CPSC/ECE 4780/6780

General-Purpose Computation on Graphical Processing Units (GPGPU)

Exam 1

06/25/2020

NAME: ____________Biyang Fu________________________ SCORE: ______ / 100

I. Single Choice (10 points)

1. (2 points) In the GT200 implementation, once a block is assigned to a Stream Multiprocessor (SM), it is further divided into thread warps. How many threads are within a warp? **C**
   (A) 8 threads
   (B) 16 threads
   (C) 32 threads
   (D) 64 threads

2. (2 points) If you are writing a GPU application which involves computation with a considerable amount of spatial locality in the memory access, which of the following memory type you'd choose to use? **D**
   (A) Global memory
   (B) Shared memory
   (C) Constant memory
   (D) Texture memory

3. (2 points) G80 has 16KB of shared memory per SM, and each SM accommodates up to 8 blocks. If each block has to use 3KB of shared memory for an application, no more than how many blocks can be assigned to each SM? **B**
   (A) 4 blocks
   (B) 5 blocks
   (C) 6 blocks
   (D) 7 blocks

4. (2 points) The atomicAdd() function invocation in CUDA **A**
   (A) is slow because requires synchronization
   (B) is very fast and typically its impact on the performance is negligible
   (C) can only be invoked by threads in the same warp
   (D) can probably be interrupted by another thread

5. (2 points) The use of streams in CUDA **C**
   (A) represents the means to increase the amount of memory space on the device
   (B) represents a mechanism for hiding latency related to the data movement over the PCIe bus
   (C) is the mechanism that enables the simultaneous execution of multiple kernels on the device
   (D) is the mechanism of executing a sequence of synchronous CUDA operations on a device in the order issued by the host code

II. Multiple Choices (20 points)

1. (4 points) Which of the following properties can be queried by cudaGetDeviceProperties()? **A, B, D**
   (A) major
   (B) minor
   (C) totalDevice
   (D) deviceOverlap

2. (4 points) If the total size of a block is limited to 512 threads, which of the following subdivision of a grid is allowable? **A, B, D**
   (A) (8, 16, 2)
   (B) (16, 16, 2)
   (C) (32, 32, 1)
   (D) (512, 1, 1)

3. (4 points) What commands can you use to profile your CUDA program? **A,D**
   (A) nvprof
   (B) cuda-gdb
   (C) vglrun
   (D) nvvp

4. (4 points) What types of concurrency can be enabled by asynchronous, stream-based kernel launches and data transfer? **A,B,C,D**
   (A) Overlapped host computation and device computation
   (B) Overlapped host computation and host-device data transfer
   (C) Overlapped host-device data transfer and device computation
   (D) Concurrent device computation

5. (4 points) Which statements are true about stream and event in multi-GPU programming? **A,B,C,D**
   (A) You can launch a kernel in a stream only if the device associated with that stream is the current device
   (B) You can record an event in a stream only if the device associated with that stream is the current device
   (C) You can query or synchronize any event or stream, even if they are not associated with the current device
   (D) You can issue a memory copy in any stream at any time, regardless of what device it is associated with or what the current device is

III. Short Answer (20 points)

1. (4 points) What is bank conflict? What's the cost of bank conflict when it happens?

   If multiple addresses of a memory request map to the same memory bank, it is called bank conflicts.

   And the **cost = max # of simultaneous accesses to a single bank.**

2. (4 points) What happens when threads in the same warp take different control paths? How to avoid different execution paths within the same warp?

When threads in the same warp take different paths through an application, all instructions in different branches are to be serialized, this is called warp divergence.

Partition data to ensure all threads in the same warp take the same control path, e.g., partitioning data to be a multiple of warp size in many cases.

3. (4 points) What happens if two threads assigned to different blocks write to the same memory location in global memory? How to resolve the problem in CUDA?

Race conditions arise when 2+ threads attempt to access the same memory location concurrently and at least one access is write. Programs with race conditions may produce unexpected, seemingly arbitrary results.
To resolve the problem in CUDA, we can use atomic function or atomic lock to execute the operation atomically.

4. (4 points) The CUDA peer-to-peer (P2P) APIs allow directly load and store addresses within a CUDA kernel and across GPUs, but it has to be enabled. What happens if the P2P access is not enabled?

If peer-to-peer access is not enabled between two GPUs, a peer-to-peer memory copy between those two devices will be staged through host memory, thereby reducing performance.

5. (4 points) List at least four approaches you may consider when you are trying to accelerate the performance of a slow but working GPU kernel.

1). Loop splitting: takes the body of a loop and splits it into two loops.
2). Loop interchange: first swap the inner loop and the outer loop to avoid conflicts between threads
3). Using registers to reduce memory accesses: load elements into automatic variables before the execution enters the loop, thus converting global memory accesses to register accesses.
4). Chunking data to fit into constant memory: sequentially process a chunk of the k-space elements that fit into the 64kB capacity of the constant memory.
5). Using hardware trigonometry functions: change the calls to sin and cos functions into their hardware versions __sin and __cos.

IV. Problem Solving (50 points)

1. (10 points) We would like to launch a matrix multiplication kernel to multiply an 80×96 matrix A with a 96×40 matrix B with the simple matrix multiplication kernel using 16×16 thread blocks. Answer the following questions:

   1) (6 points) How many blocks will be launched if each thread is responsible for one element? Justify.

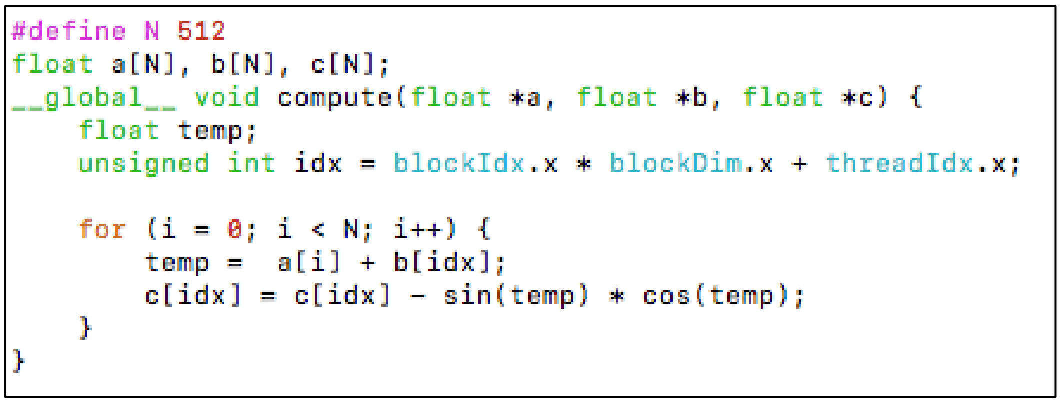
The output matrix size is 80x40. Since we need one thread to compute each output element, this is our grid size. The other dimension (96) affects the number of iterations through the dot product loop, but not the block decomposition. With 16x16 blocks, we need ceil(80/16)*ceil(40/16) = 5x3 or 15 blocks.

   2) (4 points) How many blocks if each thread is responsible for 4 elements arranged 2x2? Justify.

Assuming each thread is responsible for 4 elements arranged 2x2, we need 40x20 threads to cover the matrix. With 16x16 blocks we need ceil(40/16)*ceil(20/16) = 3x2 or 6 blocks.

2. (10 points) We have learned in Slides_05 that the compute to global memory access (CGMA) ratio is the number of floating-point calculations performed for each access to the global memory within a region of a CUDA program. It influences the highest achievable floating-point calculation throughput.

   1) (2 points) Given the following CUDA code, what is the CGMA ratio of each thread inside each iteration? Justify.



```
#define N 512
float a[N], b[N], c[N];
__global__ void compute(float *a, float *b, float *c) {
    float temp;
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;

    for (i = 0; i < N; i++) {
        temp =  a[i] + b[idx];
        c[idx] = c[idx] - sin(temp) * cos(temp);
    }
}
```

   There are 3 global memory access, and 3 floating-point calculation. So the CGMA ratio of each thread inside each iteration is 1.

   2) (6 points) Indicate for each data structure (a, b, and c) the best place for it in the memory hierarchy of a GPU (you do not need to re-write the code).

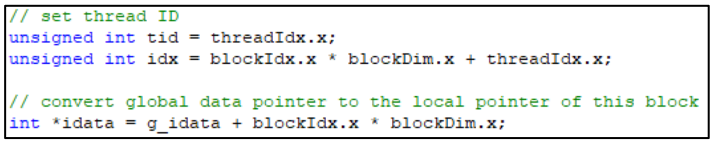
A and b should use constant memory, c should use shared memory.

   3) (2 points) How to use hardware acceleration function in this code to furtherly improve the performance if applicable?

Using hardware trigonometry functions: change the calls to sin and cos functions into their hardware versions __sin and __cos.

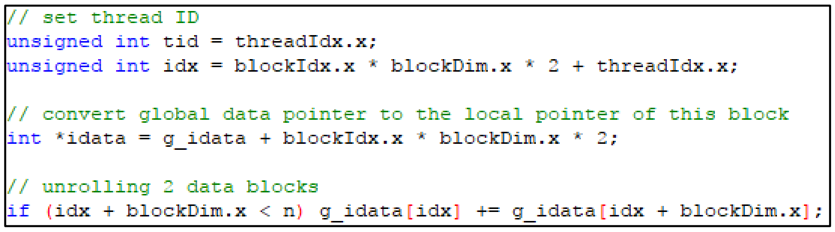
3. (10 points) In Slides_07, we discussed how to optimize a CUDA reduction code. One of the optimization was loop unrolling in which we combine *n* data blocks and replace single load with *n* loads and first add of the reduction:

Original data blocks:



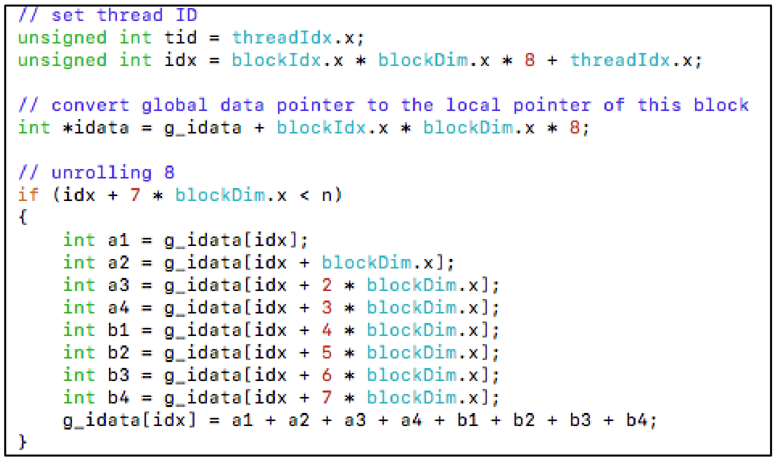
```
// set thread ID
unsigned int tid = threadIdx.x;
unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;

// convert global data pointer to the local pointer of this block
int *idata = g_idata + blockIdx.x * blockDim.x;
```

Example I: Unrolling data blocks with *n* = 2:



```
// set thread ID
unsigned int tid = threadIdx.x;
unsigned int idx = blockIdx.x * blockDim.x * 2 + threadIdx.x;

// convert global data pointer to the local pointer of this block
int *idata = g_idata + blockIdx.x * blockDim.x * 2;

// unrolling 2 data blocks
if (idx + blockDim.x < n) g_idata[idx] += g_idata[idx + blockDim.x];
```

Example II: Unrolling data blocks with *n* = 8:



```
// set thread ID
unsigned int tid = threadIdx.x;
unsigned int idx = blockIdx.x * blockDim.x * 8 + threadIdx.x;

// convert global data pointer to the local pointer of this block
int *idata = g_idata + blockIdx.x * blockDim.x * 8;

// unrolling 8
if (idx + 7 * blockDim.x < n)
{
    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;
}
```

1) (8 points) Write the code to unroll data blocks with *n* = 4.

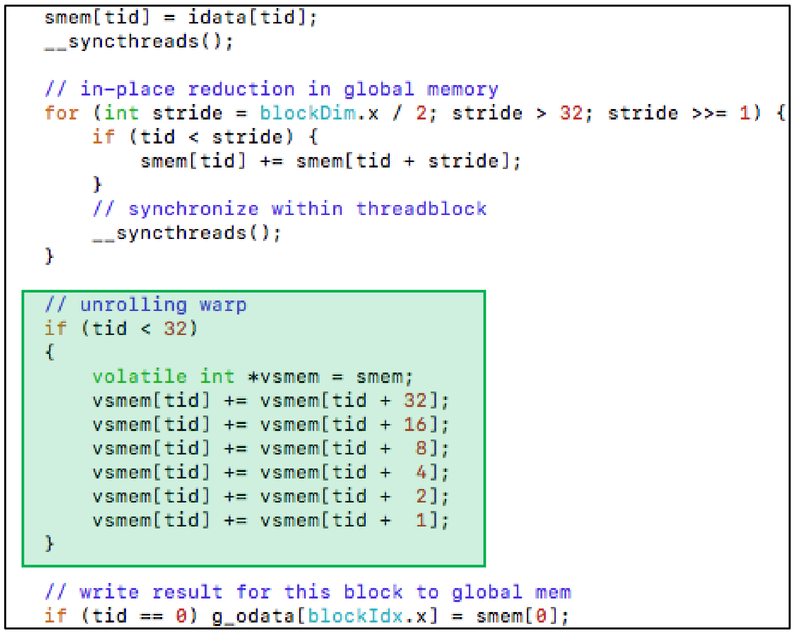
```
// set thread ID

  unsigned int tid = threadIdx.x;

  unsigned int idx = blockIdx.x * blockDim.x * 4 + threadIdx.x;



  // convert global data pointer to the local pointer of this block

  int *idata = g_idata + blockIdx.x * blockDim.x * 4;
```

```
// unrolling 8

if (idx + 3 * blockDim.x < n)

{

    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];


    g_idata[idx] = a1 + a2 + a3 + a4;

}
```

2) (2 points) If the kernel was launched with the execution configuration myKernel<<<nBlocks, nThreads>>> in the original version, what's the new execution configuration after you unroll data blocks with *n* = 4?

myKernel <<<grid.x/4, block>>>

4. (10 points) The loop unrolling implementation for Problem 3 can be completed by the following segment of code:



```
    smem[tid] = idata[tid];
    __syncthreads();

    // in-place reduction in global memory
    for (int stride = blockDim.x / 2; stride > 32; stride >>= 1) {
        if (tid < stride) {
            smem[tid] += smem[tid + stride];
        }
        // synchronize within threadblock
        __syncthreads();
    }

    // unrolling warp
    if (tid < 32)
    {
        volatile int *vsmem = smem;
        vsmem[tid] += vsmem[tid + 32];
        vsmem[tid] += vsmem[tid + 16];
        vsmem[tid] += vsmem[tid +  8];
        vsmem[tid] += vsmem[tid +  4];
        vsmem[tid] += vsmem[tid +  2];
        vsmem[tid] += vsmem[tid +  1];
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = smem[0];
```

1) (4 points) Why is __syncthreads() necessary in general?

We should use to __syncthreads() ensure all threads in a block have completed a phase of their execution of the kernel before any moves to the next phase.

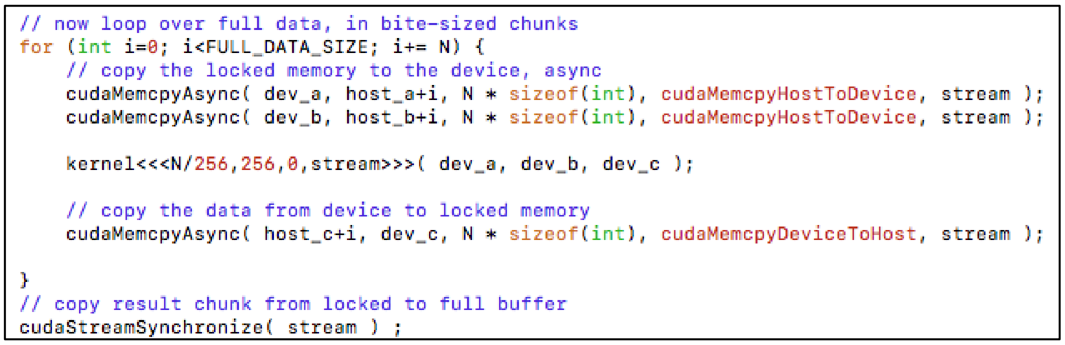
2) (4 points) Why it is safe to eliminate the call to __syncthreads() in the unrolled loop region (highlighted in green) when there are fewer than 32 active threads in a thread block?

Because

3) (2 points) What's the point of declaring *vsmem* as a volatile?

Volatile qualifier: tells the compiler that it must store vsmem[tid] back to global memory with every assignment.

5. (10 points) In Slides_09, we learned about using streams to enable grid-level concurrency for CUDA operations. Suppose we need to do some computation on a large size of data, and the computation needs to be staged in chunks because the entire buffer can't fit on the GPU at once. The code to perform this "chunkified" sequence of computations with one single stream is given below.



```
// now loop over full data, in bite-sized chunks
for (int i=0; i<FULL_DATA_SIZE; i+= N) {
    // copy the locked memory to the device, async
    cudaMemcpyAsync( dev_a, host_a+i, N * sizeof(int), cudaMemcpyHostToDevice, stream );
    cudaMemcpyAsync( dev_b, host_b+i, N * sizeof(int), cudaMemcpyHostToDevice, stream );

    kernel<<<N/256,256,0,stream>>>( dev_a, dev_b, dev_c );

    // copy the data from device to locked memory
    cudaMemcpyAsync( host_c+i, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost, stream );

}
// copy result chunk from locked to full buffer
cudaStreamSynchronize( stream ) ;
```

1) (3 points) Explain what's the major difference between cudaMemcpyAsync() and cudaMemcpy().

cudaMemcpyAsync() is asynchronous, and cudaMemcpy() is synchronous.

2) (4 points) Rewrite the loop in the code by using two streams using breadth-first assignment.

```
for (int i = 0; i < FULL_DATA_SIZE; i += N * 2) {

  cudaMemcpyAsync(dev_a0, host_a + i, N * sizeof(int),
                  cudaMemcpyHostToDevice, stream0);
  cudaMemcpyAsync(dev_a1, host_a + i + N, N * sizeof(int),
                  cudaMemcpyHostToDevice, stream1);

  cudaMemcpyAsync(dev_b0, host_b + i, N * sizeof(int),
                  cudaMemcpyHostToDevice, stream0));
  cudaMemcpyAsync(dev_b1, host_b + i + N, N * sizeof(int),
                  cudaMemcpyHostToDevice, stream1);

 kernel<<<N / 256, 256, 0, stream0>>>(dev_a0, dev_b0, dev_c0);
```

```
  kernel<<<N / 256, 256, 0, stream1>>>(dev_a1, dev_b1, dev_c1);



   cudaMemcpyAsync(host_c + i, dev_c0, N * sizeof(int),
                   cudaMemcpyDeviceToHost, stream0);
   cudaMemcpyAsync(host_c + i + N, dev_c1, N * sizeof(int),
                   cudaMemcpyDeviceToHost, stream1);
}
cudaStreamSynchronize(stream0);
cudaStreamSynchronize(stream1);
```

3) (3 points) Explain why it is better than depth-first assignment in this case.

If we use depth-first assignment in this case, we must complete first core function first, and then compute the next core function.