LAB 6 Names in Group (max 2): Liam Lefferts
Write a program in Cuda to perform Matrix multiplication.

Square Matrix Size	No Threads Exec Time	4 Threads Exec Time	GPU Non Shared	GPU Shared
512	0.754	.209	1.189	.459
1024	5.092	1.423	9.260	3.587
2048	68.543	18.071	74.067	28.336

5.2 8*8 matrix multiplication with 2*2tiling

Block0,0	Phase1			Phase2		
thread0,0	M0,0 ↓ Mds0,0	N0,0 ↓ Nds0,0	Pvalue0,0+= Mds0,0*Nds0,0 +Mds0,1*Nds1,0	M0,2 ↓ Mds0,0	N2,0 ↓ Nds0,0	Pvalue0,0+= Mds0,0*Nds0,0 +Mds0,1*Nds1,0
thread0,1	M0,1 ↓ Mds0,1	N0,1 ↓ Nds0,1	Pvalue0,1+= Mds0,0*Nds0,1 +Mds0,1*Nds1,1	M0,3 ↓ Mds0,1	N2,1 ↓ Nds0,1	Pvalue0,1+= Mds0,0*Nds0,1 +Mds0,1*Nds1,1
thread1,0	M1,0 ↓ Mds1,0	N1,0 ↓ Nds1,0	Pvalue1,0+= Mds1,0*Nds0,0 +Mds1,1*Nds1,0	M1,2 ↓ Mds1,0	N3,0 ↓ Nds1,0	Pvalue1,0+= Mds1,0*Nds0,0 +Mds1,1*Nds1,0
thread1,1	M1,1 ↓ Mds1,1	N1,1 ↓ Nds1,1	Pvalue1,1+= Mds1,0*Nds0,1 +Mds1,1*Nds1,1	M1,3 ↓ Mds1,1	N3,1 ↓ Nds1,1	Pvalue1,1+= Mds1,0*Nds0,1 +Mds1,1*Nds1,1

Block 0,1	Phase3			Phase4		
thread0,0	M0,4 ↓ Mds0,0	N4,0 ↓ Nds0,0	Pvalue0,0+= Mds0,0*Nds0,0 +Mds0,1*Nds1,0	M0,6 ↓ Mds0,0	N6,0 ↓ Nds0,0	Pvalue0,0+= Mds0,0*Nds0,0 +Mds0,1*Nds1,0
thread0,1	M0,5 ↓ Mds0,1	N4,1 ↓ Nds0,1	Pvalue0,1+= Mds0,0*Nds0,1 +Mds0,1*Nds1,1	M0,7 ↓ Mds0,1	N6,1 ↓ Nds0,1	Pvalue0,1+= Mds0,0*Nds0,1 +Mds0,1*Nds1,1
thread1,0	M1,4 ↓ Mds1,0	N5,0 ↓ Nds1,0	Pvalue1,1+= Mds1,0*Nds0,1 +Mds1,1*Nds1,1	M1,6 ↓ Mds1,0	N7,0 ↓ Nds1,0	Pvalue1,1+= Mds1,0*Nds0,1 +Mds1,1*Nds1,1
thread1,1	M1,5 ↓ Mds1,1	N5,1 ↓ Nds1,1	Pvalue1,1+= Mds1,0*Nds0,1 +Mds1,1*Nds1,1	M1,7 ↓ Mds1,1	N7,1 ↓ Nds1,1	Pvalue1,1+= Mds1,0*Nds0,1 +Mds1,1*Nds1,1

8*8 matrix multiplication with 4*4 tiling

Block0,0	Phase1			Phase2		
thread0,0	M0,0 ↓ Mds0,0	N0,0 ↓ Nds0,0	Pvalue0,0+= Mds0,0*Nds0,0 +Mds0,1*Nds1,0 +Mds0,2*Nds2,0 +Mds0,3*Nds3,0	M0,4 ↓ Mds0,0	N4,0 ↓ Nds0,0	Pvalue0,0+= Mds0,0*Nds0,0 +Mds0,1*Nds1,0 +Mds0,2*Nds2,0 +Mds0,3*Nds3,0
thread0,1	M0,1 ↓ Mds0,1	N0,1 ↓ Nds0,1	Pvalue0,1+= Mds0,0*Nds0,1 +Mds0,1*Nds1,1 +Mds0,2*Nds2,1 +Mds0,3*Nds3,1	M0,5 ↓ Mds0,1	N4,1 ↓ Nds0,1	Pvalue0,1+= Mds0,0*Nds0,1 +Mds0,1*Nds1,1 +Mds0,2*Nds2,1 +Mds0,3*Nds3,1
thread0,2	M0,2 ↓ Mds0,2	N0,2 ↓ Nds0,2	Pvalue0,2+= Mds0,0*Nds0,2 +Mds0,1*Nds1,2 +Mds0,2*Nds2,2 +Mds0,3*Nds3,2	M0,6 ↓ Mds0,2	N4,2 ↓ Nds0,2	Pvalue0,2+= Mds0,0*Nds0,2 +Mds0,1*Nds1,2 +Mds0,2*Nds2,2 +Mds0,3*Nds3,2
thread0,3	M0,3 ↓ Mds0,3	N0,3 ↓ Nds0,3	Pvalue0,3+= Mds0,0*Nds0,3 +Mds0,1*Nds1,3 +Mds0,2*Nds2,3 +Mds0,3*Nds3,3	M0,7 ↓ Mds0,3	N4,3 ↓ Nds0,3	Pvalue0,3+= Mds0,0*Nds0,3 +Mds0,1*Nds1,3 +Mds0,2*Nds2,3 +Mds0,3*Nds3,3

Reduction in global memory traffic is proportional to the selected tile's dimensions.

By loading each global memory value into shared memory it can be used multiple times. Thus we reduce the number of accesses to the global memory by some factor. In the first case, we reduce the number of accesses to the global memory by half, the second halves that again for a reduction factor of 4.

Reduction factor is found to be N if the tiles are N x N elements.

In general, if an input matrix is of dimension N and the tile size is TILE_WIDTH, the dot product would be performed in N/TILE_WIDTH phases. The creation of these phases is key to the reduction of accesses to the global memory. With each phase focusing on a small subset of the input matrix values, the threads can collaboratively load the subset into the shared memory and use the values in the shared memory to satisfy their overlapping input needs in the phase.

5.6 Assume that a kernel is launched with 1,000 thread blocks each of which has 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?

Answer: **D 512000** Shared memory variables are allocated by thread block but kernel local variables are allocated by thread, 1000 x 512 = 512000.

5.8. Explain the difference between shared memory and L1 cache.

CUDA shared memory usage is explicit, defined by the programmer's use of the __shared__ keyword. L1 cache is "automatically" managed by hardware. Performance and caching strategy of L1 cache is hardware architecture dependent, out of the programmer's control.

```
/* printMatricesCheck
 * Prints matrices check status to stdout
 *
 * Takes:
 * float * A, B;                matrices to compare
 * int numCRows, numCColumns;    matrix dimensions
 *
 * Returns: void
 */
void printMatricesCheck(float * A, float * B, int numCRows, int numCColumns) {
   float accum = 0;
        for (int i = 0; i < numCRows * numCColumns; ++i) {
                if ((accum = abs(A[i] - B[i])) >= .1) {
                        printf("FAILED at line %d: A=%.4f B=%.4f %f", i, A[i], B[i]);
                        break;
                }
        }
   if (accum <= .1) printf("SUCCESSFUL");
        /* print MM result
        for (int i = 0; i < TILE_WIDTH * TILE_WIDTH; ++i) {
                if (i % TILE_WIDTH == 0) printf("\n");
                printf("%.2f ", B[i]);
        }*/
        printf("\n");
}
```

```
//Compute C=A*B in GPU non shared memory
__global__ void matrixMultiply(float * A, float * B, float * C, int numARows,
                int numAColumns, int numBRows, int numBColumns, int numCRows,
                int numCColumns) {
        //identify row and column to work on
        int row = blockIdx.y * blockDim.y + threadIdx.y;
        int col = blockIdx.x * blockDim.x + threadIdx.x;
        int i = row * numCColumns + col;
        float Pvalue = 0;
        int k;
        if (row < numARows && col < numBColumns) {
                for (k = 0; k < numBColumns; k++) {
                        Pvalue += A[row * numAColumns + k] * B[k * numBColumns + col];
                }
                C[i] = Pvalue;
        }
}
```

```
/* matrixMultiplyShared - Compute C = A*B in GPU shared memory
 * Takes:
 * Matrix d_A, d_B;            matrices to compute
 * Matrix d_C;                 result matrix
 * Returns: void */
__global__ void matrixMultiplyShared(Matrix d_A, Matrix d_B, Matrix d_C) {
// Each thread block computes one matrix tile, tile_C, of d_C
        Matrix tile_C;
        tile_C.numCols = tile_C.numRows = TILE_WIDTH;
        tile_C.stride = d_C.stride;
        tile_C.elements = &d_C.elements[d_C.stride * TILE_WIDTH * blockIdx.y
                        + TILE_WIDTH * blockIdx.x];
// Each thread computes one element of tile_C by accumulating results into Pvalue
        float Pvalue = 0.0;
        // Iterate over matrix tiles of d_A and d_B to compute tile_C
        // Multiply each pair of matrix tiles together and accumulate tile_C results
        for (int m = 0; m < (d_A.numCols / TILE_WIDTH); ++m) {
                Matrix tile_A; //populate tile_A
                tile_A.numCols = tile_A.numRows = TILE_WIDTH;
                tile_A.stride = d_A.stride;
                tile_A.elements = &d_A.elements[d_A.stride * TILE_WIDTH * blockIdx.y
                                + TILE_WIDTH * m];
                Matrix tile_B; //populate tile_B
                tile_B.numCols = tile_B.numRows = TILE_WIDTH;
                tile_B.stride = d_B.stride;
                tile_B.elements = &d_B.elements[d_B.stride * TILE_WIDTH * m
                                + TILE_WIDTH * blockIdx.x];
// Shared memory tile_A & tile_B
                __shared__ float shared_A[TILE_WIDTH][TILE_WIDTH];
                __shared__ float shared_B[TILE_WIDTH][TILE_WIDTH];
// Each thread loads one tile_A and tile_B element from device to shared memory
                shared_A[threadIdx.y][threadIdx.x] = tile_A.elements[threadIdx.y
                                * tile_A.stride + threadIdx.x];
                shared_B[threadIdx.y][threadIdx.x] = tile_B.elements[threadIdx.y
                                * tile_B.stride + threadIdx.x];
                __syncthreads(); // Synchronize to ensure tile initialization
                for (int tile = 0; tile < TILE_WIDTH; ++tile) //Accumulation logic
                        Pvalue += shared_A[threadIdx.y][tile] * shared_B[tile][threadIdx.x];

        __syncthreads(); // Ensures tile computations
        }
// Each thread writes one sub_C element to global device memory
        tile_C.elements[threadIdx.y * tile_C.stride + threadIdx.x] = Pvalue;
}
```