Chapter Exercises Solution Manual

Programming Massively Parallel Processors: A Hands-on Approach
By Wen-mei W. Hwu, David B. Kirk, and Izzat El Hajj

Chapter 1: Introduction
No exercises.

Chapter 2: Heterogeneous data parallel computing

1. (C) i=blockIdx.x*blockDim.x + threadIdx.x;
2. (C) i=(blockIdx.x*blockDim.x + threadIdx.x)*2;
3. (D) i=blockIdx.x*blockDim.x*2 + threadIdx.x;
4. (C) 8192
5. (D) v * sizeof(int)
6. (D) (void **) &A_d
7. (C) cudaMemcpy(A_d, A_h, 3000, cudaMemcpyHostToDevice);
8. (C) cudaError_t err;
9.
    a. 128
    b. 200,064
    c. 1,563
    d. 200,064
    e. 200,000
10. The intern should use both the "__host__" keyword and the "__device__" keyword in declaring the functions that need to be executed on both the host and the device. The CUDA compiler will generate both versions of the functions.

Chapter 3: Multidimensional grids and data

1. *Not included*
2.

```
__global__ void matrixVectorMulKernel(float *A, float *B, float *C, int vectorLen) {
  int i = threadIdx.x + blockIdx.x*blockDim.x;
  float sum = 0.0f;
  if (i < vectorLen) {
    for (int j = 0; j < vectorLen; j++)
      sum += B[i*vectorLen + j] * C[j];
    A[i] = sum;
  }
}

void matrixVectorMul(float *h_A, float *h_B, float *h_C, int vectorLen) {
  float *d_A, *d_B, *d_C;
  int matrixSize = vectorLen*vectorLen*sizeof(float);
  int vectorSize = vectorLen*sizeof(float);
  cudaMalloc((void**)&d_A, vectorSize);
  cudaMalloc((void**)&d_B, matrixSize);
  cudaMalloc((void**)&d_C, vectorSize);

  // Initialize GPU memory contents with 0 or host data
  cudaMemset(d_A, 0, vectorSize);
  cudaMemcpy(d_B, h_B, matrixSize, cudaMemcpyHostToDevice);
  cudaMemcpy(d_C, h_C, vectorSize, cudaMemcpyHostToDevice);
```

```
    // Execute the GPU kernel
    dim3 threads(128);
    dim3 blocks((vectorLen+(128-1))/128);
    matrixMulKernel<<<blocks, threads>>>(d_A, d_B, d_C, vectorLen);

    // Copy results back to host
    cudaMemcpy(h_A, d_A, vectorSize, cudaMemcpyDeviceToHost);

    // Free GPU memory and return
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

3.
    a. 512
    b. 48,640
    c. 95
    d. 45,000
4.
    e. 8,010
    f. 5,020
5. 1,008,010

Chapter 4: Compute architecture and scheduling

1.
    a. 4
    b. 32
    c.
        i. 24
        ii. 16
        iii. 100%
        iv. 25%
        v. 75%
    d.
        i. 32
        ii. 32
        iii. 50%
    e.
        i. 3
        ii. 2
2. 2048
3. 1
4. 17.1%
5. No, it is not a good idea because there is no guarantee that the warp size will continue to be 32. Making assumptions about the warp size in the code may cause the code to break if the warp size changes in future architectures.
6. (c) 512 threads per block
7.
    a. Possible, 50%
    b. Possible, 50%
    c. Possible, 50%
    d. Possible, 100%

e. Possible, 100%

8.
   a. Full occupancy can be achieved
   b. Occupancy is limited by the number of blocks per SM
   c. Occupancy is limited by the number of registers per SM
9. It is not possible to launch 32x32 thread blocks if the CUDA device allows up to 512 threads per block.

Chapter 5: Memory architecture and data locality

1. No. In this case, no threads share input data, so read-sharing cannot be exploited through cooperative use of shared memory.
2. We draw the figure with access order proceeding downwards (instead of to the right, as in Figure 5.6). In each row, unique entries are given a color. Note that the number of repeated entries in a row (number of values of a particular color) is the same as the tile size. Using a tiling scheme, one global memory load services TILE_WIDTH threads.

   2x2 Tiling:

T 0,0	T 0,1	T 1,0	T 1,1
Md 0,0 * Nd 0,0	Md 0,0 * Nd 1,0	Md 0,1 * Nd 0,0	Md 0,1 * Nd 1,0
Md 1,0 * Nd 0,1	Md 1,0 * Nd 1,1	Md 1,1 * Nd 0,1	Md 1,1 * Nd 1,1
Md 2,0 * Nd 0,2	Md 2,0 * Nd 1,2	Md 2,1 * Nd 0,2	Md 2,1 * Nd 1,2
Md 3,0 * Nd 0,3	Md 3,0 * Nd 1,3	Md 3,1 * Nd 0,3	Md 3,1 * Nd 1,3
Md 4,0 * Nd 0,4	Md 4,0 * Nd 1,4	Md 4,1 * Nd 0,4	Md 4,1 * Nd 1,4
Md 5,0 * Nd 0,5	Md 5,0 * Nd 1,5	Md 5,1 * Nd 0,5	Md 5,1 * Nd 1,5
Md 6,0 * Nd 0,6	Md 6,0 * Nd 1,6	Md 6,1 * Nd 0,6	Md 6,1 * Nd 1,6
Md 7,0 * Nd 0,7	Md 7,0 * Nd 1,7	Md 7,1 * Nd 0,7	Md 7,1 * Nd 1,7

   4x4 Tiling:

T 0,0	T 1,0	T 2,0	T 3,0	T 0,0	T 1,1	T 2,1	T 3,1	T 0,2	T 1,2	T 2,2	T 3,2	T 0,3	T 1,3	T 2,3	T 3,3
M0,0* N0,0	M0,0* N1,0	M0,0* N2,0	M0,0* N3,0	M0,1* N0,0	M0,1* N1,0	M0,1* N2,0	M0,1* N3,0	M0,2* N0,0	M0,2* N1,0	M0,2* N2,0	M0,2* N3,0	M0,3* N0,0	M0,3* N1,0	M0,3* N2,0	M0,3* N3,0
M1,0* N0,1	M1,0* N1,1	M1,0* N2,1	M1,0* N3,1	M1,1* N0,1	M1,1* N1,1	M1,1* N2,1	M1,1* N3,1	M1,2* N0,1	M1,2* N1,1	M1,2* N2,1	M1,2* N3,1	M1,3* N0,1	M1,3* N1,1	M1,3* N2,1	M1,3* N3,1
M2,0* N0,2	M2,0* N1,2	M2,0* N2,2	M2,0* N3,2	M2,1* N0,2	M2,1* N1,2	M2,1* N2,2	M2,1* N3,2	M2,2* N0,2	M2,2* N1,2	M2,2* N2,2	M2,2* N3,2	M2,3* N0,2	M2,3* N1,2	M2,3* N2,2	M2,3* N3,2
M3,0* N0,3	M3,0* N1,3	M3,0* N2,3	M3,0* N3,3	M3,1* N0,3	M3,1* N1,3	M3,1* N2,3	M3,1* N3,3	M3,2* N0,3	M3,2* N1,3	M3,2* N2,3	M3,2* N3,3	M3,3* N0,3	M3,3* N1,3	M3,3* N2,3	M3,3* N3,3
M4,0* N0,4	M4,0* N1,4	M4,0* N2,4	M4,0* N3,4	M4,1* N0,4	M4,1* N1,4	M4,1* N2,4	M4,1* N3,4	M4,2* N0,4	M4,2* N1,4	M4,2* N2,4	M4,2* N3,4	M4,3* N0,4	M4,3* N1,4	M4,3* N2,4	M4,3* N3,4
M5,0* N0,5	M5,0* N1,5	M5,0* N2,5	M5,0* N3,5	M5,1* N0,5	M5,1* N1,5	M5,1* N2,5	M5,1* N3,5	M5,2* N0,5	M5,2* N1,5	M5,2* N2,5	M5,2* N3,5	M5,3* N0,5	M5,3* N1,5	M5,3* N2,5	M5,3* N3,5
M6,0* N0,6	M6,0* N1,6	M6,0* N2,6	M6,0* N3,6	M6,1* N0,6	M6,1* N1,6	M6,1* N2,6	M6,1* N3,6	M6,2* N0,6	M6,2* N1,6	M6,2* N2,6	M6,2* N3,6	M6,3* N0,6	M6,3* N1,6	M6,3* N2,6	M6,3* N3,6
M7,0* N0,7	M7,0* N1,7	M7,0* N2,7	M7,0* N3,7	M7,1* N0,7	M7,1* N1,7	M7,1* N2,7	M7,1* N3,7	M7,2* N0,7	M7,2* N1,7	M7,2* N2,7	M7,2* N3,7	M7,3* N0,7	M7,3* N1,7	M7,3* N2,7	M7,3* N3,7

3. Without a barrier in the places noted, threads may begin reading shared memory locations expecting values from other threads to be there before those other threads have actually written those values. Or conversely, threads may get ahead of other threads, and write values to shared memory on the next iteration of the outer loop while other threads still needed the old values to finish the computation of the inner loop.
4. To share common input values between threads in a block by only loading them once from global memory, and then read-sharing the values out of the shared memory. If each thread directly loaded the value from global memory into a register in this case, significantly more global memory bandwidth is consumed for the same computation.
5. 32x reduction
6. 512,000
7. 1,000
8.
    a. N
    b. N/T
9.
    a. Memory-bound
    b. Compute-bound
10.
    a. None
    b. To guarantee ordering of shared memory reads and writes across threads in this kernel, a __syncthreads() call should be placed between lines 10 and 11.
11.
    a. 1024
    b. 1024
    c. 8
    d. 8
    e. 516
    f. 0.5
12.
    a. Occupancy is limited by the amount of shared memory per SM
    b. Full occupancy can be achieved

Chapter 6: Performance considerations

1. *Not included*
2. For block sizes of 16 and 32 (if allowed by the CUDA version you have) will the kernel generate completely coalesced global memory accesses.
3.
    a. Coalesced
    b. Not applicable
    c. Coalesced
    d. Uncoalesced
    e. Not applicable

f. Not applicable
g. Coalesced
h. Not applicable
i. Uncoalesced

4.
a. 0.25
b. 8
c. 12.8

Chapter 7: Convolution

1. 22
2. {8, 21, 13, 20, 7}
3.
a. This is an identity mask – each output element is equal to the input element of the same index.
b. This mask shifts the elements of the array one index to the left.
c. This mask shifts the elements of the array one index to the right.
d. This mask computes the gradient at each element, measuring how much the input array values are changing at each location.
e. This mask will smooth or blur the values of the input array.

4.
a. $M - 1$
b. $N*M$
c. $N*M - (M^2 - 1)/4$

5.
a. $(N + M - 1)^2 - N^2$
b. N^2*M^2
c. N^2*M^2
- $4*(1 + 2 + ... + (M - 1)/2)*(1 + 2 + ...+ (M - 1)/2)$
- $8*(1 + 2 + ... + (M - 1)/2)*((M + 1)/2 + ... + (M - 1))$
- $4*(1 + 2 + ... + (M - 1)/2)*M*(N - M + 1)$

6.
a. $(N_1 + M_1 - 1)(N_2 + M_2 - 1) - N_1N_2$
b. $N_1N_2M_1M_2$
c. $N_1N_2M_1M_2$
- $4*(1 + 2 + ... + (M1 - 1)/2)(1 + 2 + ... + (M2 - 1)/2)$
- $4*(1 + 2 + ... + (M1 - 1)/2)((M2 + 1)/2 + ... + (M2 - 1))$
- $4*(1 + 2 + ... + (M2 - 1)/2)((M1 + 1)/2 + ... + (M1 - 1))$
- $2*(1 + 2 + ... + (M1 - 1)/2)*M2*(N2 - M2 + 1)$
- $2*(1 + 2 + ... + (M2 - 1)/2)*M1*(N1 - M1 + 1)$

7.
a. N^2/T^2
b. $(T + M - 1)^2$
c. $(T + M - 1)^2*4$ bytes
d.
    a. N^2/T^2
    b. T^2
    c. T^2*4 bytes

8. *Not included*
9. *Not included*
10. *Not included*

Chapter 8: Stencil

1.
    a. 1,726,576
    b. 3375
    c. 8000
    d. 64
2.
    a. 18,432
    b. 14,400
    c. 2.54
    d. 12,288
    e. 4096

Chapter 9: Parallel histogram

1. 10 MOP/s
2. 73.5 MOP/s
3. 50 MOP/s
4. 4.55 GFLOPS
5. (d) atomicAdd(&Total, Partial);
6.
    a. 524,288
    b. 65,536
    c. 16,384

Chapter 10: Reduction

1. 32
2. 0
3. *Not included*
4. *Not included*
5. *Not included*
6.
    a.

Initial array:	6	2	7	4	5	8	3	1
After step 1:	8	2	11	4	13	8	4	1
After step 2:	19	2	11	4	17	8	4	1
After step 3:	36	2	11	4	17	8	4	1

b.

Initial array:	6	2	7	4	5	8	3	1
After step 1:	11	10	10	5	5	8	3	1
After step 2:	21	15	10	5	5	8	3	1
After step 3:	36	15	10	5	5	8	3	1

Chapter 11: Prefix sum (scan)

1.

Initial array:	4	6	7	1	2	8	5	2
After step 1:	4	10	13	8	3	10	13	7
After step 2:	4	10	17	18	16	18	16	17
After step 3:	4	10	17	18	20	28	33	35

2. *Not included*
3. Threads exit the loop when the stride is greater than thread index. Because stride doubles every iteration, once the stride equals the warp size, all subsequent strides will be a multiple of the warp size, ensuring that all threads within a particular warp have the same behavior from that point on.
4. 20,481
5.

Initial array:	4	6	7	1	2	8	5	2
After step 1:	4	10	7	8	2	10	5	7
After step 2:	4	10	7	18	2	10	5	17
After step 3:	4	10	7	18	2	10	5	35
After step 4	4	10	7	18	2	28	5	35
After step 5	4	10	17	18	20	28	33	35

6. 4,083
7.

```
__global__ void exclusive_scan_kernel(float *X, float *Y, int InputSize) {
  __shared__ float XY[SECTION_SIZE];
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  // Shift the input elements by one index,
  //   and set the element for index 0 to the value 0
  if (i < InputSize - 1) {
    XY[threadIdx.x + 1] = X[i];
  }
  if (threadIdx.x == 0) {
    XY[0] = 0.0f;
  }
```

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

8. *Not included*

Chapter 12: Merge

1. i = 5, j = 3
2. i = 5, j = 1
3. *Not included*
4.
    a. 204,800
    b. 200
    c. 204,800

Chapter 13: Sorting

1. *Not included*
2. *Not included*
3. *Not included*
4. *Not included*

Chapter 14: Sparse matrix computation

1.
    a. COO:

Row indices:	0	0	1	2	2	3	3
Column indices:	0	2	2	1	2	0	3
Data:	1	7	8	4	3	2	1

    b. CSR:

Row pointers:	0	2	3	5	7		
Column indices:	0	2	2	1	2	0	3
Data:	1	7	8	4	3	2	1

    c. ELL:

Column indices:			Data:	
	0	2	1	7
	2	-	8	-
	1	2	4	3
	0	3	2	1

d. JDS:

Row permutations:	0	2	3	1

Iteration pointers	0	4	7

Column indices:	0	1	0	2	2	2	3

Data:	1	4	2	8	7	3	1

2.
   a. 3z
   b. 2z + m + 1
   c. The missing information is the maximum number of non-zeros per row. If we denote this by k, the storage requirement becomes: 2mk
   d. The missing information is the maximum number of non-zeros per row. If we denote this by k, the storage requirement becomes: 2z + m + k + 1
3. *Not included*
4.

```
// Declare host variables
unsigned int numRows, numCols;
// Original CSR
unsigned int *rowPtrsCSR, *colIdxCSR;
float *dataCSR;
// ELL
unsgined int numElemELL;
unsigned int *colIdxELL;
float *dataELL;
// COO
unsigned int nnzCOO;
unsigned int *rowIdxCOO, *colIdxCOO;
float *dataCOO;
// Vector
float *vecSrc, *vecDst;

// Initialize original host CSR matrix and other variables
[...]

// Transform to hybrid ELL-COO
for(unsigned int rowIdx = 0; rowIdx < numRows; ++rowIdx) {
  unsigned int nnz = rowPtrsCSR[rowIdx + 1] - rowPtrsCSR[rowIdx];
  for(unsigned int idx = 0; idx < nnz; ++idx) {
    if(idx < numElemELL) {
      unsigned int ellIdx = idx*numRows + rowIdx;
      colIdxELL[ellIdx] = colIdxCSR[rowPtrsCSR[rowIdx] + idx];
      dataELL[ellIdx] = dataCSR[rowPtrsCSR[rowIdx] + idx];
    } else {
      rowIdxCOO[nnzCOO] = rowIdx;
      colIdxCOO[nnzCOO] = colIdxCSR[rowPtrsCSR[rowIdx] + idx];
      dataCOO[nnzCOO] = dataCSR[rowPtrsCSR[rowIdx] + idx];
      ++nnzCOO;
    }
  }
}

// Declare device memory variables
unsigned int *colIdxELL_d;
```

```
float *dataELL_d;
float *vecSrc_d, *vecDst_d;

// Allocate device memory
cudaMalloc((void**) &colIdxELL_d, numRows*numElemELL*sizeof(unsigned int));
cudaMalloc((void**) &dataELL_d, numRows*numElemELL*sizeof(float));
cudaMalloc((void**) &vecSrc_d, numCols*sizeof(float));
cudaMalloc((void**) &vecDst_d, numRows*sizeof(float));

// Copy device memory
cudaMemcpy(colIdxELL_d,    colIdxELL,    numRows*numElemELL*sizeof(unsigned    int),
cudaMemcpyHostToDevice);
cudaMemcpy(dataELL_d,          dataELL,          numRows*numElemELL*sizeof(float),
cudaMemcpyHostToDevice);
cudaMemcpy(vecSrc_d, vecSrc, numRows*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(vecDst_d, vecDst, numRows*sizeof(float), cudaMemcpyHostToDevice);

// Launch kernel
SpMV_ELL <<< (numRows - 1)/BLOCK_SIZE + 1 , BLOCK_SIZE >>>(numRows, dataELL_d,
colIdxELL_d, vecSrc_d, vecDst_d);

// Copy device memory back
cudaMemcpy(vecDst, vecDst_d, numRows*sizeof(float), cudaMemcpyHostToDevice);

// Complete COO contributions
for(unsigned int idx = 0; idx < nnzCOO; ++idx) {
  vecDst[rowIdxCOO[idx]] += vecSrc[colIdxCOO[idx]]*dataCOO[idx]
}
```

5. *Not included*

Chapter 15: Graph traversal

1.

a.

	0	1	2	3	4	5	6	7
0			1			1		
1	1				1			1
2				1				
3	1						1	
4				1				
5		1						1
6					1			
7			1		1		1	

b.

srcPtrs	0	2	5	6	8	9	11	12	15

dst	2	5	0	4	7	3	0	6	3	1	7	4	2	4	6

c. Answers for iteration going from level 1 to level 2

i.

1. 8
2. 2

ii.

1. 8

2. 5
3. 3

iii.

1. 15
2. 4

iv.

1. 2
2. 2

2. *Not included*
3. *Not included*

Chapter 16: Deep learning

1. *Not included*
2. *This question contains an error that will be fixed in the next edition.*
3. *Not included*
4. The reads to X are coalesced. The lowest dimension is indexed by (w_out + q) where q is uniform across threads and w_out is stride 1 across threads.

Chapter 17: Iterative magnetic resonance imaging reconstruction

1. Denote by A_m the statements before the inner loop in the m^{th} iteration, and by B_m the inner loop as a whole in the m^{th} iteration.
    a. Before fission, the order will be: A_0, B_0, A_1, B_1, A_2, B_2, … A_{M-1}, B_{M-1}
    b. After fission, the order will be: A_0, A_1, A_2, … A_{M-1}, B_0, B_1, B_2, …, B_{M-1}. Yes, the execution results will be identical
2.
    a. Order of loop iterations (m,n) before loop interchange: (0,0), (0,1), (0,2), …, (0,N-1), (1,0), (1,1), (1,2), …, (1,N-1), …, (M-1,0), (M-1,1), (M-1,2), …, (M-1,N-1)
    b. Order of loop iterations (m,n) after loop interchange: (0,0), (1,0), (2,0), …, (M-1,0), (0,1), (1,1), (2,1), …, (M-1,1), …, (0,N-1), (1,N-1), (2,N-1), …, (M-1,N-1)
    c. Yes, the results will be identical.
3. In accessing x[], threads in a warp will access adjacent areas in memory so the access is coalesced. Moreover, the value each thread loads is used multiple times throughout the kernel which makes it worth storing that value in a register. In accessing kx[], threads in a warp will all load the same value at the same time, so the value only needs to be loaded once. If the L1 cache participates, the values for the next few iterations will also be available. Since each value loaded is only used once per thread, it does not make sense to store that value in a register.

Chapter 18: Electrostatic potential map

1. *Not included*
2. In Figure 18.6, each loop iteration performs 4 loads, 11 floating point operations (3 subtractions, 3 additions, 3 multiplications, 1 division, and 1 sqrtf instrinsic), and 1 branch. Hence, 8 iterations perform 32 load operations, 88 floating point operations, and 8 branches. In Figure 18.8 with a coarsening factor of 8, each loop iteration performs 4 loads, 59 floating point operations (3 subtractions, 24 additions, 16 multiplications, 8 divisions, and 8 rsqrtf instrinsic), and 1 branch, which is a substantial reduction.
3. (1) Increased resource usage per thread which may decrease occupancy. (2) Reduction in the amount of parallelism exposed which may underutilize hardware for smaller workloads.

4. For bins at the boundary, some points may be in bounds or out of bounds for different threads causing different threads to follow different control paths.

Chapter 19: Parallel programming and computational thinking

No exercises.

Chapter 20: Programming a heterogeneous computing cluster

1.
    a. 524,288
    b.
        i. 32,768
        ii. 16,384
    c.
        i. 32,768
        ii. 32,768
    d.
        i. 491,520
        ii. 491,520
    e.
        i. 131,072
        ii. 65,536
2. (c) 4 bytes
3. a, b
4.

```
// Remove cudaMemcpyAsync
// Remove cudaMemcpyAsync
cudaStreamSynchronize(stream0);
MPI_Sendrecv(d_output + num_halo_points, num_halo_points, MPI_FLOAT,
             left_neighbor, i, d_output + right_halo_offset,
             num_halo_points, MPI_FLOAT, right_neighbor, i,
             MPI_COMM_WORLD, &status);
MPI_Sendrecv(d_output + right_stage1_offset + num_halo_points, num_halo_points,
             MPI_FLOAT, right_neighbor, i, d_output + left_halo_offset,
             num_halo_points, MPI_FLOAT, left_neighbor, i,
             MPI_COMM_WORLD, &status);
// Remove cudaMemcpyAsync
// Remove cudaMemcpyAsync
cudaDeviceSynchronize();
...
```

Chapter 21: CUDA dynamic parallelism

1. (b)
2. (b). The tree expands by a factor of 4 at each level. Therefore, the number of levels is $log_4(64)$ which is 3, plus the root level which is a total of 4 levels.
3. (a). Level 0 launches 1 kernel, level 1 launches 4 kernels, and level 2 launches 16 kernels. The total number of child kernel launches is 1+4+16 = 21.
4. False. Constant memory has to be initialized on the host.
5. False. Shared and local memories are not visible to children.
6. (c) 6. Since streams are not explicitly used, each parent thread block will use its default stream which means that all child kernels from within the same parent thread block are serialized. Since there are 6 parent thread blocks, there can be up to 6 child kernels running concurrently, one from each parent thread block.

Chapter 22: Advanced practices and future evolution
No exercises.

Chapter 23: Conclusion and outlook
No exercises.