**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 16×16 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:**

**Assumptions:**

blockDim.x = blockDim.y = TILE\_WIDTH = 4

width = 16

Number of threads = 16 \* 16

Figure 5.5.1 shows the 2D arrangement of threads. Threads of block(0,0) have been numbered.

**Figure 5.5.1 Arrangement of threads in block(0,0)**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 00 | 01 | 02 | 03 |  |  |  |  |  |  |  |  |  |  |  |  |
| 10 | 11 | 12 | 13 |  |  |  |  |  |  |  |  |  |  |  |  |
| 20 | 21 | 22 | 23 |  |  |  |  |  |  |  |  |  |  |  |  |
| 30 | 31 | 32 | 33 |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

The elements accessed by each thread of block(0,0) during their different phases of execution are given in table 5.5.2. It can be inferred that the memory access is coalesced for threads during each phase of execution.

**Table 5.5.2 Access pattern of threads in block(0,0)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Phase 0** | **Phase 1** | **Phase 2** | **Phase 3** |
| **Thread(0,0)** | M[0], N[0] | M[4], N[64] | M[8], N[128] | M[12], N[192] |
| **Thread(0,1)** | M[1], N[1] | M[5], N[65] | M[9], N[129] | M[13], N[193] |
| **Thread(0,2)** | M[2], N[2] | M[6], N[66] | M[10], N[130] | M[14], N[194] |
| **Thread(0,3)** | M[3], N[3] | M[7], N[67] | M[11], N[131] | M[15], N[195] |
| **Thread(1,0)** | M[16], N[16] | M[20],N[80] | M[24], N[144] | M[28], N[208] |
| **Thread(1,1)** | M[17], N[17] | M[21], N[81] | M[25], N[145] | M[29], N[209] |
| **Thread(1,2)** | M[18], N[18] | M[22], N[82] | M[26], N[146] | M[30], N[210] |
| **Thread(1,3)** | M[19], N[19] | M[23], N[83] | M[27], N[147] | M[31], N[211] |
| **Thread(2,0)** | M[32], N[32] | M[36], N[96] | M[40], N[160] | M[44], N[224] |
| **Thread(2,1)** | M[33], N[33] | M[37], N[97] | M[41], N[161] | M[45], N[225] |
| **Thread(2,2)** | M[34], N[34] | M[38], N[98] | M[42], N[162] | M[46], N[226] |
| **Thread(2,3)** | M[35], N[35] | M[39], N[99] | M[43], N[163] | M[47], N[227] |
| **Thread(3,0)** | M[48], N[48] | M[52], N[112] | M[56], N[176] | M[60], N[240] |
| **Thread(3,1)** | M[49], N[49] | M[53], N[113] | M[57], N[177] | M[61], N[241] |
| **Thread(3,2)** | M[50], N[50] | M[54], N[114] | M[58], N[178] | M[62], N[242] |
| **Thread(3,3)** | M[51], N[51] | M[55], N[115] | M[59], N[179] | M[63], N[243] |

**Sample calculation:**

**1. Thread(0,0)**

threadIdx.y =0; threadIdx.x=0;

row = 0\*4 =0 = 0;

col = 0\*4 +0 = 0;

phase = 0

Mds[0][0] = M[0\*16 + 0\*4 + 0] = M[0]

Nds[0][0] = N[(0\*4+0)\*16 + 0] = N[0]

phase = 1

Mds[0][0] = M[0\*16 + 1\*4 + 0] = M[4]

Nds[0][0] = N[(1\*4 + 0)\*16 + 0] = N[64]

phase = 2

Mds[0]0]= M[0\*16 + 2\*4 + 0] = M[8]

Nds[0][0] = N[(2\*4 + 0)\*16 + 0] = N[128]

phase = 3

Mds[0][0] = M[0\*16 + 3\*4 + 0] = M[12]

Nds[0][0] = N[(3\*4+0)\*16 + 0] = N[192]

**2. Thread(0,1)**

phase = 0

Mds[0][1] = M[0\*16 + 0\*4 + 1] = M[1]

Nds[0][1] = N[(0\*4+0)\*16 + 1] = N[1]

phase = 1

Mds[0][1] = M[0\*16 + 1\*4 + 1] = M[5]

Nds[0][1] = N[(1\*4+0)\*16 + 1] = N[65]

phase = 2

Mds[0][1] = M[0\*16 + 2\*4 + 1] = M[9]

Nds[0][1] = N[(2\*4+0)\*16 + 1] = N[129]

phase = 3

Mds[0][1] = M[0\*16 + 3\*4 + 1] = M[13]

Nds[0][1] = N[(3\*4+0)\*16 + 1] = N[193]

**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 + 2];

shared[tid] += shared[tid + 1];

}

**Solution:**

There are several syntax errors in the engineer’s kernel

1. The engineer uses undeclared variable shared instead of partialSum.

2. The input values have not been loaded into shared or partialSum

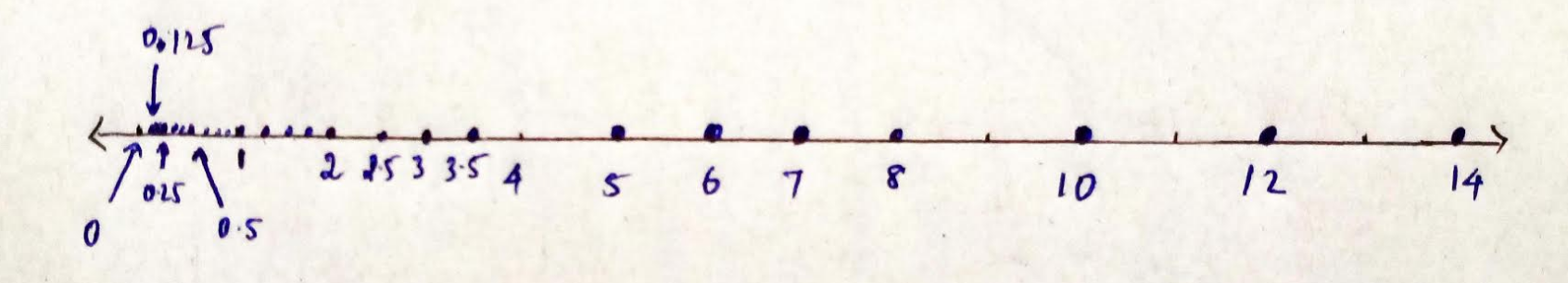
3. The purpose of variable n is unclear

4. Usage of two if statments in the kernel increases the chances of thread divergence thus impacting performance.

**Exercise 6.2**

Draw the equivalent of Fig. 6.5 for another 6-bit format (1-bit sign, 2-bit mantissa, 3-bit exponent). Use your result to explain what each additional exponent bit does to the set of representable numbers on the number line.

**Solution:**



The additional exponent bit doubles the number of major intervals of representable numbers compared to Fig. 6.5. The intervals to the left of zero are not represented in the daigram.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Exponent** | **Mantissa** | **E10** | **e** | **Value** |
| 000 | 00 | 0 | -3 | 0.125 |
|  | 01 |  |  | 0.15625 |
|  | 10 |  |  | 0.1875 |
|  | 11 |  |  | 0.21875 |
| 001 | 00 | 1 | -2 | 0.25 |
|  | 01 |  |  | 0.3125 |
|  | 10 |  |  | 0.375 |
|  | 11 |  |  | 0.4375 |
| 010 | 00 | 2 | -1 | 0.5 |
|  | 01 |  |  | 0.625 |
|  | 10 |  |  | 0.75 |
|  | 11 |  |  | 0.875 |
| 011 | 00 | 3 | 0 | 1 |
|  | 01 |  |  | 1.25 |
|  | 10 |  |  | 1.5 |
|  | 11 |  |  | 1.75 |
| 100 | 00 | 4 | 1 | 2 |
|  | 01 |  |  | 2.5 |
|  | 10 |  |  | 3 |
|  | 11 |  |  | 3.5 |
| 101 | 00 | 5 | 2 | 4 |
|  | 01 |  |  | 5 |
|  | 10 |  |  | 6 |
|  | 11 |  |  | 7 |
| 110 | 00 | 6 | 3 | 8 |
|  | 01 |  |  | 10 |
|  | 10 |  |  | 12 |
|  | 11 |  |  | 14 |

**Exercise 6.3**

Assume that in a new processor design, due to technical difficulty, the floating-point arithmetic unit that performs addition can only do “round to zero” (rounding by truncating the value toward 0). The hardware maintains sufficient number of bits that the only error introduced is due to rounding. What is the maximal ulp error value for add operations on this machine?

**Solution:**  0.5D ULP

By rounding to zero we would introduce an error that is half the place value of the least significant bit. Hence the maximal ULP error value is 0.5D ULP.

**Exercise 6.4**

A graduate student wrote a CUDA kernel to reduce a large floating-point array to the sum of all its elements. The array will always be sorted with the smallest values to the largest values. To avoid branch divergence, he decided to implement the algorithm of Fig. 5.16. Explain why this can reduce the accuracy of his results.

**Solution:**

The algorithm adds each element with another element that is blockDim.x/2 units apart. As the elements are in ascending order such an algorithm would result in adding a value with small exponent with a value with large exponent. This entails the mantissa of the element with small exponent being right shifted till the exponents of both the elements are equal. Such an approach leads to truncation of bits of the sum of elements thus reducing the accuracy.