**ECE 451-566: Introduction to Parallel and Distributed Computing, Fall 2020**

**Instructor: Maria Striki**

**Homework 3: 2 out of 3: 25 + 25 = 50**

**Groups of >=2 members will work on 2 out of the 3 problems.**

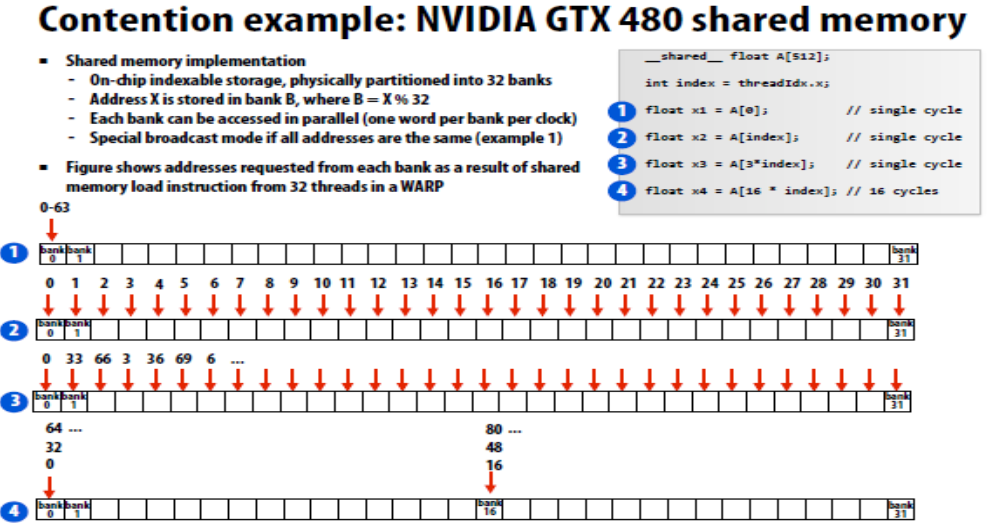
**Groups of 1 member will work on 1 out of the 3 problems.**

**Issue Date: Fri Nov 27th 2020, Due Date: Fri Dec 11th 2020 at 10.00pm**

**One submission per group.**

**PROBLEM 1: (10 points) (Bank Conflicts and Instruction Dependencies)**

**Some background or lecture refresher**: Shared memory consists of 32 banks of width 4 bytes. Element i is in bank i % 32. A bank conflict occurs when 2 threads in a warp access different elements in the same bank. Bank conflicts cause serial memory accesses rather than parallel, are bad for performance and are the “non-coalesced access” equivalent for shared memory. Note that stride 1 accesses are both conflict-free and coalesced.

****

**Fig.1** The above example illustrates addresses requested from each bank as a result of shared memory load instruction from 32 threads of a WARP.

**Part 1: (13 points)**

**Problem Description**: You are asked to multiply a 32 x 128 matrix with a 128 x 32 element matrix. This outputs a 32 x 32 matrix. We can 32 ^ 2 = 1024 threads and each thread will compute 1 output element. For the sake of simplicity let's use a single block, so:

grid shape = (1, 1, 1), block shape = (32, 32, 1).

Further assume both the left and right matrices have already been stored in shared memory and are stored in **column major format**. In column-major format the element in the *i*th row and *j*th column is accessible at lhs[i + 32 \* j] for the left hand side and rhs[i + 128 \* j] for the right hand side. This is different (opposite) than the row-major format.

This kernel will write to a variable called output stored in shared memory.

Consider the following kernel code:

int i = threadIdx.x;

int j = threadIdx.y;

for (int k = 0; k < 128; k += 2) {

output[i + 32 \* j] += lhs[i + 32 \* k] \* rhs[k + 128 \* j];

output[i + 32 \* j] += lhs[i + 32 \* (k + 1)] \* rhs[(k + 1) + 128 \* j];

}

**Question:** Are there bank conflicts in the code? If so, how many way is the bank conflict (2-way, 4-way, etc)? Or else how many clock cycles does it take to resolve it. Remember how warps are defined with respect to its coordinates: x-coordinate major order!

**Solution:**

**PART 2: (12 points)**

The below sequential computation lends itself easily for a parallel GPU implementation by partitioning the array elements into threads and blocks. Assume that all data is initially stored in global memory.

1. Indicate the partition of your selection
2. For your response above, indicate whether global memory accesses will be coalesced. Explain why or why not.

float a[1024][1024], b[1024]

for (i=1; i<1024; i++)

for (j=1; j<1024-i; j++)

b[i+j] += a[i][j];

**Solution**:

**PROBLEM 2 (25 points): Tiled Matrix Multiplication (optimization)**

**Part 1: (9 pts)**

**Optimizations-Memory-Tiling:** We wish to conduct matrix multiplication, of the below two matrices: A, B. Write CUDA host and kernel code or pseudo-code as close to the real code, where you conduct the memory multiplication from:

1. Global memory
2. Shared memory (conducting optimized tiled multiplication).

Search on how to do tiled multiplication (or look through the resources posted to the class).

If you need to partition the arrays in blocks, you may make the selection that fits your problem most appropriately. You may also have to make a transformation to the loop order.

#define N 512

float A[512][512];

float B[512][512];

for (k=0; k<N;k++) {

for (j=0; j<N; j++) {

for (i=0; i<N; i++) {

B[i][j] += A[j][k];

}

}

}

**Part 2: (16 pts)**

The CUDA program below computes the outer product of two vectors, u and v.  The outer product is a specific case of matrix multiplication in which the first matrix is a (column) vector of M elements and the second matrix is the transpose of a vector (also called a row vector) of N elements.  When we multiply an M×1 matrix by a 1×N matrix, the result is an M×N matrix, which we call the outer product of the two vectors.  The code below calculates the outer product of vector u with vector v and returns the answer as matrix A.

#define BLOCK\_DIM\_X 16

#define BLOCK\_DIM\_Y 16

\_\_global\_\_ outer\_product\_kernel (float\* u, float\* v, float\* A, unsigned int M, unsigned int N)

{

/\* perform the outer product of u and v

\* u is of size M, v is of size N, A is of size M x N

\*/

unsigned int row = blockIdx.y \* blockDim.y + threadIdx.y

unsigned int col = blockIdx.x \* blockDim.x + threadIdx.x;

if (row < M && col < N) {

A[row \* N + col] = u[row] \* v[col];

}

}

void outer\_product (float\* u, float\* v, float\* A, unsigned int M, unsigned int N)

{

dim3 blockDim (BLOCK\_DIM\_X, BLOCK\_DIM\_Y, 1);

dim3 gridDim ((N-1)/BLOCK\_DIM\_X + 1, (M-1)/BLOCK\_DIM\_X + 1, 1);

outer\_product\_kernel <<< gridDim, blockDim >>> (u, v, A, M, N);

}

**Questions:**

**Part 1: (8 points)** Rewrite the kernel to make use of tiling and shared memory.  The tile sizes should correspond to the thread block size, BLOCK\_DIM\_X wide by BLOCK\_DIM\_Y high. You should assume that both thread block dimensions are greater than 1, and that their product does not require more threads than are available in one streaming multiprocessor. You should not make other assumptions about the thread block dimensions in your code.

**Part 2: (4 points)** How many times is each element of u loaded from global memory in the original version of the code?  And in the tiled version?

**Part 3: (4 points)** How many times is each element of v loaded from global memory in the original version of the code?  And in the tiled version?

**Problem 3: GPU Memory Hierarchy (25 points)**

A GPU has the following characteristics: 8 cores running at 1GHz. Each core provides execution context for up to 128 CUDA threads. The cores use SIMD execution, running 16 consecutively numbered CUDA threads together using the same instruction stream (processor implements 16-wide “warps”, and has hence execution contexts for 8 warps/core). The cores will fetch/decode one single-precision arithmetic instruction (add, multiply, etc.) per clock. This instruction executes on an entire warp in that clock. All CUDA thread blocks on a single core cannot exceed 16 KB of shared memory storage.

**Part 1 (4 points)**: Running at peak utilization: what is the processor’s maximum throughput of single precision math operations? (consider 1 multiply of 2 single-precision numbers as one “operation”.)

**Part 2 (7 points)**: Consider a CUDA kernel launch that executes the following CUDA kernel on the processor. In this program each CUDA thread computes one element of the results array Y using 1000 elements from the input array X. Assume the program is compiled using a thread-block size of 128 threads, and enough thread blocks are created so there is exactly one thread per output array element.

\_\_global\_\_ void foo(float\* X, float\* Y) {

// get array index from CUDA block/thread id

int idx = blockIdx.x \* blockDim.x + threadIdx.x;

int input\_idx = 1000 \* idx;

float result = 0.f;

for (int i=0; i<1000; i++) { // 0 cycles (ignore arithmetic here)

float val = X[input\_idx+i]; // memory load (ignore arithmetic here)

if (((int)val / 1000) % 2 == 0) // 2 arithmetic cycles

result += doA(val); // 7 arithmetic cycles

else

result += doB(val); // 7 arithmetic cycles

}

Y[idx] = result; // memory store

}

The GPU is hooked up to a memory system that provides 32 GB/sec of bandwidth and has a memory request latency of 100 cycles. The code is run on a 128,000-element input array initialized as:

X[ ] = {1.f, 2.f, 3.f, 4.f, 5.f, 6.f, ...}.

The output array Y[ ] is only 128 elements.

You observe that the program does not attain peak performance (100% efficiency) on the GPU. What is the primary reason for the low performance? **Hint**: consider the amount of work performed by the program. (Assume the input and output arrays are resident in GPU memory at the time of the kernel launch. Transfer between host and GPU memory is not relevant in any part of this problem.)

**Part 3 (7 points)**: Run the same program on bigger output arrays of size 128×1024 and 128×1024×1024 elements. Do you observe significantly different processing throughput from the GPU on the two workloads? Why or why not? (Assume both small and large workloads fit comfortably in GPU memory.)

**Part 4 (7 points)**: This program is still not achieving 100% utilization of the processor’s execution units. What is the problem limiting performance? What percentage of max GPU throughput can you observe? **Remember**: the processor runs an instruction on an entire 16-wide warp worth of CUDA threads in a single cycle and the memory system provides 32 GB/sec of bandwidth with request latency of 100 cycles. (When a warp issues a load instruction, the data for all threads is ready in 100 cycles).

**Solution**: