 74.143 µs |
| csrMm_hyb_core <float, int=7, int=6, …> | 16.726 ms | 26.378 ms |
| cusparseDenseToCsr_kernel2 | 445.80 µs | 543.08 µs |
| cusparseParseDenseByRows_kernel | 177.27 µs | 209.73 µs |
| csrMm_hyb_core <float, int=7, int=4, …> | 84.690 µs | 100.03 µs |
| cusparseInclusive_scan_domino_v1_core | 5.3630 µs | 5.3490 µs |
| cusparseInclusive_localscan_core | 4.7910 µs | 4.7750 µs |
| cusparseIreduce_domino_core | 4.6470 µs | 4.6490 µs |
| cusparseInclusive_scan_merge_core | 2.6460 µs | 2.6130 µs |
| cusparseDense2CsrCopySetBase_kernel | 1.7010u µs | 1.6790 µs |

5 *Take the* `generate_random_dense_matrix` *function from* `cublas.cu`*. First, reorder the outer loops to iterate over rows and then columns (the opposite of what it is doing right now) without modifying the indices used to reference the array* `A`*. Use the* `seconds` *function to compare execution time of* `generate_random_dense_matrix` *before and after this change. If there is no significant difference, try increasing the value of* `M` *and/or*

*N. What do you observe? What causes this difference in performance? Next, modify the indices used to reference the array A so that the array is now in row-major order. Re-measure performance. How did it change, and why?*

An example of these modifications is included in access-ordering.cu. generate_random_dense_matrix_iter implements a version with re-ordered loops. generate_random_dense_matrix_row implements a version with re-ordered loops and row-major ordering in the array A. In general, you should observe that performance drops dramatically with generate_random_dense_matrix_iter but improves back to match the original version with generate_random_dense_matrix_row. This is a result of poor cache-locality when accessing a multi-dimensional array across its outermost dimension, rather than its innermost dimension. For example, on a sample platform the original implementation completed in 14.1 seconds, the re-ordered loop version completed in 73.8 seconds, and the final row-major version completed in 14.1 seconds again.

6 *Using a cuBLAS Level 3 function and the generate_random_dense_matrix function from cublas.cu, perform a matrix-matrix multiplication.*

An example solution is provided in cublas-matrix-matrix.cu. The cublasSgemm function is used to perform multiplication of the A and B matrices to produce the matrix C.

7 *Add CUDA streams to the code you wrote for question 5. You may only use asynchronous functions for transferring data between the host and device (for example, cublasSetMatrixAsync, cublasGetMatrixAsync). Recall that all executable functions in cuBLAS are already asynchronous by default.*

An example implementation of an asynchronous cublas-matrix-matrix implementation is provided in cublas-matrix-matrix-async.cu. It uses cublasSetStream to associate a CUDA stream with all computational cuBLAS functions and uses asynchronous versions of cublasSetMatrix and cublasGetMatrix, associated with that same stream.

8 *cuFFT supports both forward and inverse FFTs. In the cufft.cu example from this chapter, cufftExecC2C receives CUFFT_FORWARD as its last argument to indicate that a forward FFT is desired. What would you need to add to the example to add an inverse operation following the forward operation? After you do so, how do the outputs change? How do these new outputs relate to the original signal*

*samples? Keep in mind, FFTs often require normalization, as they retain information on the frequencies in the signal but not the amplitude.*

An example of the changes required for this exercise is shown in `cufft-inverse.cu`. The only change required for a new inverse operation is an additional call to `cufftExecC2C` with the flag `CUFFT_INVERSE`, instead of `CUFFT_FORWARD`. With that change, the outputs become the same as the inputs in terms of relative magnitudes but generally are much larger in terms of absolute magnitudes. `cufft-inverse.cu` includes a normalization step on the CPU that returns the maximum magnitude for the calculated values to 1.0.

9 *What is the difference between pseudo-random and quasi-random random sequences of numbers?*

Each sampling of a pseudo-random sequence of numbers is statistically independent. Earlier values samples from a pseudo-random sequence do not affect later values.

On the other hand, values sampled from a quasi-random sequence of numbers are not statistically independent and the values sampled earlier in the sequence will affect values sampled later. A quasi-random sequence attempts to spread observed values evenly over the valid range of values.

10 *Consider an application you have worked on in the past that used random numbers. How would the behavior of that application differ between using a pseudo-random number generator and a quasi-random number generator?*

Obviously, this question is specific to each application. In general, you should focus on the statistical independence of pseudo-random numbers and the even distribution of quasi-random numbers. Each property is desirable or undesirable in different applications based on their intended use. Consider both the changes to the application's internal behavior that would result from one or the other, as well as how that would manifest itself to the end user (if any).

For example, in a security-related application that uses random numbers, you most likely would prefer a pseudo-random number generator as it offers more unpredictability and so is less open to attack by malicious entities.

On the other hand, when using random numbers in physical simulations you may prefer the even distribution of quasi-random numbers to ensure that all dimensions of the problem are fully explored.

11 *Consider a massive vector addition done across multiple GPUs. Although you have not yet seen the CUDA APIs for multi-device execution, how do you think the multiple, separate address spaces affect tasks like memory allocation and data transfer? How would you*

*partition the work to be done in the vector addition across multiple GPUs? What kind of insight can this give you in to the implementation of multi-GPU libraries? How might you design functions like* `cufftXtMalloc` *and* `cufftXtMemcpy` *by combining* `cudaMalloc` *and* `cudaMemcpy` *with the ability to specify a target device for these CUDA operations?*

This is a speculative question simply intended to get you thinking, so it is understandable if you do not come to a conclusion on some of these points.

For memory allocation and data transfer, you generally divide a single memory operation evenly across multiple GPUs. For example, if you wanted to allocate a single array of length `N` on a platform with two GPUs, you would likely instead want to allocate an array of length `N/2` on each GPU. Data transfers would then also have to be partitioned into multiple transfers with a different array on a different GPU as a destination of each.

Partitioning work across multiple GPUs is the same as portioning work among multiple threads. However, when partitioning work across GPUs the granularity is much larger (that is, you don't assign a single element to a GPU) and generally a block partitioning is used (rather than cyclic, as you do with GPU threads).

While you do not have the source code for multi-GPU libraries, it is likely that they take a similar approach to splitting work across multiple GPUs by allocating partitions of memory on multiple GPUs using `cufftXtMalloc` and copying across multiple GPUs using `cufftXtMemcpy`. Then you would simply need to launch a kernel on each GPU for only the data placed there (more on how to do this in Chapter 9).

Pseudo-code implementations of modified `cufftXtMalloc` and `cufftXtMemcpy` are below. In general, they simply perform operations that you are already familiar with (`cudaMalloc` and `cudaMemcpy`) but iterate over GPUs while doing so.

```
void **cufftXtMalloc(nBytes)
{
    nGpus = ...
    void **dPtrs = (void **)malloc(sizeof(void *) * nGpus);
    spacePerGpu = (nBytes + nGpus - 1) / nGpus;
    for i = 0, i < nGpus, i++:
        switch to GPU i
        start = i * spacePerGpu
        end = (i + 1) * spacePerGpu
        if end > nBytes:
            end = nBytes
        CHECK(cudaMalloc(dPtrs + i, end - start));

    return dPtrs
}
```

```
void cufftXtMemcpyToDevice(void *host, void **device, N, direction)
{
    nGpus = ...
    spacePerGpu = (nBytes + nGpus - 1) / nGpus;
    for i = 0, i < nGpus, i++:
        switch to GPU i
        start = i * spacePerGpu
        end = (i + 1) * spacePerGpu
        if end > nBytes:
            end = nBytes
        if direction == cudaMemcpyHostToDevice:
            CHECK(cudaMemcpy(device[i], (char *)host + start, end - start,
                             cudaMemcpyHostToDevice))
        else:
            CHECK(cudaMemcpy((char *)host + start, device[i], end - start,
                             cudaMemcpyDeviceToHost))
}
```

*12 Define the following OpenACC terms: gang-redundant mode, gang-partitioned mode, worker-single mode, worker-partitioned mode.*

a. Gang-redundant mode: Each OpenACC gang has only a single active vector element in a single active worker. All active vector elements execute the same code on the same data.

b. Gang-partitioned mode: Like gang-redundant mode, each OpenACC gang has only a single active thread of execution but work is partitioned across gangs and so different gangs may execute different control flow on different data.

c. Worker-single mode: Only a single worker is active per gang.

d. Worker-partitioned mode: The work of a parallel region is divided both among multiple gangs and across multiple workers in each gang.

*13 Compare and contrast the `parallel` and `kernels` compiler directives in OpenACC. Be sure to discuss programmability and performance.*

While the `kernels` directive is automated and gives the compiler freedom to automatically parallelize a code region without placing that burden on the programmer, the `parallel` directive provides more programmer control over how code is parallelized.

In terms of programmability, the `parallel` directive certainly places more burden on the programmer to analyze their code and come up with a good parallelization strategy.

However, the `parallel` directive also gives you the power to manipulate the behavior of a parallel region at a much finer granularity, therefore making it possible that improved performance will be achieved relative to the `kernels` directive.

14 *How is the `loop` compiler directive used in OpenACC? Use it in the example below to achieve maximum parallelism when executing the loop.*

```
#pragma acc parallel
{
    for (i = 0; i < N; i++) {
        ...
    }
}
```

The `loop` compiler allows you to mark certain loops as parallelizable. In regions marked by the `parallel` directive, `loop` directives are required for the compiler to perform any parallelization. For `kernels` regions, the compiler may be able to auto-parallelize some loops without the `loop` directive, but the `loop` directive can explicitly mark parallelizable loops and provide more control over how that parallelization is performed.

The `loop` directive could be added to the above example as shown below.

```
#pragma acc parallel
{
#pragma acc loop
    for (i = 0; i < N; i++) {
        ...
    }
}
```

# Chapter 9

1 *Refer to the file `simpleMultiGPU.cu`. Use events to record GPU elapsed time and replace the code segment of CPU timer, and then compare the results.*

An example solution is provided in `simpleMultiGPUEvents.cu`. Your timing results should be nearly identical with both CUDA event timing and CPU timers.

2 *Run the previous executable with `nvprof`, as follows, and identify what is different in the operations performed for two devices versus one device.*

```
$ nvprof --print-gpu-trace ./simpleMultiGPU 1
```

Differences to note between the two GPU traces include:

- Twice as many `[CUDA memcpy HtoD]` and `[CUDA memcpy DtoH]` events for two devices as for a single device.
- Grid dimensions that are half as large when using two devices.

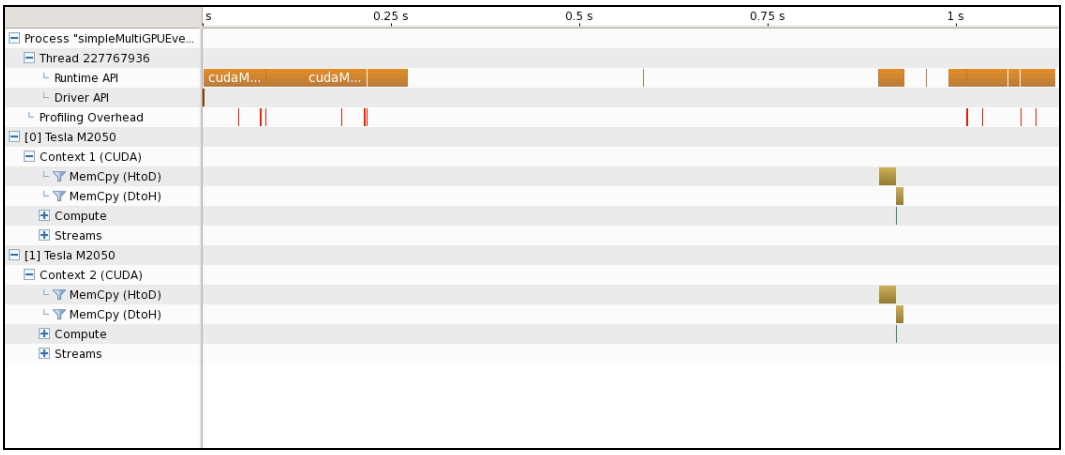
- Two kernel launches instead of a single kernel launch for two devices.
- The size of memory transfer operations halves when using two devices since each transfer now uses half as much data.
- The throughput of memory transfer operations decreases when using two devices. Since each transfer includes half as much data, the transfers are not able to use the host-device bandwidth as fully.
- Device information for each event in the GPU trace shows two different GPUs when running with two devices, and the context and stream identifiers also change with GPU devices.

3 *Run the previous executable with* `nvvp` *as follows:*

```
$ nvvp ./simpleMultiGPU
```

*Check the results in the Console Tab and Details Tab. Next, set the arguments to 1 in the Settings Tab to run the code on one GPU only. Compare the results with the two GPU case.*

Sample screen captures of the Timeline view, Details view, and Console view when using two devices are shown in that order below.

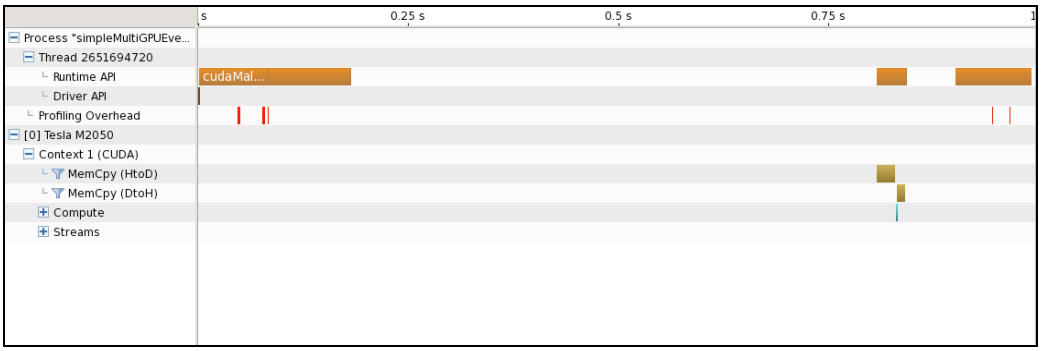


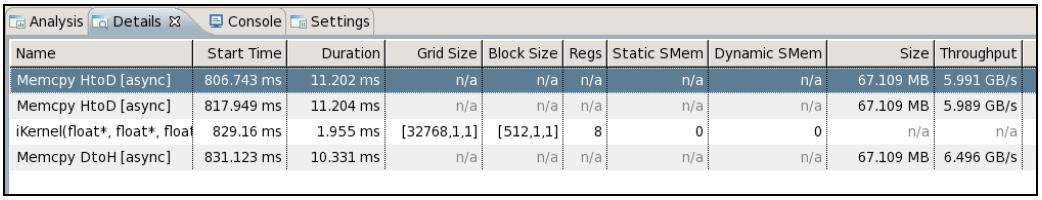

Analysis | Details | Console | Settings

| Name | Start Time | Duration | Grid Size | Block Size | Regs | Static SMem | Dynamic SMem | Size | Throughput |
|---|---|---|---|---|---|---|---|---|---|
| Memcpy HtoD [async] | 897.623 ms | 11.046 ms | n/a | n/a | n/a | n/a | n/a | 33.554 MB | 3.038 GB/s |
| Memcpy HtoD [async] | 897.708 ms | 10.974 ms | n/a | n/a | n/a | n/a | n/a | 33.554 MB | 3.058 GB/s |
| Memcpy HtoD [async] | 908.676 ms | 11.055 ms | n/a | n/a | n/a | n/a | n/a | 33.554 MB | 3.035 GB/s |
| Memcpy HtoD [async] | 908.69 ms | 10.982 ms | n/a | n/a | n/a | n/a | n/a | 33.554 MB | 3.055 GB/s |
| iKernel(float*, float*, float | 919.684 ms | 980.862 µs | [16384,1,1] | [512,1,1] | 8 | 0 | 0 | n/a | n/a |
| iKernel(float*, float*, float | 919.738 ms | 977.333 µs | [16384,1,1] | [512,1,1] | 8 | 0 | 0 | n/a | n/a |
| Memcpy DtoH [async] | 920.673 ms | 9.991 ms | n/a | n/a | n/a | n/a | n/a | 33.554 MB | 3.359 GB/s |
| Memcpy DtoH [async] | 920.724 ms | 9.964 ms | n/a | n/a | n/a | n/a | n/a | 33.554 MB | 3.367 GB/s |

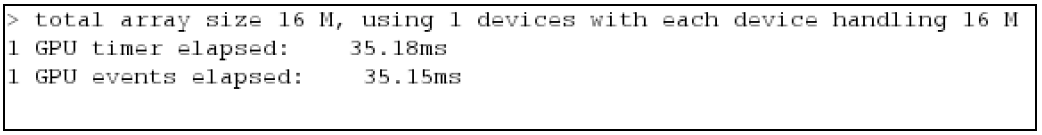
```
> total array size 16 M, using 2 devices with each device handling 8 M
2 GPU timer elapsed:    33.55ms
2 GPU events elapsed:    33.48ms
```

Sample screen captures of the Timeline view, Details view, and Console view when using one device are shown in that order below.




| Name | Start Time | Duration | Grid Size | Block Size | Regs | Static SMem | Dynamic SMem | Size | Throughput |
|---|---|---|---|---|---|---|---|---|---|
| Memcpy HtoD [async] | 806.743 ms | 11.202 ms | n/a | n/a | n/a | n/a | n/a | 67.109 MB | 5.991 GB/s |
| Memcpy HtoD [async] | 817.949 ms | 11.204 ms | n/a | n/a | n/a | n/a | n/a | 67.109 MB | 5.989 GB/s |
| iKernel(float*, float*, floa | 829.16 ms | 1.955 ms | [32768,1,1] | [512,1,1] | 8 | 0 | 0 | n/a | n/a |
| Memcpy DtoH [async] | 831.123 ms | 10.331 ms | n/a | n/a | n/a | n/a | n/a | 67.109 MB | 6.496 GB/s |




You can observe transfers and kernels running concurrently on multiple devices in the Timeline view. The Details view shows that the multi-GPU version runs twice as many copies and kernels as the single-GPU version, in order to distribute that work across multiple devices. The transfers are approximately half as large on the multi-device version, but kernel execution time does not drop by half due to kernel launch overheads, even though half as much data is being processed. On more realistic kernels, kernel time would demonstrate much more relative decrease when using multiple devices. Finally, the prints in the Console view confirm that the number of GPU timers and events being used changes to adapt to the number of devices.

4 *Put the following line at the end of the main loop in* `simpleMultiGPU`*:*

```
cudaStreamSynchronize(stream[i]);
```

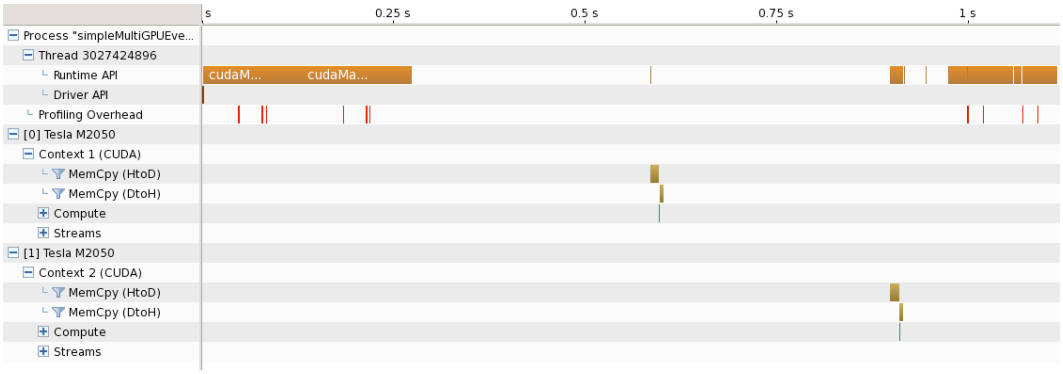
*Recompile the code and run it with* `nvvp` *for both cases of one GPU and two GPU, then compare the results with the code in Exercise 9.3 and explain the reason for the difference.*

By adding a `cudaStreamSynchronize` call in the job distribution loop, you explicitly synchronize on all work on the device in that stream before the host can continue execution. Your results may vary by GPU, but you should expect to see execution time go up by a factor of `ngpus`. There will no overlap of

any kind in the `nvvp` visualization, as no stream operations are allowed to execute in parallel.

5 *In `simpleMultiGPU.cu`, move data initialization (`initialData`) into the main kernel loop. Run it with `nvvp` to see what changes.*

An example implementation of this solution is provided in `simpleMultiGPU-initial.cu`. Running it with `nvvp` produces the timeline below. You can see the gap introduced by `initialData` between launching the transfers and kernel on the first device and on the second device. This demonstrates a small amount of overlap between the host computation in `initialData` for the second device, and the asynchronous copies and kernels for the first device. If the computation and communication for these kernels were larger, you could see the work on the first device completely overlap with `initialData` for the second device.




6 *Refer to the file `simpleP2P_PingPong.cu`. Modify the code section labeled `unidirectional gmem copy` to use asynchronous copies.*

An example implementation is included in an expanded `simpleP2P_PingPong.cu`. The relevant code snippet is included below. It uses `cudaMemcpyAsync` calls issued in the same stream to ping-pong the data from one GPU to another and back. `cudaEventSynchronize` is used at completion to wait for all asynchronous transfers to complete.

```
//  unidirectional asynchronous gmem copy
CHECK(cudaSetDevice(0));
CHECK(cudaEventRecord(start, 0));

for (int i = 0; i < 100; i++)
{
    if (i % 2 == 0)
    {
        CHECK(cudaMemcpyAsync(d_src[1], d_src[0], iBytes,
                    cudaMemcpyDeviceToDevice, stream[0]));
    }
    else
    {
        CHECK(cudaMemcpyAsync(d_src[0], d_src[1], iBytes,
                    cudaMemcpyDeviceToDevice, stream[0]));
```

```
        }
    }

    CHECK(cudaSetDevice(0));
    CHECK(cudaEventRecord(stop, 0));
    CHECK(cudaEventSynchronize(stop));
```

7 *Refer to the file* `simpleP2P_PingPong.cu`. *Add code that ping-pongs data between two GPUs based on the bidirectional, asynchronous ping-pong example provided. Use the function:*

```
cudaMemcpyPeerAsync(void* dst,int dstDev,const void* src,int srcDev,
                size_t count, cudaStream_t stream)
```

An example implementation is included in an expanded `simpleP2P_PingPong.cu`. `cudaMemcpyPeerAsync` is used to transfer data directly between GPUs. The relevant code snippet is included below.

```
    CHECK(cudaEventRecord(start, 0));

    for (int i = 0; i < 100; i++)
    {
        CHECK(cudaMemcpyPeerAsync(d_src[1], 1, d_src[0], 0, iBytes, stream[0]));
        CHECK(cudaMemcpyPeerAsync(d_rcv[0], 0, d_rcv[1], 1, iBytes, stream[1]));
    }

    CHECK(cudaSetDevice( 0  ));
    CHECK(cudaEventRecord(stop, 0));
    CHECK(cudaEventSynchronize(stop));
```

8 *Refer to the file* `simpleP2P_PingPong.cu`. *Use a default stream in the asynchronous memory copy runtime function. Compare the result with a non-default stream.*

A version of `simpleP2P_PingPong.cu` that replaces all explicitly created streams with the default stream is provided in `simpleP2P_PingPongDefault.cu`. Using only the default CUDA stream prevents multiple, bidirectional memory transfers from executing concurrently. While the hardware would support it, software constraints placed on execution of operations in the same CUDA stream prevent it. As a result, all unidirectional and bidirectional transfers in `simpleP2P_PingPingDefault.cu` will demonstrate similar achieved bandwidth.

9 *Refer to the file* `simpleP2P_PingPong.cu`. *Disable P2P access first, and then compare both results of unidirectional versus bidirectional memory copies, and synchronous versus asynchronous functions.*

Disabling P2P access can be accomplished by commenting out the following two lines in simpleP2P_PingPong.cu and recompiling.

```
// if (ngpus > 1) enableP2P(ngpus);
...
// disableP2P(ngpus);
```

Doing so means that direct transfers from one GPU to another over the PCIe bus are no longer supported, and all GPU-to-GPU transfers must use host memory as a staging area. As a result, the bandwidth achieved by all tests will drop no matter whether they are synchronous or asynchronous, unidirectional or bidirectional. You will also likely observe all transfers achieving approximately the same bandwidth as there can be no bi-directional transfers if all copies are going through host memory.

10 *Refer to the file* `simple2DFD.cu`*. Rearrange wave propagation with the following logic: (1) calculate halo on stream halo, (2) exchange halo on stream halo, (3) calculate internal on stream internal, and (4) synchronize each device. Compare the results with the original implementation and explain the reason for performance changes.*

An example implementation of the solution for this exercise is provided in `simple2DFDModified.cu`. The main modifications are 1) separating the halo computation and internal computation into separate multi-GPU loops, and 2) re-ordering the internal computation loop to happen after the halo exchange. In general, you should see a minor loss in performance with these transformations, caused by 1) less possible overlap between computation and communication, and 2) lower device utilization from fewer concurrent kernels on the GPU at once. For example, on an example platform the original version demonstrated a bandwidth of 737.91 MCells/s while the modified version only achieves 684.84 MCells/s.

11 *Explain how CPU and GPU affinity each impact application execution time. Suppose you ran a MPI-CUDA application twice, as follows:*

```
$ mpirun_rsh -np 2 node01 node01 MV2_ENABLE_AFFINITY=1 ./simplec2c
$ mpirun_rsh -np 2 node01 node01 MV2_ENABLE_AFFINITY=0 ./simplec2c
```

*Each command launches two MPI processes on one node, with or without CPU affinity. Describe the techniques you would use to establish GPU affinity for each MPI process in both scenarios, and explain why.*

CPU affinity forces certain CPU threads or CPU processes to run on a single core, preventing the operating system from moving threads and processes around between cores. This can benefit performance by reducing overheads associated with thread/process migration. For instance, if a CPU thread switches cores, all of the data it has already pulled into the L1 cache on the original core is likely not in the L1 cache on the new core. Hence, switching cores can incur more uncached accesses to system memory by CPU threads or processes in addition to other overheads.

GPU affinity prescribes which device is used by a given CPU thread. The goal of GPU affinity is to 1) limit the bus distance between the CPU core a CPU thread is executing on and the GPU device it is issuing commands to, and 2) ensure that CPU threads are evenly distributed among GPUs in a platform. CPU

cores and GPUs that are farther from each other will incur higher latencies over the PCIe bus, making data transfer consume more time. Oversubscribing a single GPU with many CPU threads leads to overhead from contention among the CPU threads for the GPU and may lead to CPU threads spending more time waiting for the GPU to become available while other GPUs idle. When setting GPU affinity, it is therefore necessary to balance both PCIe locality and GPU load balancing.

If the CPU affinity of an MPI process is already set when it starts, as in the command:

```
$ mpirun_rsh -np 2 node01 node01 MV2_ENABLE_AFFINITY=1 ./simplec2c
```

then the primary goal of GPU affinity is to ensure that each CPU thread has a GPU that is nearby. As the threads are already distributed evenly among CPU cores, you can expect that distributing GPUs by locality with CPU cores will result in an even distribution of threads among GPUs as well. Doing so requires the use of the `hwloc` package, an example of which is shown in Chapter 9.

However, if CPU affinity is not set for MPI processes, as in the command:

```
$ mpirun_rsh -np 2 node01 node01 MV2_ENABLE_AFFINITY=0 ./simplec2c
```

then the primary goal of GPU affinity should be to ensure threads are evenly distributed among GPUs, as locality cannot be guaranteed if the operating system is free to move threads around. For MVAPICH, you can use the `MV2_COMM_WORLD_LOCAL_RANK` environment variable to determine the rank of an MPI process within a single node and use that information to evenly distribute threads in the same node among GPUs in that node. Similar environment variables exist for other MPI implementations as well (for example, OpenMP has `OMPI_COMM_WORLD_LOCAL_RANK`). An example of using the local rank of an MPI process to assign GPU affinity is shown in Chapter 9.

*12 What is GPUDirect RDMA? How can it improve performance? Describe the three versions of GPUDirect. What are the hardware and software requirements for using GPUDirect RDMA?*

GPUDirect RDMA is a technology that allows direct communication of data between a GPU and arbitrary other PCIe devices on the same PCIe bus. One of the common use cases for this is performing transfers between GPUs in different nodes directly through Infiniband adapters attached to the same buses as those GPUs. In this situation, the transfer does not pass through CPU system memory but can go directly over the PCIe bus. This reduces overheads on the CPU by not requiring additional CPU memory to stage the transfer through or CPU management, and improves latency between GPUs in different nodes by reducing the distance travelled by the data.

There have been three iterations of GPUDirect that led to GPUDirect RDMA. The first iteration used a pinned host CPU buffer to allow direct transfer of data between GPUs in different nodes. The second iteration added P2P and

UVA, improving the programmability of GPUDirect in general and improving performance when performing GPU-to-GPU transfers within a single node. Finally, the third iteration implemented GPUDirect RDMA.

Using GPUDirect RDMA requires that the GPU and the PCIe device it interacts with share the same PCIe root. The GPU must also be a Kepler device, and the CUDA installation version must be sufficient to support P2P transfers (CUDA 5.5 or higher).

*13 How could you use MPI functions, `cudaMemcpyAsync`, and stream callbacks to build an asynchronous version of `simpleP2P.c`?*

simpleP2P.c sends messages between two GPUs attached to different MPI processes by synchronously copying the data from the source GPU, sending it to the destination process using `MPI_Isend`, receiving it on the destination process using `MPI_Irecv`, waiting for the transfer to complete using `MPI_Waitall`, and then transferring the received data to the GPU synchronously.

While this whole process is synchronous from the perspective of the host application, it could be made asynchronous using CUDA streams. This could be done with two primary steps: 1) converting the synchronous transfers to/from the GPU to use `cudaMemcpyAsync`, and 2) placing the MPI calls inside of a CUDA callback function and enqueuing that callback in the same stream and between the `cudaMemcpyAsync` calls. While implementing this is not required for this exercise, an example implementation is provided in `simpleP2P-async.c`. The main modifications from `simpleP2P.c` are the addition of the CUDA callback `mpi_callback` and a `struct` for passing parameters to it:

```
typedef struct _user_data {
    char *h_rcv, *h_src;
    int size;
    int other_proc;
    int recv_id, send_id;
    MPI_Request send_request, recv_request;
    MPI_Status reqstat;
} user_data;

void CUDART_CB mpi_callback(cudaStream_t stream, cudaError_t status, void *data)
{
    user_data *ctx = (user_data *)data;
    // bi-directional transmission
    MPI_Irecv(ctx->h_rcv, ctx->size, MPI_CHAR, ctx->other_proc, ctx->recv_id,
            MPI_COMM_WORLD, &(ctx->recv_request));
    MPI_Isend(ctx->h_src, ctx->size, MPI_CHAR, ctx->other_proc, ctx->send_id,
            MPI_COMM_WORLD, &(ctx->send_request));

    MPI_Waitall(1, &(ctx->recv_request), &(ctx->reqstat));
    MPI_Waitall(1, &(ctx->send_request), &(ctx->reqstat));
}
```

as well as enqueuing this callback using `cudaStreamAddCallback` in place of blocking MPI calls:

```
/*
 * Transfer data from the GPU to the host to be transmitted to
 * the other MPI process.
 */
CHECK(cudaMemcpyAsync(h_src, d_src, size,
        cudaMemcpyDeviceToHost, stream));

CHECK(cudaStreamAddCallback(stream, mpi_callback,
        &ctx, 0));
/*
 * Transfer the data received from the other MPI process to
 * the device.
 */
CHECK(cudaMemcpyAsync(d_rcv, h_rcv, size,
        cudaMemcpyHostToDevice, stream));
```

14 *Refer to the file `simpleP2P.c`. Change the pinned host memory to pageable host memory to see how the performance will change and explain the reason. If you cannot run it, describe what you would expect.*

A modified version of `simpleP2P.c` is available in `simpleP2P_Pageable.c`. Recall that using pinned memory to transfer data to and from the GPU improves transfer performance because the CUDA runtime can transfer directly from pinned memory to the GPU. Because the physical location of pageable memory can change while a transfer is being performed, using pageable memory adds overhead as the transfer is staged through a runtime-allocated pinned memory buffer. These concepts apply here as well. As a result, using pageable memory with `cudaMemcpy` adds performance overhead and an increase in execution time and loss in achieved bandwidth will be observed.

Note that using pinned host memory does not impact performance of the MPI part of the inter-GPU transfer in `simpleP2P_Pageable`. This performance increase is purely a result of CUDA's interactions with pinned and pageable memory.

15 *Consider `simpleP2P_CUDA_Aware.c`. For platforms without GPUDirect (that is, cannot directly transfer data between PCIe devices) how do you think `MPI_Isend` works when passed a device pointer?*

When passed a GPU pointer on platforms without GPUDirect RDMA, `MPI_Isend` likely uses a pinned buffer in host memory to transfer the contents of GPU memory to before sending it over the network to the destination node, where it is then temporarily stored in another pinned buffer before being copied to the destination GPU. This is essentially identical to the first iteration of GPUDirect. Using a pinned buffer with a limited size ensures good performance for the transfers while limiting interference with host application performance. It would also be possible for `MPI_Isend` to use multiple pinned buffers so as to overlap transfers to/from the GPU with transfers over the network.

16 *MVAPICH's CUDA-Aware MPI allows you to change the chunk size for copying data. Describe how chunk size might affect the internals of*

*CUDA-Aware MPI. Why do larger chunk sizes generally perform better?*

The chunk size set for CUDA-Aware MPI likely impacts the size of internal buffers used to temporarily store data being transferred to or from a GPU. This buffer size would of course dictate the granularity at which data is transferred to/from the GPU and over the interconnect between nodes.

Larger chunk sizes likely cause improved performance because they reduce overheads associated with both CUDA and MPI transfers and allow for better bus/interconnect utilization. Consider that in many of the examples in this book that have been studied using nvprof, larger amounts of data transferred to or from GPU global memory generally meant higher observed load and store throughput. The same rule applies to transfers over the PCIe bus and over the interconnect.

# Chapter 10

1 *Name the four stages of APOD and the goal of each.*
   a. Assess: The Assess stage aims to identify computational bottlenecks in an application and determine if GPUs can be used to accelerate their performance. If so, one or more strategies should be developed for porting these bottlenecks to CUDA.
   b. Parallelize: During the Parallelize stage, bottlenecks identified in the Assess stage are ported to CUDA. The goal of this stage is to simply get a working CUDA implementation; performance may be suboptimal. The Parallelize stage may use hand-coded CUDA kernels, CUDA libraries, or OpenACC to implement the parallelization strategies developed as a part of the Assess stage.
   c. Optimize: In the Optimize stage, the initial CUDA port developed in the Parallelize stage is optimized using CUDA profiling tools and optimization techniques. Profile-driven development should be used to identify bottlenecks in your CUDA implementation and improve their performance.
   d. Deploy: In the Deploy stage, the CUDA application is readied for deployment to the production environment. This stage can imply a number of steps and is more customized on a case-by-case basis. For instance, if your applications run on multi-GPU platforms, the Deploy stage would be the time to ensure that they are sufficiently flexible to take advantage of all hardware resources.
2 *An application can be accelerated using CUDA libraries, OpenACC, or hand-coded CUDA kernels. In which APOD stage should the decision*

*be made between the two approaches? What are the main differences you envision being introduced to each stage of APOD depending on the approach chosen?*

The decision of which technology to use when accelerating a legacy application with CUDA should be made during the Assess stage. This decision should be made based on the characteristics of the bottlenecks discovered and how amenable they are to each CUDA technology. For example, if an application is bottlenecked by computation that is already implemented in a CUDA library (such as linear algebra operations) then using CUDA libraries to port the application will likely result in an accelerated porting process and better performance. However, if the application relies on highly-customized logic then your only choice may be implementing CUDA kernels by hand.

While the Assess stage determines which technology to use, each of the following stages is affected by this decision.

In Parallelize, the choice of whether to port using CUDA libraries, OpenACC, or hand-coded CUDA obviously has a major impact on the steps taken and how parallelization is performed.

In Optimize, different technologies enable or limit the possible optimizations. CUDA libraries are the most restrictive, many of them are blackboxes that you have little control over. OpenACC provides more flexibility in changing execution configurations, modifying memory accesses, and other common optimization efforts. Hand-coded CUDA is the most tunable but as a result requires the most effort to optimize.

In Deploy, making each technology production-ready requires different steps. CUDA libraries may require ensuring that any libraries used by the application are accessible from all environments the application may run in. All three options expose multi-GPU execution differently and so require different work to prepare an application to adapt to arbitrary platforms.

3 *What capabilities are added by separate compilation in CUDA 5? What compiler flags must be added to use separate compilation?*

Separate compilation allows you to:

- Reduce build times using incremental library recompilation
- Increase code reuse
- Combine object files into static libraries
- Link and call external device code
- Create and use third-party libraries

In general, separate compilation makes CUDA compilation more flexible and similar to C/C++ compilation. To enable separate compilation, you need CUDA 5

or newer. To compile multiple `.cu` files separately into a single executable requires first building re-locatable objects for each `.cu` file:

```
$ nvcc -dc foo.cu -o foo.o
$ nvcc -dc bar.cu -o bar.o
```

and then linking those objects together into a single output object file using `nvcc`:

```
$ nvcc -dlink foo.o bar.o -o link.o
```

The final output object file can then be linked into a host application normally:

```
$ g++ host.o link.o -L$CUDA_HOME -lcudart
```

*4 Which tool is best used to analyze out-of-bounds accesses in a kernel? Why?*

When analyzing an application for out-of-bounds accesses on the GPU you should use the `memcheck` subtool of `cuda-memcheck`. It analyzes all I/O performed by kernels for invalid operations and reports those results to the user. `memcheck` enables quick and simple automated instrumentation of CUDA kernels for a wide range of memory errors.

*5 What tool is best used to analyze `__shared__` memory usage?*

For analyzing shared memory usage, use the `racecheck` subtool of `cuda-memcheck`. `racecheck` targets shared memory in particular, and makes invalid accesses to shared memory much easier to identify than attempting to do so manually.

*6 What are the four modes of analysis in `nvprof`? What information is each best at gathering?*

`nvprof`'s four modes are: Summary mode, Trace mode, Event/Metric Summary mode, and Event/Metric Trace mode.

Summary mode provides a quick view of application behavior and is good at providing a high-level understanding of where time is being spent. It does little to explain the actual reasons for any bottlenecks in an application, but can be useful in identifying them.

Trace mode allows you to quickly view a timeline of operations in your application. Trace mode can be helpful in debugging unexpected application behavior as it shows the operations being performed by your application at a course granularity that is easily understood. For example, if a `cudaMemcpy` is missing and leading to uninitialized state on the GPU, Trace mode would be the best way to find it.

The Event/Metric Summary and Trace modes are more useful when trying to gain insight into very low-level behaviors and characteristics of an application by studying specific metrics about different kernels' performance on hardware.

These modes are useful after having already identified a bottleneck, and can be used to explain poor kernel performance as well as validate that optimizations are working as expected.

7 *What are the benefits of using `nvvp` versus another profiling tool?*

Because `nvvp` is GUI-based, it is useful for high-level visualization of an application's execution timeline. This visualization capability can be useful for determining the most promising high-level optimizations, such as improving computation and communication overlap.

`nvvp` combines the high-level GUI view with detailed performance metrics like the information you would get from `nvprof`, meaning that you can intuitively visualize lost performance and immediately explain it with low-level metrics.

`nvvp` also automatically analyzes gathered metrics on your kernels and automatically suggests areas to target optimizations that would yield improved performance. In general, it is a more high-level profiling tool compared to `nvprof`.

8 *Consider your day-to-day development environment. How would `nvprof` and `nvvp` best fit in it? For example, if you regularly work with a remote machine containing GPUs on the same LAN as your local workstation, you might use `nvprof` to gather profiling dumps on the remote machine, transfer them to your local workstation, and analyze them using `nvvp`.*

This question is specific to your particular development environment and platform, and is simply intended to ensure that you think about how to best use these tools. As you gain more experience in CUDA optimization strategies, `nvprof` will quickly become the most efficient tool to use. It offers low overhead performance profiling at a fine granularity and does not require a heavyweight GUI to operate. However, using `nvvp` can be more efficient and helpful while you are still learning about CUDA optimization and what works best in different situations. Because `nvvp` is a comprehensive tool that attempts to analyze your application from every direction and on every metric, it is also easier to use and provides more confidence in the fact that all optimization opportunities have been explored.