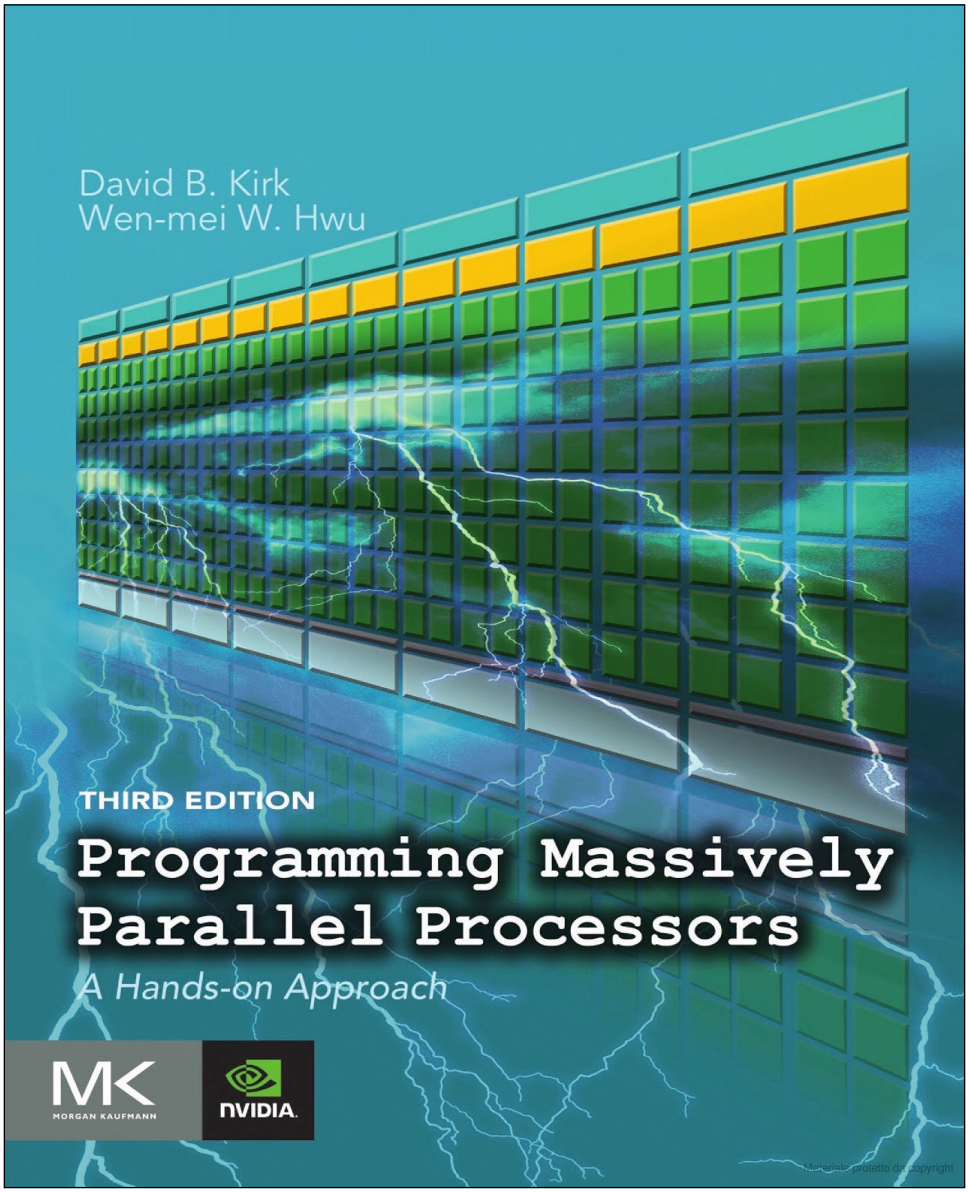
Programming Massively Parallel Processors (3rd edition)

Exercises and solutions



Contents

Chapter 2

Exercise 2.1

If we want to use each thread to calculate one output element of a vector addiction, what would be the expression for mapping the thread/block to data index?

Solution:

C. `i=blockIdx.x*blockDim.x+threadIdx.x`

Exercise 2.2

Assume that we want to use each thread to calculate two (adjacent) elects of a vector addiction. What would be the expression for mapping the thread/block indices to i, the data of the first element to be processed by a thread.

Solution:

C. `i=(blockIdx.x*blockDim.x + threadIdx.x)*2`

Every thread covers 2 consecutive elements. The starting data index is simply twice the global thread index. Another way to look at it is that all previous blocks cover (blockIdx.x*blockDim.x)*2. Within the block, each thread covers 2 consecutive elements so the beginning position for a thread is 2*threadIdx.x.

Exercise 2.3

We want to use each thread to calculate two elements of a vector addition. Each thread block processes 2*blockDim.x consecutive elements that form two sections.All threads in each block will first process a section first, each processing one element. They will then all move to the next section, each processing one element. Assume that variable i should be the index for the first element to be processed by a thread. What would be the expression for mapping the thread/block indices to data index of the first element?

Solution:

D. `i = blockIdx.x * blockDim.x * 2 + threadIdx.x`

For instance if we have vectors of length 4, and block of size 2, the first index of every thread/block will be the follow:

```
Block 0:    i=0*2*2 + 0=0          Block 1:    i=1*2*2 + 0=4
            i=0*2*2 + 1=1                      i=1*2*2 + 1=5
```

Exercise 2.4

For a vector addition. assume that the vector length is 8000, each thread calculates one output element, and the thread block size is 1024 threads. The programmer configures the kernel launch to have a minimal number of thread blocks to cover all output elements. How many threads will be in the grid?

Solution:

C. `Sup(8000/1024)*1024=8192 threads in the grid`

Exercise 2.5

If we want to allocate an array of v integer elements in CUDA device global memory, what would be an appropriate expression for the second argument of the cudaMalloc() call?

Solution:

D. `v*sizeof(int)`

Exercise 2.6

If we want to allocate an array of n floating-point elements and have a floating-point pointer variable d_A to point to the allocated memory, what would be an appropriate expression for the first argument of the cudaMalloc() call?

Solution:

D. `(void**) &d_A`

Exercise 2.7

If we want to copy 3000 bytes of data from host array h_A (h_A is a pointer to element 0 of the source array) to device array d_A (d_A is a pointer to element 0 of the destination array). what would be an appropriate API call for this data copy in CUDA?

Solution:

C. `cudaMemcpy(d_A, h_A, 3000, cudaMemcpyHostToDevice);`

Exercise 2.8

How would one declare a variable err that can appropriately receive returned value of a CUDA API call?

Solution:

C. `cudaError_t err;`

Exercise 2.9

A new summer intern was frustrated with CUDA. He has been complaining that CUDA is very tedious: he had to declare many functions that he plans to execute on both the host and the device twice, once as a host function and once as a device function. What is your response?

Solution:

Make a function with `__device__` and `__host__` keywords above it.

Chapter 3

Exercise 3.1

A matrix addition takes two input matrices A and B and produces one output matrix C.
Each element of the output matrix C is the sum of the corresponding elements of the input matrices A and B, i.e., C[i][j] = A[i][j] + B[i][j].
For simplicity, we will only handle square matrices whose elements are single-precision floating-point numbers. Write a matrix addition kernel and the host stub function that can be called with four parameters: pointer-to-the-output matrix, pointer-to-the-first-input matrix, pointer-to-the-second-input matrix, and the number of elements in each dimension. Follow the instructions below:

A. Write the host stub function by allocating memory for the input and output matrices, transferring input data to device; launch the kernel, transferring the output data to host and freeing the device memory for the input and output data. Leave the execution configuration parameters open for this step.

B. Write a kernel that has each thread to produce one output matrix element. Fill in the execution configuration parameters for this design.

C. Write a kernel that has each thread to produce one output matrix row. Fill in the execution configuration parameters for the design.

D. Write a kernel that has each thread to produce one output matrix column. Fill in the execution configuration parameters for the design.

E. Analyze the pros and cons of each kernel design above.

Solution:

A.

```
void matrixAdd(float *h_Mout, float *h_Min1, float *h_Min2, int rows, int
cols) {
    float *d_Mout, *d_Min1, *d_Min2;
    int size = rows*cols *sizeof(float);

    CHECK_ERROR(cudaMalloc((void**)&d_Mout, size));
    CHECK_ERROR(cudaMalloc((void**)&d_Min1, size));
    CHECK_ERROR(cudaMalloc((void**)&d_Min2, size));

    cudaMemcpy(d_Min1, h_Min1, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_Min2, h_Min2, size, cudaMemcpyHostToDevice);

    dim3 dimGrid(…);
    dim3 dimBlock(…);

    matrixAddKernel<<<dimGrid,dimBlock>>>(d_Mout,d_Min1,d_Min2,rows,cols);

    cudaMemcpy(h_Mout, d_Mout, size, cudaMemcpyDeviceToHost);

    cudaFree(d_Mout); cudaFree(d_Min1); cudaFree(d_Min2);
}
```

B.

```
__global__
void matrixAddKernel(float *d_Mout, float *d_Min1, float *d_Min2, int rows,
int cols) {
     int i = blockIdx.y * blockDim.y + threadIdx.y;
     int j = blockIdx.x * blockDim.x + threadIdx.x;

     // each thread produce one output matrix element
     if (i < rows && j < cols) {
       d_Mout[i*rows + j] = d_Min1[i*rows + j] + d_Min2[i*rows + j];
     }
}
// configuration parameters:
     dim3 dimGrid(ceil(rows / 4.0), ceil(cols / 4.0), 1);
     dim3 dimBlock(4.0, 4.0, 1);
```

C.

```
__global__
void matrixAddKernel(float *d_Mout, float *d_Min1, float *d_Min2, int rows,
int cols) {
     int j = blockIdx.x * blockDim.x + threadIdx.x;

     // each thread produce one output matrix row
     if (j < rows) {
       int i = 0;
       while (i < cols) {
             d_Mout[j*rows + i] = d_Min1[j*rows + i] + d_Min2[j*rows + i];
             i++;
       }
     }
}
// configuration parameters:
     dim3 dimGrid(ceil(rows / 4.0), 1, 1);
     dim3 dimBlock(4.0, 1, 1);
```

D.

```
__global__
void matrixAddKernel(float *d_Mout, float *d_Min1, float *d_Min2, int rows,
int cols) {
     int i = blockIdx.y * blockDim.y + threadIdx.y;

     // each thread produce one output matrix column (SOLUTION D)
     if (i < cols) {
       int j = 0;
       while (j < rows) {
             d_Mout[j*rows + i] = d_Min1[j*rows + i] + d_Min2[j*rows + i];
             j++;
       }
     }
}
// configuration parameters:
     dim3 dimGrid(1, ceil(rows / 4.0), 1);
     dim3 dimBlock(1, 4.0, 1);
```

The entire source code is located in /cudaExercise/chapter03/matrixAdd.cu
For this exercise I built two 100x100 square matrices, so the configuration parameters chosen are optimized to reduce the number of extra threads in this specific case, but they are not optimized for the hardware efficiency because the number of threads in each dimension of thread blocks should be multiplies of 32. Furthermore, we can see that in kernel written in solution B, the program take more less time than kernels in solutions C and D to perform the addictions because the calculus is parallelized, instead in kernels C and D the calculus is semi-parallelized because part of computation is executed in a while loop by the same thread.

Exercise 3.2

A matrix—vector multiplication takes an input matrix B and a vector C and produces one output vector A. Each element of the output vector A is the dot product of one row of the input matrix B and C, i.e.,A[i] = $\sum_j$ B[i][j] + C[j]. For simplicity, we will only handle square matrices whose elements are single-precision floating-point numbers. Write a matrix—vector multiplication kernel and a host stub function that can be called with four parameters: pointer-to-the-output matrix, pointer-to-the-input matrix, pointer-to-the-input vector, and the number of elements in each dimension. Use one thread to calculate an output vector element.

Solution:

```
void matrixVecMulKernel(float *d_Vout, float *d_M, float *d_V, int size) {
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    int j = blockIdx.x * blockDim.x + threadIdx.x;

    // each thread produce one output matrix element
    if (i < size && j < size) {
        float sum = 0;
        for (int k = 0; k < size; k++) {
            sum += d_M[i*size + k] * d_V[k];
        }
        d_Vout[i] = sum;
    }
}

void matrixVecMul(float *h_M, float *h_V, float *h_Vout, int size) {
    float *d_Vout, *d_M, d_V;

    CHECK_ERROR(cudaMalloc((void**)&d_Vout, size*sizeof(float)));
    CHECK_ERROR(cudaMalloc((void**)&d_M, size*size*sizeof(float)));
    CHECK_ERROR(cudaMalloc((void**)&d_V, size*sizeof(float)));

    cudaMemcpy(d_M, h_M, size*size*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_V, h_V, size*sizeof(float), cudaMemcpyHostToDevice);

    dim3 dimGrid(ceil(size / 32.0), ceil(size / 32.0), 1);
    dim3 dimBlock(32.0, 32.0, 1);

    matrixVecMulKernel<<<dimGrid, dimBlock>>>(d_Vout, d_M, d_V, size);

    cudaMemcpy(h_Vout, d_Vout, size*sizeof(float), cudaMemcpyDeviceToHost);

    cudaFree(d_Vout);
    cudaFree(d_M);
    cudaFree(d_V);
}
```

Exercise 3.3

If the SM of a CUDA device can take up to 1536 threads and up to 4 thread blocks. Which of the following block configuration would result in the largest number of threads in the SM?

Solution:

B. 256 thread per block
Because this CUDA device allows up to 1536/4 = 384 thread per block.

Exercise 3.4

For a vector addiction, assume that the vector length is 2000, each thread calculates one output element, and the thread block size is 512 threads. How may threads will be in the grid?

Solution:

D. 2048
sup(2000 threads / 512 thread per block) = 4 block. So there are 4*512 = 2048 threads in the grid.

Exercise 3.5

With reference to the previous question, how many wraps do you expect to have divergence due to the boundary check on vector length?

Solution:

B. 2
2000 - 2048 = 48 threads that corresponds to 1 wrap (32 threads) and 16 threads + 16 padded threads to fill the second wrap.

Exercise 3.6

You need to write a kernel that operates on an image of size 400x 900 pixels. You would like to assign one thread to each pixel. You would like your thread blocks to be square and to use the maximum number of threads per block possible on the device (your device has a compute capability 3.0). How would you select the grid dimensions and block dimensions of your kernel?

Solution:

We consider the follow table of compute capability:

Technical specifications of compute capability of 3.0 device	
16	Maximum number of resident blocks per multiprocessor
2048	Maximum number of resident threads per multiprocessor
1024	Maximum number of threads per block

We can verify that with 32 x 32 block size we'll need up to 2048/1024 = 2 blocks in the SM.
This number is within the 16 block limitation. So we could use the follow configuration parameters:

```
dim3 dimGrid(ceil(400 / 32.0), ceil(900 / 32.0), 1);
dim3 dimBlock(32.0, 32.0, 1);
```

As result we'll generate 13*29 = 377 blocks, and 377*1024 = 386048 threads.

Exercise 3.7

With reference to the previous question, how many idle threads do you expect to have?

Solution:

We have generated 386048 threads and the number of pixel to perform are 360000, so we expect to have 386048 - 360000 = 26048 idle threads.

Exercite 3.8

Consider a hypothetical block with 8 threads executing a section of code before reaching a barrier. The threads require the following amount of time (in microseconds) to execute the sections: 2.0, 2.3, 3.0, 2.8, 2.4, 1.9, 2.6, and 2.9 and to spend the rest of their time waiting for the barrier. What percentage of the total execution time of the thread is spent waiting for the barrier?

Solution:

The maximum execution time is 3.0 microseconds, so all others threads will have to wait this thread.
We consider the follow table:

Thread number	execution time	waiting time
1	2.0	1.0
2	2.3	0.7
3	3.0	0.0
4	2.8	0.2
5	2.4	0.6
6	1.9	1.1
7	2.6	0.4
8	2.9	0.1

The percentage of the total execution time of the thread is spent waiting for the barrier is calculate as:

$$\frac{\sum_{i=1}^{8} waiting\_time\_of\_thread_i}{\sum_{i=1}^{8} execution\_time\_of\_thread_i} * 100 = 20,6\%$$

Exercise 3.9

Indicate which of the following assignments per multiprocessor is possible. In the case where it is not possible, indicate the limiting factor(s).

A. 8 blocks with 128 threads each on a device with compute capability 1.0
B. 8 blocks with 128 threads each on a device with compute capability 1.2
C. 8 blocks with 128 threads each on a device with compute capability 3.0
D. 16 blocks with 64 threads each on a device with compute capability 1.0
E. 16 blocks with 64 threads each on a device with compute capability 1.2
F. 16 blocks with 64 threads each on a device with compute capability 3.0

Solution:

A. No, maximum 768 threads in the SM
B. Yes
C. Yes
D. No, maximum 8 blocks and up to 768 threads in the SM
E. No, maximum 8 blocks
F. Yes

Exercise 3.10

A CUDA programmer says that if they launch a kernel with only 32 threads in each block, they can leave out the __syncthreads( ) instruction wherever barrier synchronization is needed. Do you think this is a good idea? Explain.

Solution:

When a thread calls __syncthreads( ), it will be held at the calling location until every thread in the block reaches the location. This process ensure that all threads in a block have completed a phase of their execution of the kernel before any of them can proceed to the next phase. If, as said in the question, the barrier is needed, the programmer can't omitted it, because, for example, if a thread in the next phase try to execute an instruction where it use a variable that is not still properly set, by a slower thread, the execution can fails or the output of the program may be wrong.

Exercise 3.11

A student mentioned that he was able to multiply two 1024 x 1024 matrices by using a tiled matrix multiplication code with 32 X 32 thread blocks. He is using a CUDA device that allows up to 512 threads per block and up to 8 blocks per SM. He further mentioned that each thread in a thread block calculates one element of the result matrix. What would be your reaction and why?

Solution:

We can see that he used the following configuration parameters:

```
dim3 dimGrid(32.0, 32.0, 1);
dim3 dimBlock(32.0, 32.0, 1);
```

We note that he used 1024 size of thread blocks, but the device that he used allows up to 512 thread per block, so it is impossible that he calculated the matrix multiplication.
We can propose the following configuration parameters instead:

```
dim3 dimGrid(64.0, 64.0, 1);
dim3 dimBlock(16.0, 16.0, 1);
```

With the configuration parameters we used 256 size of thread blocks, this number is in range of limitation of the device that he used.To verify additional considerations about the performances, we would have to know how many threads the device can host.

Chapter 4

Exercise 4.1

Consider matrix addition. Can one use shared memory to reduce the global memory bandwidth consumption? Hint: Analyze the elements accessed by each thread and see if there is any commonality between threads.

Solution:

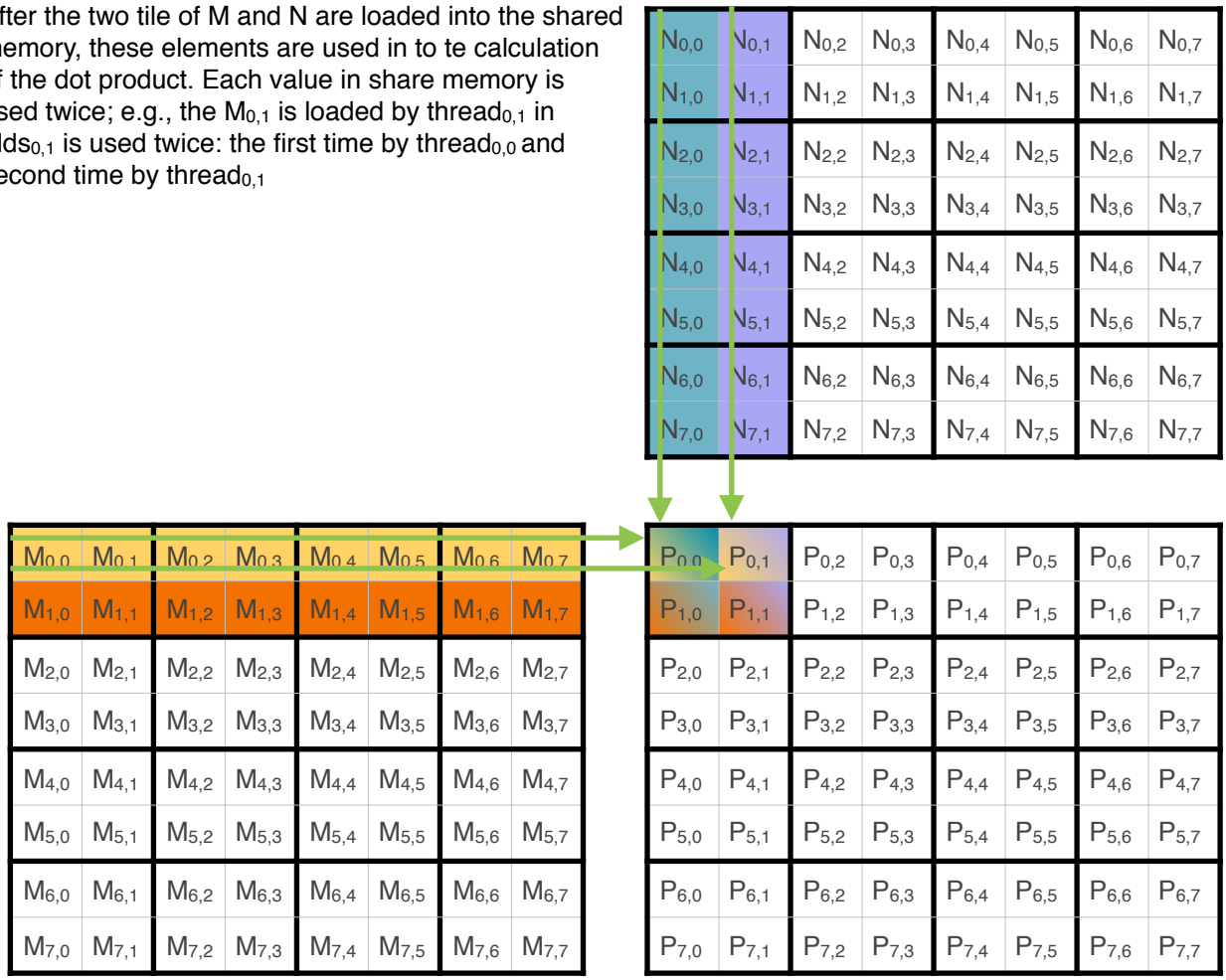
No because each thread accesses to an element of the matrices one time only.

Exercise 4.2

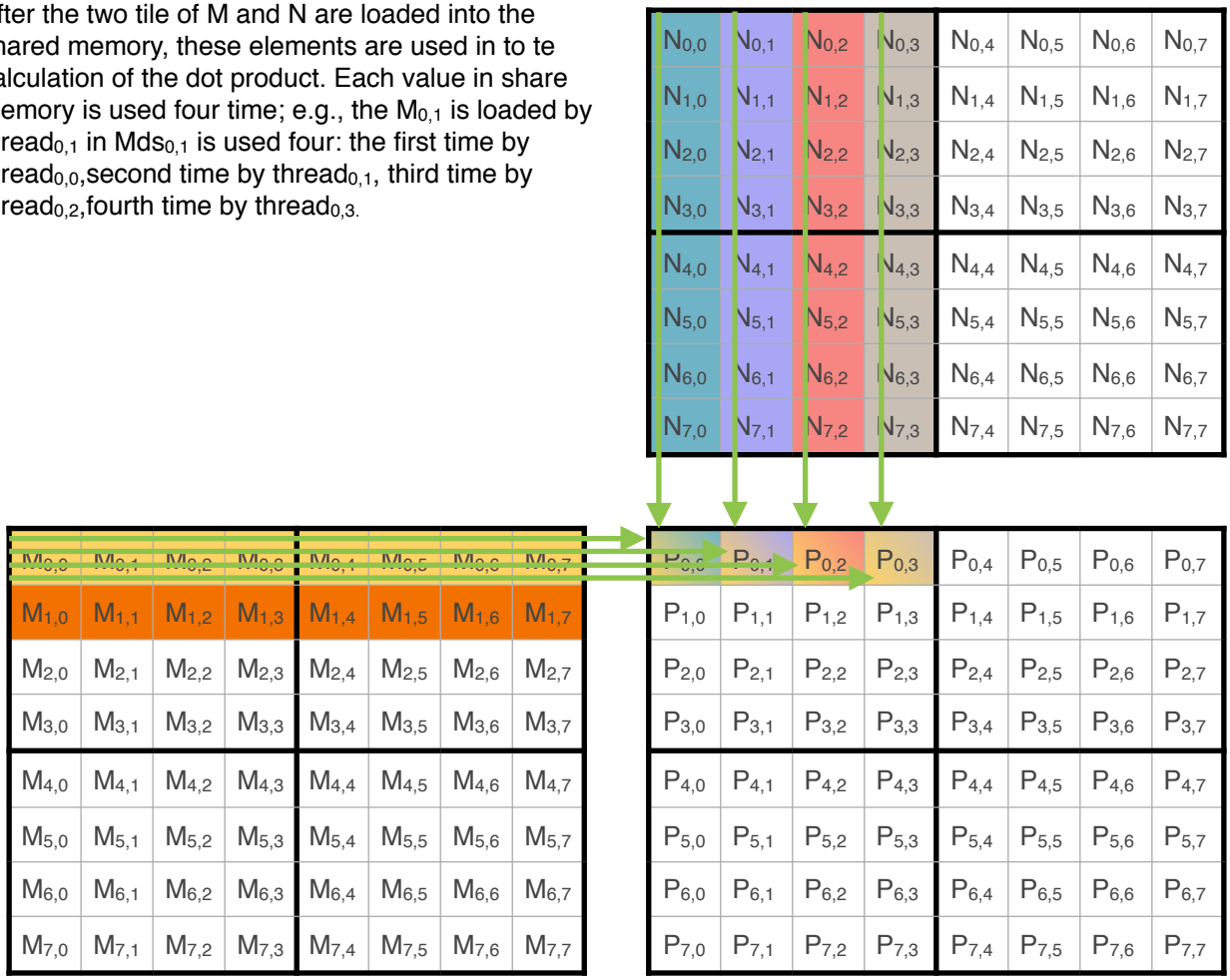
Draw the equivalent of Fig. 4.14 for an 8x 8 matrix multiplication with 2 x 2 tiling and 4 x 4 tiling. Verify that the reduction in global memory bandwidth is indeed proportional to the dimensions of the tiles.

Solution:

After the two tile of M and N are loaded into the shared memory, these elements are used in to te calculation of the dot product. Each value in share memory is used twice; e.g., the $M_{0,1}$ is loaded by $thread_{0,1}$ in $Mds_{0,1}$ is used twice: the first time by $thread_{0,0}$ and second time by $thread_{0,1}$



After the two tile of M and N are loaded into the shared memory, these elements are used in to te calculation of the dot product. Each value in share memory is used four time; e.g., the $M_{0,1}$ is loaded by $thread_{0,1}$ in $Mds_{0,1}$ is used four: the first time by $thread_{0,0}$,second time by $thread_{0,1}$, third time by $thread_{0,2}$,fourth time by $thread_{0,3}$.



Exercise 4.3

What type of incorrect execution behavior can happen if one or both __syncthreads() are omitted in the kernel of Fig. 4.16?

Solution:

if the first __syncthreads() is omitted in the kernel of Fig. 4.16, could happen that one or more thread calculate a wrong pValue, because of one or more threads haven't updated the shared variables yet. So the first synchronization make sure that all elements of a tile are loaded at the beginning on the phase, so all thread are ready to start.
if the second __syncthreads() is omitted, could happen that a wrong pValue is written (only half) in d_P matrix, because of one or more threads haven't increase the pValue variable yet.So the second synchronization make sure that all elements of a tile are consumed at the end of the phase.

Exercise 4.4

Assuming that capacity is not an issue for registers or shared memory, give one important reason why it would be valuable to use shared memory instead of registers to hold values fetched from global memory? Explain your answer.

Solution:

Using registers instead of shared memory,we aren't able to reduce the number of accesses to load the values from global memory, because of the scope of registers is within the single thread, and in case of matrix multiplication we need to access to the same data more than one time for each thread.

Exercise 4.5

For our tiled matrix—matrix multiplication kernel, if we use a 32x32 tile. what is the reduction of memory bandwidth usage for input matrices M and N?

A. 1/8 of the original usage
B. 1/16 of the original usage
C. 1/32 of the original usage
D. 1/64 of the original usage

Solution:

A 1/8
With 32x32 tile size we have only 1024 threads, thus only 1 thread block instead the 8 thread blocks with 16x16 tile size.

Exercise 4.6

Assume that a CUDA kernel is launched with 1,000 thread blocks, with each having 512 threads. If a variable is declared as a local variable in the kernel, how many versions of the variable will be created through the lifetime of the execution of the kernel?

A. 1
B. 1000
C. 512
D. 512000

Solution:

D 512000
1000blocks * 512threads per block because each thread has its private variable.

Exercise 4.7

In the previous question, if a variable is declared as a shared memory variable, how many versions of the variable will be created throughout the lifetime of the execution of the kernel?

A. 1
B. 1000
C. 512
D. 51200

Solution:

B 1000
Because each thread of a block share the same variable

Exercise 4.8

Consider performing a matrix multiplication of two input matrices with dimensions N x N. How many times is each element in the input matrices requested from global memory in the following Situations?

A. There is no tiling.
B. Tiles of size T x T are used.

Solution:

In case there is no tiling each elements will be requested N times from global memory.
In case there is tile of size T x T, each elements will be requested N/2 times from global memory.

Exercise 4.9

A kernel performs 36 floating-point operations and 7 32-bit word float memory accesses per thread. For each of the following device properties, indicate whether this kernel is compute- or memory-bound.

A. Peak FLOPS: 200 GFLOPS, Peak Memory Bandwidth: 100 GB/s
B. Peak FLOPS: 300 GFLOPS, Peak Memory Bandwidth: 250 GB/s

Solution:

$$(A)\quad \frac{28B/thread}{1\cdot 10^{11}B/s} = 2.8\cdot 10^{-10} \qquad \frac{36FLOPS\ /\ thread}{2\cdot 10^{11}} = 1.8\cdot 10^{-10}$$

$$(B)\quad \frac{28B/thread}{2.5\cdot 10^{11}B/s} = 1.12\cdot 10^{-10} \qquad \frac{36FLOPS\ /\ thread}{3\cdot 10^{11}} = 1.2\cdot 10^{-10}$$

As we can se, using device A, the kernel is memory-bound, because it spends most of time in executing memory instructions, and using device B the kernel is compute-bound, because it spend most of time in ALU-FPU operations.

Exercise 4.10

To manipulate tiles, a new CUBA programmer has written the following device kernel, which will transpose each tile in a matrix. The tiles are of size BLOC K_wI DTH by BLOC K)“ DTH, and each of the dimensions of matrix A is known to be a multiple of BLOCK_WIDTH. The kernel invocation and code are shown below. BLOCK_WIDTH is known at compile time, but could be set anywhere from I to 20.

```
1  dim3 blockDim(BLOCK_WIDTH,BLOCK_WIDTH);
2  dim3 gridDim(A_width/blockDim.x,A_height/blockDim.y);
3  BlockTranspose<<<gridDim, blockDim>>>(A, A_width, A_height);
4  __global__ void
5  BlockTranspose(float* A_elements, int A_width, int A__height)
6  {
7     __shared__ fl oat blockA[BLOCK_WIDTH][BLOCK_WIDTH]:
8     int baseIdx=blockIdx.x * BLOCK_SIZE + threadIdx.x;
9     baseIdx += (blockIdx.y * BLOCK_SIZE + threadIdx.y) * A__width;
10    blockA[threadIdx.y][threadIdx.x]=A_elements[baseIdx];
11    A_elements[baseIdx]=blockA[threadIdx.x][threadIdx.y];
12 }
```

A. Out of the possible range of values for BLOCK_SIZE, for what values 0 BLOCK_SIZE will this kernel function execute correctly on the device?

B. If the code does not execute correctly for all BLOCK_SIZE values, suggest a fix to the code to make it work for all BLOCK_SIZE values.

Solution:

The true problem is not the BLOCK_SIZE values, but it’s about the access to the shared memory values without any synchronization keywords! To fix it I suggest the follow correction:

```
10    blockA[threadIdx.y][threadIdx.x]=A_elements[baseIdx];
11    __sychthreads();
12    A_elements[baseIdx]=blockA[threadIdx.x][threadIdx.y];
```

Chapter 5

Exercise 5.1

The kernels in Figs. 5.13 and 5.15 are wasteful in their use of threads; half of the threads in each block never execute. Modify the kernel to eliminate such waste. Give the relevant execute configuration parameters values at the kernel launch. Is there a cost in terms of extra arithmetic operation needed ? Which resource limitation can be potentially addressed with such modification?
(Hint: (1) Line 2 and/or Line 3 can be adjusted in each case. (2) The number of elements in a section may need to increase.)

Solution(a.1):

See the source code in /cudaExercise/chapter05/reductionOptimized_v1.cu

```
__shared__ int partialSum[2 * BLOCK_SIZE]; // <- increase shared-memory space
int tx = threadIdx.x;
int i = blockIdx.x * blockDim.x + tx;
partialSum[tx] = X[i];
partialSum[tx + blockDim.x] = 0;

for (int stride = blockDim.x; stride > 0;stride/=2) //<- start from blockDim.x
{
     __syncthreads();
     if (tx <= stride)
       partialSum[tx] += partialSum[tx + stride];
}

// Launch a kernel on the GPU with one thread for each element.
partialSumKernel <<< 1, BLOCK_SIZE >>>(d_X, size);
```

For convenience, in this kernel code, we assumed that the number of elements is equal to number of threads, so we have one block of N threads (where n is also the number of elements).
As we can see, in this kernel code we doubled the dimension of shared memory and padded the extra space si zeros. Starting the loop from blockDim.x we are now sure that at the first iteration of the loop all threads in a block perform an operation, obviously these are extra operations because all threads sums 0 to partialSum[tx] element.These costs (cost to double to 2 the shared memory space plus the cost of 1 iteration through the for loop) will add to the total amount of cost to perform the partial sum.

Solution(b.1)

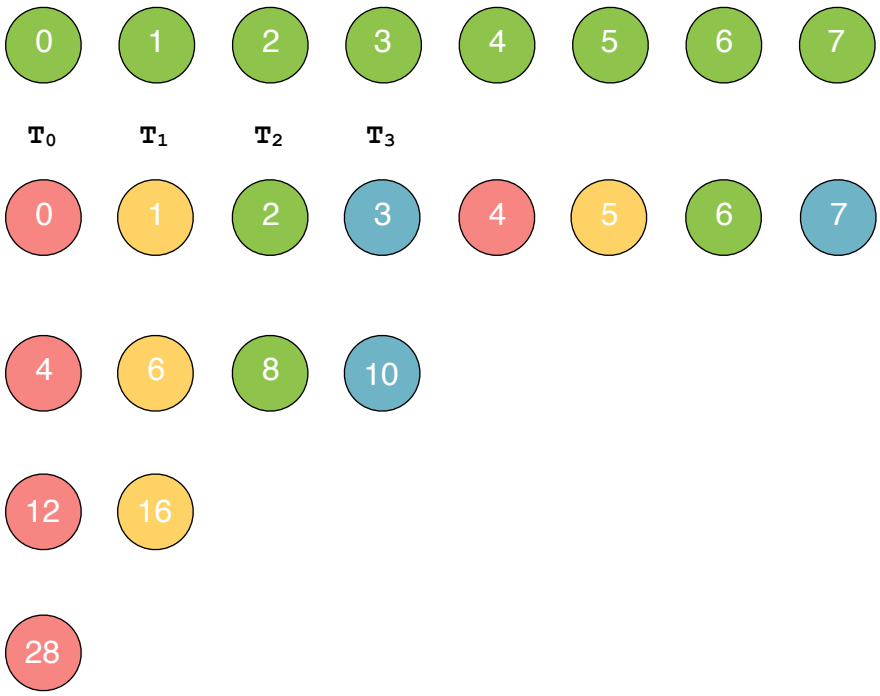
```
__shared__ int partialSum[BLOCK_SIZE];
int tx = threadIdx.x;
int i = blockIdx.x * blockDim.x + tx;
if (i < N) {
     partialSum[tx] = X[i];
     partialSum[tx + blockDim.x] = X[i + blockDim.x];
}
else
     partialSum[tx] = 0; // in case of extra threads may pad with 0's

for (int stride = blockDim.x; stride > 0; stride = stride/2)
{
     __syncthreads();
     if (tx < stride)
            partialSum[tx] += partialSum[tx + stride];
}

// Launch a kernel on the GPU with "half" thread for each element.
partialSumKernel<<<ceil(((float)size)/BLOCK_SIZE),BLOCK_SIZE/2 >>>(d_X, size);
```

See the source code in /cudaExercise/chapter05/reductionOptimized_v2(ex5.4).cu
In this solution we reduced the number of launched threads to minimum necessary to perform the partial sum. To do this, we note that is enough divide by 2 the BLOCK_SIZE, so the result of this operation is the number of threads necessary to perform the partial sum within a block (in this case). After the kernel launch, each threads load two values of the input array, one relative to their index and one relative to the next element starting from the half of the number of elements and the adding the row index calculated. This solution is more optimized then the solution (a.1), because there aren't threads that never perform an operation, and there are no need extra space for the shared memory.

Input array:



Exercise 5.2

Compare the modified kernels you wrote for Exercise 5.1. Which kernel incurred fewer additional arithmetic operations from the modification?

Solution(a.2):

Of course in the original kernel perform fewer additional operation that the kernel wrote in exercise 5.1 (see solution in exercise 5.1). More precisely in modified kernel we perform BLOCK_SIZE operation in addiction to the rest of the sum.

Solution(b.2):

The kernel in solution (b.1) perform the same number of additional arithmetic operation of the original kernel.

Exercise 5.3

Write a complete kernel based on Exercise 5.1 by (1) adding the statements that load a section of the input array from global memory to shared memory, (2) using blockIdx.x to allow multiple blocks to work on different sections of the input array, (3) writing the reduction value for the section to a location according to the blockIdx.x so that all blocks will deposit their section reduction value to the lower part of the input array in global memory.

Solution(a.3):

```
#define BLOCK_SIZE 16

__global__ void partialSumKernel(int *X, int N)
{
     __shared__ int partialSum[2 * BLOCK_SIZE];
     int tx = threadIdx.x;
     int i = blockIdx.x * blockDim.x + tx;
     partialSum[tx] = (i < N) ?  X[i] : 0; //<- last block may pad with 0's
     partialSum[tx + blockDim.x] = 0;

     for (int stride = blockDim.x; stride > 0; stride = stride/2)
     {
       __syncthreads();
       if (tx <= stride) {
             partialSum[tx] += partialSum[tx + stride];
       }
     }
     if (tx == 0)
       X[blockIdx.x] = partialSum[tx];
}
```

In this kernel code we can have multiple block to work on different section.
At the end of computation every thread with threadIdx.x in each block save the result in the lower part of the input array. If we suppose to have an array with 32 elements, at the end of computation we'll have 120 and 375 in h_X[0] and h_X[1] respectively.

Solution(b.3):

```
#define BLOCK_SIZE 512
__global__ void partialSumKernel(int *X, int N)
{
     __shared__ int partialSum[BLOCK_SIZE];
     int tx = threadIdx.x;
     int i = blockIdx.x * blockDim.x + tx;

     if (i < N) {
       partialSum[tx] = X[i];
       partialSum[tx + blockDim.x] = X[i + gridDim.x * blockDim.x];
     }
     else
       partialSum[tx] = 0; // last block may pad with 0's

     for (int stride = blockDim.x; stride > 0; stride = stride/2)
     {
       __syncthreads();
       if (tx < stride) {
             partialSum[tx] += partialSum[tx + stride];
       }
     }
     if (tx == 0)
       X[blockIdx.x] = partialSum[tx];
}
```

In this solution we can consider to make multiple block to perform the partial sum, always using the number of threads pair to the half of the number of elements of the input array. The kernel configuration parameters are the same of the solution (b.1).

Exercise 5.4

Design a reduction program based on the kernel you wrote for Exercise 5.3. The host code should (1) transfer a large input array to the global memory, (2) use a loop to repeatedly invoke the kernel you wrote for Exercise 5.3 with adjusted execution configuration parameter values so that the reduction result for the input array will eventually be produced.

Solution:

```
// Launch a kernel on the GPU with one thread for each element.
partialSumKernel<<<ceil(((float)size)/BLOCK_SIZE), BLOCK_SIZE >>>(d_X, size);
```

You can see the entire program source code of each solution (a.x and b.x) in:

- /cudaExercise/chapter05/reductionOptimized_v1.cu
- /cudaExercise/chapter05/reductionOptimized_v2(ex5.4).cu

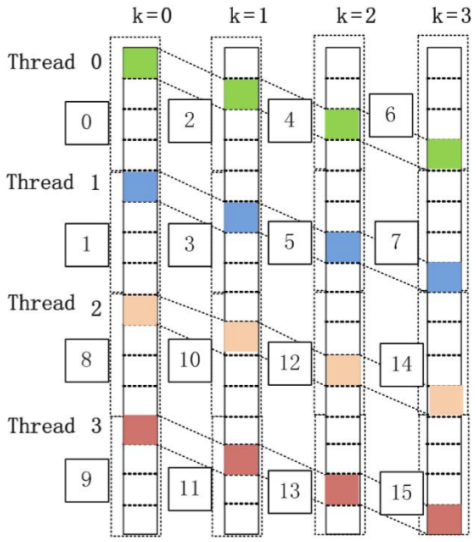
Changing “BLOCK_SIZE” and “size” variables’s values, you can perform different partial addition in with different configuration.

Exercise 5.5

For the tiled matrix multiplication kernel in Fig. 5.6, draw the access patterns of threads in a warp of Lines 9 and 10 for a small 16x16 matrix size. Calculate the tx values and ty values for each thread in a warp and use these values in the M and N index calculations in Lines 9 and 10. Show that the threads indeed access consecutive M and N locations in global memory during each iteration.

Solution:

In line 9 and 10 the access pattern of threads in a warp is the follow:



Below it show one or more phases of the loop, where it's visible the consecutive access in global memory by each threads in a warp (it have been used a Nvidia GTX 780 with 32 warp size).

```
bIdx(0,0) ── Mds[0][0] = M[0],      Nds[0][0] = N[0]
bIdx(0,0) ── Mds[0][1] = M[1],      Nds[0][1] = N[1]
bIdx(0,0) ── Mds[0][2] = M[2],      Nds[0][2] = N[2]
bIdx(0,0) ── Mds[0][3] = M[3],      Nds[0][3] = N[3]
bIdx(0,0) ── Mds[1][0] = M[16],     Nds[1][0] = N[16]
bIdx(0,0) ── Mds[1][1] = M[17],     Nds[1][1] = N[17]
bIdx(0,0) ── Mds[1][2] = M[18],     Nds[1][2] = N[18]
bIdx(0,0) ── Mds[1][3] = M[19],     Nds[1][3] = N[19]
bIdx(0,0) ── Mds[2][0] = M[32],     Nds[2][0] = N[32]
bIdx(0,0) ── Mds[2][1] = M[33],     Nds[2][1] = N[33]
bIdx(0,0) ── Mds[2][2] = M[34],     Nds[2][2] = N[34]
bIdx(0,0) ── Mds[2][3] = M[35],     Nds[2][3] = N[35]
bIdx(0,0) ── Mds[3][0] = M[48],     Nds[3][0] = N[48]
bIdx(0,0) ── Mds[3][1] = M[49],     Nds[3][1] = N[49]
bIdx(0,0) ── Mds[3][2] = M[50],     Nds[3][2] = N[50]
bIdx(0,0) ── Mds[3][3] = M[51],     Nds[3][3] = N[51]
```

Exercise 5.6

For the simple matrix multiplication (P = M*N) based on row—major layout, which input matrix will have coalesced accesses?

A. M

B. N

C. M, N

D. Neither

Solution:

B. N

Within the thread block, the index for accessing N is simply the value of thread.Idx.x. The N elements accessed during the iteration 0 by T_0, T_1, T_2, T_3 are N[0], N[1], N[2] and N[3]. These elements are in consecutive locations in the global memory. The hardware detects that these accesses are made by threads in a wrap and to consecutive locations in the global memory it coalesces accesses into a consolidated access. This allows the DRAMs to supply data at hight rate.
For the next iterations is the same thing, so the correct answer is D.

Exercise 5.7

For the tiled matrix-matrix multiplication (M *N) based on row-major layout, which input matrix will have coalesced accesses?

A. M

B. N

C. M, N

D. Neither

Solution:

C. M, N

The M elements are loaded in line 9, where the index calculation for each thread uses ph to locate the left end of the tile. The linearized index calculation is equivalent, the two-dimensional array access expression M[Row][ph*TILE SIZE+tx]. Note that the column index used by the threads only differs in terms of threadIdx. The row index is determined by blockIdx.y and threadIdx.y which means that threads in the same thread block with identical blockIdx.y / threadIdx.y and adjacent threadIdx.x values will access adjacent M elements. That is each row of the tile is loaded by TILE_WIDTH threads whose thread Idx are identical in the y dimension and consecutive in the x dimension. The hardware will coalesce these loads.
In the case of N, the row index ph*TILE_SIZE+ty has the same value for all threads with the same threadIdx.y value. The first term, bx*TILE_SI ZE, is the same for all threads in the same block. The second term, tx, is simply the the threadIdx.x value. Therefore, threads with adjacent threadIdx.x values access adjacent N elements in a row. The hardware will coalesce these loads.

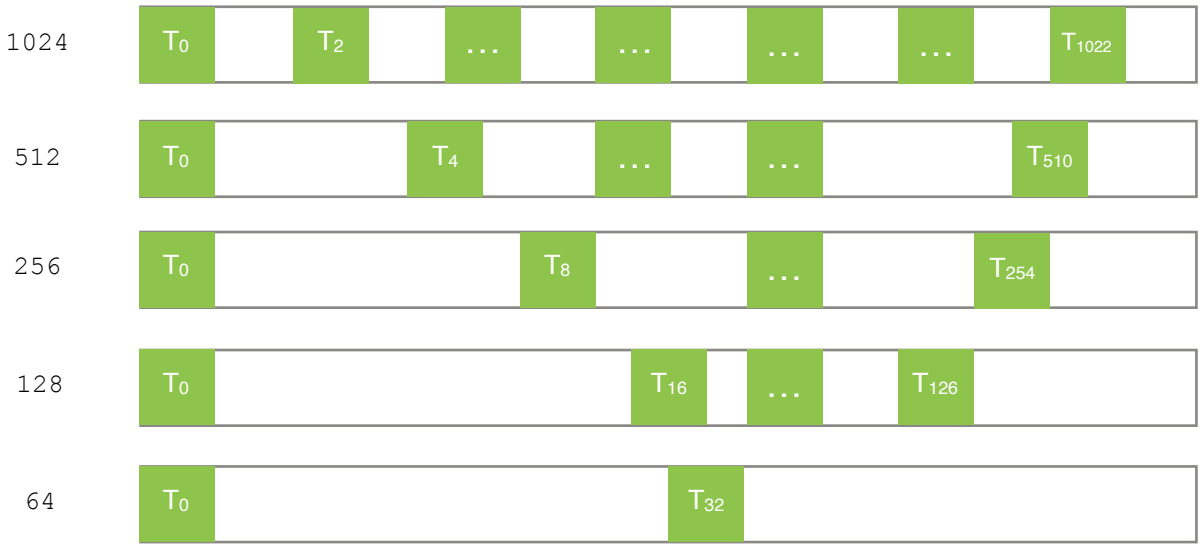
Exercise 5.8

For the simple reduction kernel, if the block size is 1024 and warp size is 32, how many warps in a block will have divergence during the 5th iteration?

A. 0
B. 1
C. 16
D. 32

Solution:

B. 1
To compute the partial sum using a simple reduction kernel, we need to 1024 threads that corresponds to 32 warps, but half of threads in each warp take a path and the other half of the same warps path take another path. During the 5th iteration there are 2 warps (64 threads), and only 1 thread in each warps perform the addition operation, while the the others 31 threads of the warps)have skipped the if statement in the previous iterations, so now none of them are executing in the kernel.



The last step of the above figure, corresponds to the 5th iteration of the simple kernel reduction. The green squares are indicating the active threads that perform the addition operation, as we can see T_0 will perform the sum from 0 to 31 and other 31 threads do nothing (take the same path in the if statement).

Exercise 5.9

For the improved reduction kernel, if the block size is 1024 and the warp size is 32, how many warps will have divergence during the 5th iteration.

A. 0
B. 1
C. 16
D. 32

Solution:

A. 0
During the 5th iteration we'll have 32 threads that perform the addition and 32 threads that skip it, so there aren't warps divergence during this iteration. Instead we'll have divergence during the next five iteration because the number of threads that perform the addition will be below 32.

Exercise 5.10

Write a matrix multiplication kernel function that corresponds to the design illustrated in Figure 5.17.

Solution:

```
#define TILE_WIDTH 2
#define DIM 8
__global__
void matrixMulKernel(float *P, float *M, float *N) {
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[TILE_WIDTH][2*TILE_WIDTH]; // <-- INCREASE SHARED MEMORY!!!

    int tx = threadIdx.x, bx = blockIdx.x;
    int ty = threadIdx.y, by = blockIdx.y;

    // identify row and column of the d_P element to work on
    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;

    float pValue = 0, pValue2 = 0;

    // Loop over the d_M and d_N tiles required to compute the d_P element
    for (int ph = 0; ph < DIM/TILE_WIDTH; ph++) {

       // Collaborative loading of d_M and d_N tiles n to the shared memory
       Mds[ty][tx] = M[Row * DIM + ph * TILE_WIDTH + tx];
       Nds[ty][tx] = N[(ph * TILE_WIDTH + ty) * DIM + Col];
       Nds[ty][tx + TILE_WIDTH] = N[(ph*TILE_WIDTH+ty)*DIM+Col+(DIM/2)];
        __syncthreads();

       // EVERY THREAD PERFORM TWO DOT PRODUCT
       for(int k = 0; k < TILE_WIDTH; k++){
           pValue  += Mds[ty][k]*Nds[k][tx];
           pValue2 += Mds[ty][k]*Nds[k][tx+TILE_WIDTH];
       }
      __syncthreads();
    }
    P[Row*DIM+Col] = pValue;
    P[Row*DIM+Col + (DIM/2)] = pValue2;
}

//2. Kernel launch code - with TILE_WIDTH^2 threads per block
dim3 dimGrid(ceil((DIM/TILE_WIDTH)/2.0), ceil(DIM/TILE_WIDTH), 1); // <-- THE GAIN!!!
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);
matrixMulKernel<<<dimGrid, dimBlock>>>(d_P, d_M, d_N);
```

We mustn't sub value this kernel code, how we can see if we have to do a matrix multiplication of two NxN matrices (assume square matrices), we reduced the total thread block by half and the global memory access by one-quarter!!!
Every thread in a block load 2 elements from global memory to shared memory and perform two dot product in each phase: one as the original kernel code of matrix multiplication, and one that was performed from the thread block that loaded the elements in position N[(ph*TILE_WIDTH+ty)*DIM+Col+(DIM/2)], but that now not exist of course. Furthermore we have increase the allocated space in share memory for Nds to permit to the thread blocks to load the other elements of N that will be performed using the same row elements of M or Mds in this case(see chapter 5 pag.128 for more information).

Exercise 5.11

For tiled matrix multiplication out of the possible range of values for BLOCK_SIZE, for what values of BLOCK_SIZE will the kernel completely avoid un-coalesced accesses to global memory? (You need to consider only square blocks.)

Solution:

For all values of BLOCK_SIZE that are submultiple of the matrix dimension.

Exercise 5.12

In an attempt to improve performance, a bright young engineer changed the reduction kernel into the following. (A) Do you believe that the performance will improve? Why or why not? (B) Should the engineer receive a reward or a lecture? Why?

```
__shared__ float partialSum[ ];
unsigned int tid=threadIdx.x;
for (unsigned int stride = n>>1; stride >= 32; stride >>= 1) {
      __syncthreads();
      if (tid < stride)
            shared[tid] += shared[tid + stride];
}
__syncthreads();
if (tid < 32) { // unroll last 5 predicated steps
      shared[tid] += shared[tid + 16];
      shared[tid] += shared[tid + 8]:
      shared[tid] += shared[tid + 4];
      shared[tid] += shared[tid + Z];
      shared[tid] += shared[tid + 1];
}
```

Solution:

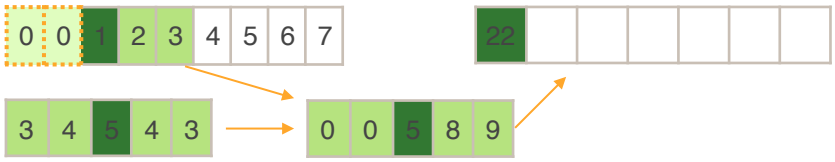
There are many problems in this code: First, there are syntax error in the use of the shared variable, that is the programmer uses “shared” instead of “partialSum”. The if we look over di error, there is a secondo problem about the missing preload before the use of the share memory.
So we may conclude saying that the young engineer wouldn't receive a reward, but a lecture to the chapter 4 and 5 of “programming massively paralleled processors 3rd edition Kirk, Hu”.

Chapter 7

Exercise 7.1

Calculate the P[0] value in Fig. 7.3

Solution:




```
P[0] = N[-2]*M[0] + N[-1]*M[1] + N[0]*M[2] + N[1]*M[3] + N[2]*M[4]
     = 0*2 + 0*4 + 1*5 + 2*4 + 3*3
     = 22
```

Exercise 7.2

Consider performing a 1D convolution on array N = {4,1,3,2,3} with mask M = {2,1,4}. What is the resulting output array?

Solution:

```
P[0] = 0*2 + 4*1 + 1*4 = 8
P[1] = 4*2 + 1*1 + 3*4 = 21
P[2] = 1*2 + 3*1 + 2*4 = 13
P[3] = 3*2 + 2*1 + 3*4 = 20
P[4] = 2*2 + 3*1 + 0*4 = 7
```

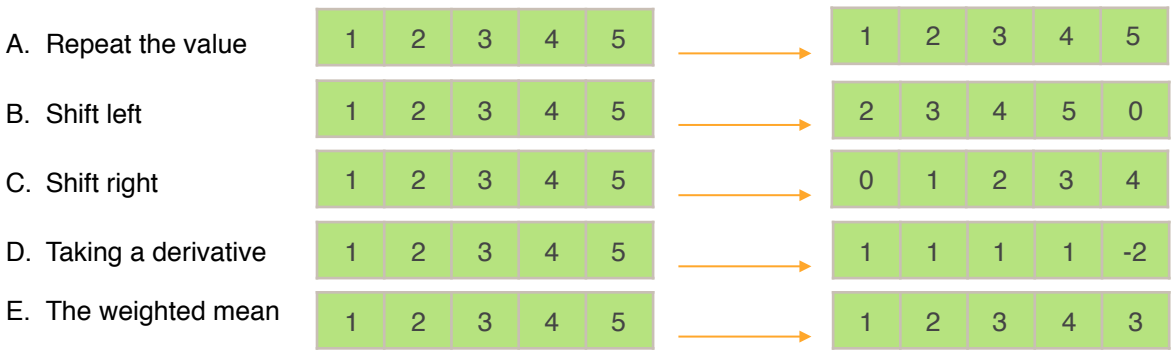
8	21	13	20	7

Exercise 7.3

What do you think the following 1D convolution masks are doing?

A. [0 1 0]
B. [0 0 1]
C. [1 0 0]
D. [-1/2 0 1/2]
E. [1/3 1/3 1/3]

Solution:



Exercise 7.4

Consider performing a 1D convolution on an array of size n with a mask of size m:

a. How many halo cells are there in total?
b. How many multiplications are performed if halo cells are treated as multiplications (by 0)?
c. How many multiplications are performed if halo cells are not treated as multiplications?

Solution:

a. There are m-1 total ghost cells: (m-1)/2 to the left of first element of array and (m-1)/2 to the right of the input array.
b. If halo cells are treated as multiplications (by 0), are performed n*m multiplications
c. If halo cells are not treated as multiplications, are performed (n*m) - (m-1) multiplications

Exercise 7.5

Consider performing a 2D convolution on a square matrix of size n x n with a square mask of size m x m:

a. How many halo cells are there in total?
b. How many multiplications are performed if halo cells are treated as multiplications (by 0)?
c. How many multiplications are performed if halo cells are not treated as multiplications?

Solution:

a. There are (n+m-1)*(m-1) + (n*(m-1)) = (m-1)*(2n + m - 1) total halo cells.
b. If halo cells are treated as multiplications (by 0) are performed n^2 * m^2 multiplications.
c. If halo cells are not treated as multiplications, are performed (n^2 * m^2) - (m-1)*(2n + m - 1).

Exercise 7.6

Consider performing a 2D convolution on a rectangular matrix of size n1 x n2 with a rectangular mask of size m1 x m2:

a. How many halo cells are there in total?
b. How many multiplications are performed if halo cells are treated as multiplications (by 0)?
c. How many multiplications are performed if halo cells are not treated as multiplications?

Solution:

a. There are (n2 + m2-1)*(m1-1) + (n1(m2-1)) halo cells.
b. If halo cells are treated as multiplications (by 0) are performed (n1*n2)*(m1*m2) multiplications.
c. If halo cells are not treated as multiplications, are performed (n1*n2)*(m1*m2) - (n2 + m2-1)*(m1-1) + (n1(m2-1)).

Exercise 7.7

Consider performing a 1D tiled convolution with the kernel shown in Fig. 7.11 on an array of size n with a mask of size m using a tiles of size t:

a. How many blocks are needed?
b. How many threads per block are needed?
c. How much shared memory is needed in total?
d. Repeat the same questions if you were using the kernel in Fig. 7.13.

Solution:

a. Are needed Sup(n / t) blocks of size t.
b. Are needed t threads per block
c. In total is needed Sup(n / (t + m-1)) space of shared memory per block
d. Are needed the same number of threads and block of points a and b, but now is enough t space of share memory per block.

Exercise 7.8

Revise the 1D kernel in Fig. 7.6 to perform 2D convolution. Add more width parameters to the kernel declaration as needed.

Solution:

```
__global__
void convolution_2D_basic_kernel(float *N, float *M, float *P, int height, int width){

    int Row = blockIdx.y * blockDim.y + threadIdx.y;
    int Col = blockIdx.x * blockDim.x + threadIdx.x;

    if ( Row < height && Col < width) {

        int y = Row - (MASK_WIDTH) / 2;
        int x = Col - (MASK_WIDTH) / 2;
        float Pvalue = 0.0f;

        for (int i = 0; i < MASK_WIDTH; i++) {
            if (y + i >= 0 && y + i < height) {
                for (int j = 0; j < MASK_WIDTH; j++) {
                    if (x + j >= 0 && x + j < width) {
                        Pvalue += N[(y+i) * width + (x+j)] * M[i * MASK_WIDTH + j];
                    }
                }
            }
        }
        P[ Row * width + Col] = Pvalue;
    }
}
```

Exercise 7.9

Revise the tiled 1D kernel in Fig. 7.8 to perform 2D convolution. Keep in mind that the host code also needs to be changed to declare a 2D M array in the constant memory. Pay special attention to the increased usage of shared memory. Also, the N_ds needs to be declared as a 2D shared memory array.

Solution:

```
__global__void convolution_2D_basic_kernel(float *N, float *P, int height, int width){
    int Row = blockIdx.y * blockDim.y + threadIdx.y;
    int Col = blockIdx.x * blockDim.x + threadIdx.x;
    if ( Row < height && Col < width) {
        int y = Row - (MASK_WIDTH / 2);
        int x = Col - (MASK_WIDTH / 2);
        float Pvalue = 0.0f;

        for (int i = 0; i < MASK_WIDTH; i++) {
            if (y + i >= 0 && y + i < height) {
                for (int j = 0; j < MASK_WIDTH; j++) {
                    if (x + j >= 0 && x + j < width)
                        Pvalue += N[(y+i) * width + (x+j)] * M[i][j];
                }
            }
        }
        P[ Row * width + Col] = Pvalue;
    }
}
```

Exercise 7.10

Revise the tiled 1D kernel in Fig. 7.11 to perform 2D convolution. Keep in mind that the host code also needs to be changed to declare a 2D M array in the constant memory. Pay special attention to the increased usage of shared memory. Also, the N_ds needs to be declared as a 2D shared memory array.

Solution:

```
__global__
void convolution_2D_basic_kernel(float *N, float *P, int height, int width, const int
Mask_Width) {
     int tx = threadIdx.x, ty = threadIdx.y;
     int row_o = blockIdx.y * O_TILE_WIDTH + ty;
     int col_o = blockIdx.x * O_TILE_WIDTH + tx;
     int row_i = row_o - (Mask_Width / 2);
     int col_i = col_o - (Mask_Width / 2);

     __shared__ float N_ds[TILE_SIZE+MAX_MASK_WIDTH-1][TILE_SIZE+MAX_MASK_WIDTH-1];

     if ((row_i >= 0) && (row_i < height) && (col_i >= 0) && (col_i < width))
       N_ds[ty][tx] = N[row_i * width + col_i];
     else
       N_ds[ty][tx] = 0.0f;

     __syncthreads();
     float Pvalue = 0.0f;
     if (ty < O_TILE_WIDTH && tx < O_TILE_WIDTH) {
       for (int i = 0; i < Mask_Width; i++) {
              for (int j = 0; j < Mask_Width; j++)
                     Pvalue += N_ds[i+ty][j+tx] * M[i * Mask_Width + j];
       }
       if (row_o < height && col_o < width)
              P[row_o * width + col_o] = Pvalue;
     }
}
```

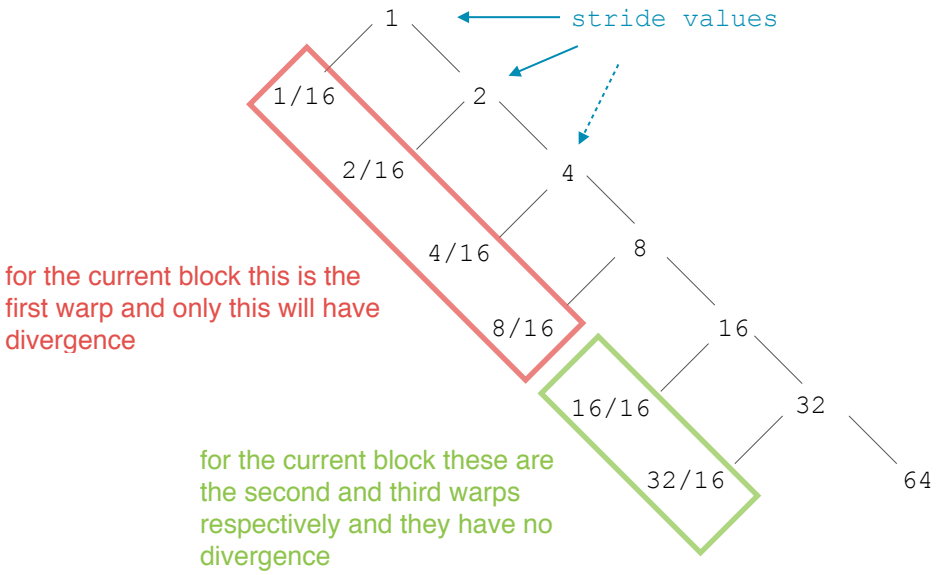
Chapter 8

Exercise 8.1

Analyze the parallel scan kernel 1n Fig. 8. 2. Show that control divergence only occurs in the first warp of each block for stride values up to half the warp size; i. e., for warp size 32, control divergence will occur to iterations for stride values 1, 2, 4, 8, and 16.

Solution:

The control divergence only occurs in the first warp of each block for stride values up to half the warp size, because after this point the number of threads for each block that follow each branch of the if statement is multiple of the warp size.
For instante if we have warp size 16 and we have block size 128:



Exercise 8.2

For the Brent—Kung scan kernel, assume that we have 2048 elements. How many additions will be performed in both the reduction tree phase and the inverse reduction tree phase?

a. (2048-1)*2
b. (1024-1)*2
c. 1024*1024
d. 10*1024

Solution:

a. (2048-1)*2
Reduction tree phase: $(N-1)$ operations
Reverse tree: $((N-1)-\log_2(N))$ operations
If we solve the equation: $(N-1)+((N-1)-\log_2(N))=4083\sim 4096$

Exercise 8.3

For the Kogge—Stone scan kernel, based on reduction trees, assume that we have 2048 elements.Which of the following gives the closest approximation of addition that will be performed?

a. (2048-1)*2
b. (1024-1)*2
c. 1024*1024
d. 10*1024

Solution:

d. 10*1024 = 10240
Number of operations is in general:

$$\begin{aligned}\sum(N-stride) &= (N-1)+(N-2)+(N-4)+...+(N-N/2)\\ &= N\cdot\log_2(N)-(N-1)\\ &= 2048\cdot 10-2047\\ &= 18433\end{aligned}$$

Exercise 8.4

Use the algorithm in Fig. 8.3 to complete an exclusive scan kernel.

Solution:

```
__global__ void Kogge_Stone_scan_kernel(float *X, float *Y, int InputSize)
{
     __shared__ float XY[SECTION_SIZE];
     int i = blockIdx.x * blockDim.x + threadIdx.x;

     if (i < InputSize && threadIdx.x != 0) {
       XY[threadIdx.x] = X[i - 1];
     }
     else {
       XY[threadIdx.x] = 0;
     }

     if (threadIdx.x < InputSize)
     {
       // Perform iterative exclusive scan on XY
       for (unsigned int stride = 1; stride < blockDim.x; stride *= 2) {
              if (threadIdx.x >= stride) {
                     __syncthreads();
                     XY[threadIdx.x] += XY[threadIdx.x - stride];
              }
       }
       Y[i] = XY[threadIdx.x];
     }
}
```

Exercise 8.5

Complete the host code and all three kernels for the hierarchical parallel scan algorithm in Fig. 8.9.

Solution:

See the sources in: /cudaExercises/chapter08/

Exercise 8.6

Analyze the hierarchical parallel scan algorithm and show that it is work-efficient and the total number of additions is no more than 4*N - 3.

Solution:

- Kernel 1. Use Kogge-Stone, Brent-Kung, or three-phase algorithm to perform scan on each section (i.e. Scan Block) and writes XY to Y and final element of XY to auxiliary array S.
- Kernel 2. Use Kogge-Stone, Brent-Kung, or three-phase algorithm to perform scan on S. Write output back to S.
- Kernel 3. Adds S to every element in corresponding scan block of Y. Use block size = section size.

So we have NSB * (N/SSB * log N/SSB) + NSB log * NSB + NSB * NSB/SSB - 3, that is about no more than 4*N-3 (where NSB = number of section blocks and SSB = size of section block).

Exercise 8.7

Consider the following array: {4, 6, 7, 1, 2, 8, 5, 2}. Perform a parallel inclusive prefix scan on the array by using the Kogge-Stone algorithm. Report the intermediate states of the array after each step.

Solution:

Array input: { 4 6 7 1 2 8 5 2 }

Inclusive Kogge-stone scan:

- A = { 4 10 13 8 3 10 13 7 }
- A = { 4 10 17 18 16 18 16 17 }
- A = { 4 10 17 18 20 28 33 35 }

Exercise 8.8

Repeat the previous problem by using the work-efficient algorithm.

Solution:

Three-phase scan:

- A = { 4 10 7 8 2 10 5 7 }
- A = { 4 10 7 18 2 28 5 35 }
- A = { 4 10 17 18 20 28 33 35 }

Exercise 8.9

By using the two-level hierarchical scan discussed in Section 8.5, determine the largest possible dataset that can be handled if computing on a:

a. GeForce GTX 280?
b. Tesla C2050?
c. GeForce GTX 690?

Solution:

For the technical specifications see the follow table considering to use the Brent-Kung kernel:

DEVICE	CAPABILTIY	MAXIMUM X-DIMENSION OF A GRID OF THREAD BLOCKS	MAXIMUM NUMBER OF THREADS PER BLOCK	LARGEST DATASET
GeForce GTX 280	1.3	65535	512	67108864
Tesla C2050	2.0	65535	1024	134217728
GeForce GTX 690	3.0	2^{31}-1	1024	$2.2e^{12}$