CS 4444 Parallel Computing Spring 2017

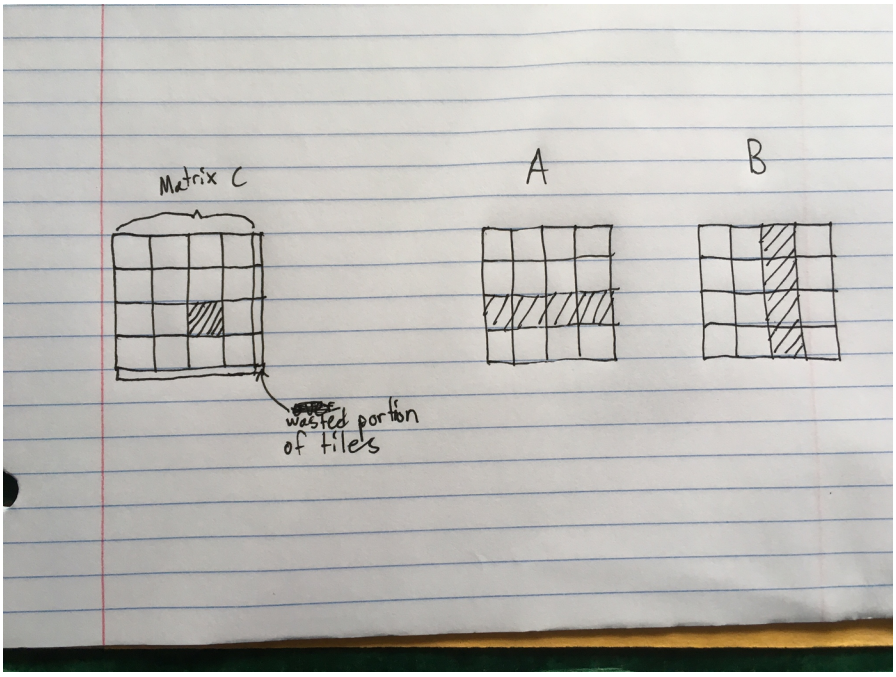
Homework 5: GPU Matrix Multiplication

Eamon Collins

When run, my code successfully multiplies two randomly initialized matrices of dimensions specified in sub.slurm and produces the product matrix. When compared to the CPU produced matrix, my code proves to be correct for dimensions 1000 to 3000 and all tested block sizes. My 3000x3000 matrix multiplication is a full 200 times faster than the naive CPU implementation, and the fastest I was ever able to multiply 10000x10000 matrices was in 17.913 seconds.

Approach:

My approach to matrix multiplication involved tiling the matrices and calculating the multiplication of small segments of matrix C at a time in order to better use shared memory. Each block takes responsibility for a single tile in C, and loops through the tiles of A and B that are necessary to calculate all the values in that tile of C, loading them into shared memory one at a time. The diagram below shows which tiles in A and B are necessary to calculate a single certain tile in C.



The leftmost tile of A is loaded into shared memory at the same time as the uppermost tile of B. The tile is a perfect square, with each dimension being the square root of the number of threads in the block. Each thread is responsible for one cell in the C matrix tile, and therefore one row in A and one column in B. Each thread multiplies the corresponding values in A and B together, accumulating the values into a single float. This accumulated value is carried over between tiles as each tile operation represents only part of the work necessary to

calculate the value of C. After all necessary tiles have been operated on, each thread places its value into the correct cell in C and the block has finished its work.

The tile dimensions are unlikely to divide perfectly into the dimensions of the overall matrix, so most of the time there will be some amount of tile-space on the right and bottom edges of the matrix that does not correspond to an existing region of C. For this space, I set the values of the shared memory matrices to 0 so that they will not affect the value of the computation, but this does add an extra factor when choosing block size, as there is some useless computation performed by each block. Therefore the best block sizes in this regard are ones where the dimension of the overall matrix is perfectly divisible by the square root of the thread count (i.e. the size of each tile dimension.)

There are other important considerations when choosing block size, but I will save that for the analysis section and for now just say how I implement it. The variable I change is a preprocessor macro for the tile dimension size. The block size varies proportionally to the square of the tile dimension, and the number of blocks is automatically adjusted based on how many blocks of that size are needed to cover the whole matrix.

Results:

All binaries used have been compiled with -O3
Speedup listed is as compared to the sequential version of the same size for matrix dimensions of 3000 and below. Other speedups are not listed
If not specified, the block size was 64 threads.

Below is a table with comparison of CPU and GPU times for matrix sizes 1000 to 3000

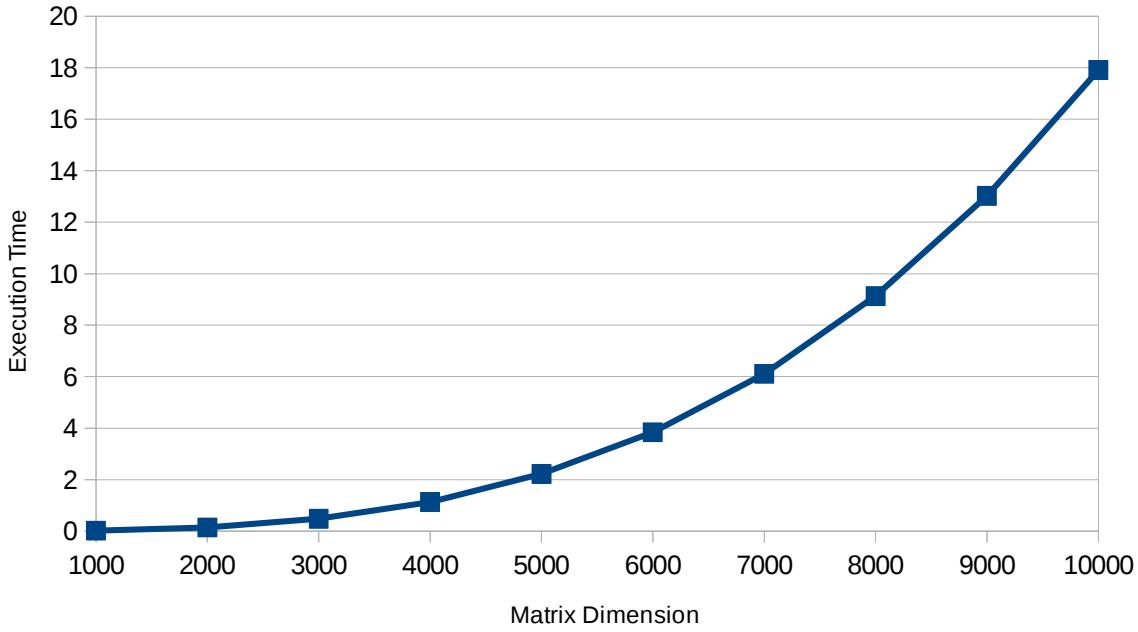
Matrix Dimension	**CPU or GPU**	**Execution Time (sec)**	**Speedup**
1000	CPU	2.898839	1
1000	GPU	.018057	160.5
2000	CPU	24.023708	1
2000	GPU	.142536	168.5
3000	CPU	96.264783	1
3000	GPU	.4800085	200.5

Below is a table detailing GPU matrix multiplication results for all matrix sizes

Matrix Dimension	**Execution Time (sec)**
1000	.018057
2000	.142536
3000	.4800085
4000	1.13596
5000	2.220956
6000	3.843874
7000	6.107615

8000	9.130169
9000	13.018716
10000	17.913

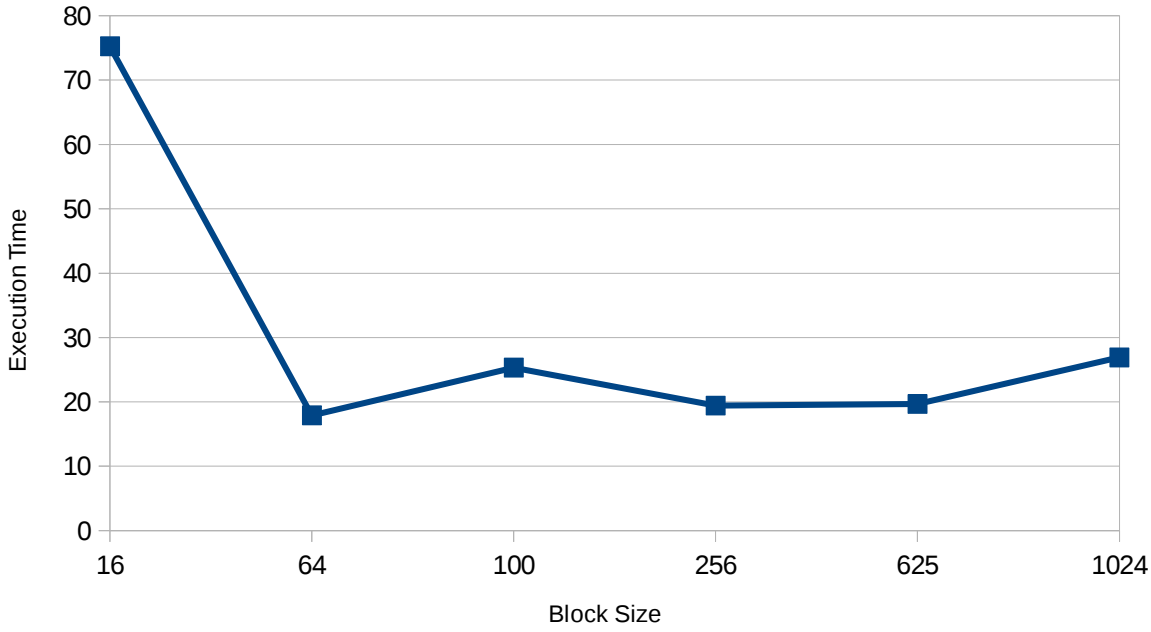
And a graph detailing the same:



Below is a table detailing performance of matrix multiplication of size 10000

Block Size (in threads/block)	**Execution Time**
16	75.243025
64	17.913
100	25.320222
256	19.425853
625	19.666022
1024	26.905952

Below is a graph detailing this same information



Analysis:

The results from the CPU and matrix size variant tests don't give me much information as I don't have other implementations to compare them to. From the CPU tests we can see the sheer amount of parallelization GPUs can provide, as a 200 times speedup is almost unheard of for the price of one chip (even at ~$2000, a K20 GPU is much less expensive than the amount of multicomputer nodes it would take to approximate the same effect, if they could even reach that.) I would loosely estimate the theoretical maximum would be around a 400 times speedup for this GPU, as it can be executing 32 instructions at a time per SM, and K20s have 13 SMs. Given that, 200 seems respectable to me. From the benchmarks of the different matrix dimensions, I can conclude that my strategy of tiling the matrices has not harmed the logarithmic complexity (not that I can conceive of a way it would.) A small amount of math shows that this curve falls almost exactly along an n^3 curve, try y= (1.8057*10^-11)*x^3, where x is the matrix dimension.

The interesting analysis comes from looking at how the block size affects performance. The important aspects to consider regarding this are the cache line size and whether the tile dimension fits perfectly into the matrix dimension.

The L1 cache line size is 128 bytes, so all reads to global memory will come in those increments. In my implementation, the maximum tile dimension is 32 floats wide, because its square is 1024, the maximum number of threads CUDA will allow in a block. 32 floats is exactly 128 bytes, so when performing well, my code should only need one memory access per row in the block. However, block sizes that create tile dimensions that don't factor into 32 can cause degradation here. For example, for block sizes of 100 (tile dimensions of 10) at the 13^{th} block the program would retrieve the 120^{th}-128^{th} bytes in one access and the 129^{th} and 130^{th} bytes would take a whole other access, doubling the access time for that block. This can explain why the block size of 100 performs so poorly, and why block sizes that correspond to tile dimensions that are factors of 32 do better, such as 256 and 64.

A block size of 16 does abysmally, attributed to the fact that you are only using half of one warp per block, which of course will do poorly.

As to why a block size of 1024 does so poorly, I believe that part of it can be attributed to the fact that it has to do some amount of extra computation due to the tiles extending over the edge. 10000 % 32 is 16, whereas both both 8 and 16 (tile dimensions of block sizes 64 and 256) divide cleanly into 10000. This means that at block sizes of 1024, the program is effectively doing the work of a matrix of size 10016 instead of 10000. However, even accounting for the fact that it's an n^3 algorithm, this is only .4% more work. So I dug deeper and came up with another likely explanation for the poor performance of block size 1024.

Shared memory is banked into 32 different banks, with each successive 32 bit word on a successive bank. Since floats are 32 bits exactly, this means that every 32 floats is on the same bank. In block sizes of 1024, my shared memory arrays have rows 32 floats long, and due to my shared memory access pattern, it is common for successive threads to access the same column one row down in the shared memory array, i.e. addresses exactly 32 floats apart. This creates a massive bank conflict, with potentially all the threads in a warp trying to access the same shared memory bank at once, serializing all their accesses. This could explain the slower speed. This causes it to be slow, even though the global access pattern may be faster, there are many more shared memory accesses per iteration, so an inefficient pattern there could slow it down substantially.

Conclusion:

Of all the programming paradigms and machines we have used in this class, GPUs are the most sensitive to the manner in which you program. The utmost care must be taken to maximize warp usage by percentage, reduce global memory accesses, and account for the unintuitive way shared memory accesses work.

RELEVANT FILES:

mmm.cu – main code for my matrix multiply, currently has sequential testing commented out for benchmarking at higher dimension levels.
sub.slurm – submission script, specify the dimension of the multiply here
Makefile – my makefile to make the mmm binary.
Output – directory with my output files from benchmarking

HOW TO RUN MY CODE:

log onto cs compute cluster
set block size by changing the tileDim preprocessor macro in mmm.cu to the square root of how many threads you want per block (maximum tileDim=32)
make
set dimension size in sub.slurm
sbatch sub.slurm
check output file for timing

MMM.CU

```
#include<stdio.h>
#include<sys/time.h>
#include<stdlib.h>
#include<iostream>
#include<math.h>

using namespace std;

//---------------------------------- Structures and Globals----------------------------------------
#define tileDim 32

typedef struct {
	int dimension1;
	int dimension2;
} ArrayMetadata2D;

// metadata variables describing dimensionalities of all data structures involved in the computation
ArrayMetadata2D A_MD, B_MD, C_MD;
// pointers for input and output arrays in the host memory
float *A, *B, *C, *C_CPU;
// pointers for input and output arrays in the device memory (NVIDIA DRAM)
float *A_GPU, *B_GPU, *C_GPU;
//dimension of each subarray (also called tiles.) All subarrays are square and equal in size. Should be the square root of threads per block for best occupancy

//---------------------------------- host function definitions ---------------------------------------

void allocateAndInitializeAB();
void computeCpuMMM();
void copyMatricesToGPU();
void copyResultFromGPU();
void compareHostAndGpuOutput();
void die(const char *error);
void check_error(cudaError e);

//---------------------------------- CUDA function definitions ---------------------------------------
// A GPU kernel that computes the vector sum A + B
__global__ void matrix_mult(float *A, float *B, float *C, int dim) {

	int i, j;
	// determine the index of the thread among all GPU threads
	int blockId = blockIdx.x;
	//I don't know if the grid info supplied by CUDA is in registers or what, but based on the sample code above
	//I'm explicitly placing them there just in case. I figure I have 64 registers per thread when using 1024 threads/block, might as well use them
	//int blockDim = blockDim.x;
```

```
	//int threadId = blockId * blockDim.x + threadIdx.x;
	int threadId = threadIdx.x;
	//int threadCount = gridDim.x * blockDim.x;
	//assumes multiplication of square matrices
	int totalDim = dim;
	int totalCells = totalDim * totalDim;
	//number of tiles in each row/column of the C matrix (and therefore the other two matrices)
	int tiles_per_dim =(totalDim / tileDim) +1;
	//index of the top left corner of the tile represented by this block in the original C array
	int tileCorner = (blockId / tiles_per_dim * tileDim * totalDim) + (blockId % tiles_per_dim * tileDim);
	//declaring the subarrays for tiles of A and B in shared memory
	__shared__ float s_A[tileDim*tileDim], s_B[tileDim*tileDim];
	//accumulator variable for the value this thread will place into it's designated spot in C
	float c_acc = 0.0f;
	//offset for the cell transferred to shared memory by this thread navigating from the current tile corner (which is calculated in loop)
	//differences between the matrices here (and in the corner calculation) are intended to get A by rows and B by columns such that
	//B is effectively transposed tile by tile
	int a_off = (threadId / tileDim) * totalDim + (threadId % tileDim);
	int b_off = (threadId % tileDim) * totalDim + threadId / tileDim;
	//rows of each tile that the thread in question will be responsible for multiplying together
	//this ensures that each thread has the correct unique combination of one row from a and one row from b
	int a_row = threadId / tileDim;
	int b_row = threadId % tileDim;
	for(i = 0; i < tiles_per_dim; i++){
		//top left corners for the current operating tile of A and B arrays
		int Acorner = (blockId / tiles_per_dim) * tileDim * totalDim + i * tileDim;
		int Bcorner = i * tileDim * totalDim + blockId % tiles_per_dim * tileDim;
		//detects if the current threads responsibility is on the grid, ie not in the part of a tile that extends past the boundaries
		//of the matrix  threadId
		int a_ongrid = (((i != tiles_per_dim-1) || (threadId % tileDim < totalDim % tileDim)) && ((Acorner + a_off) < totalCells)) ? 1 : 0;
		int b_ongrid = (((blockId % tiles_per_dim != tiles_per_dim-1) || (threadId < (totalDim % tileDim) * tileDim)) && ((Bcorner + b_off) < totalCells)) ? 1 : 0;
	//	printf("%d:  %d + %d  a_ongrid: %d\n",blockId, Acorner, a_off, a_ongrid);
		//tiles on the edge will go over the bounds of the matrix a small amount, in cells where that happens
		//I simply set them equal to 0 so they don't affect the computation
		if(a_ongrid)
			s_A[threadId] = A[Acorner + a_off];
		else
			s_A[threadId] = 0.0f;
		if(b_ongrid)
			s_B[threadId] = B[Bcorner + b_off];
```

```
		else
			s_B[threadId] = 0.0f;

		//synchronize so that no thread is operating on the tiles before they are properly
initialized
		__syncthreads();
		for(j = 0; j < tileDim; j++){
			c_acc += (s_A[a_row*tileDim + j] * s_B[b_row*tileDim + j]);
		}
		//sync so that one warp doesn't start changing the data as other warps are using
it to calculate
		__syncthreads();
	}
	//set the correct cell in C to be equal to the accumulated value
	//when using the correct tileCorner according to the blockId, a_off will actually also be
the correct C offset.
	//if statement because it is possible this is one of the tiles that extend past the border of
the matrix
	if((tileCorner + a_off < totalCells) && ((blockId % tiles_per_dim != tiles_per_dim-1) ||
(threadId % tileDim < totalDim % tileDim)))
		C[tileCorner + a_off] = c_acc;
}


//---------------------------------------------------------------------------------------------
int main(int argc, char **argv) {

	A_MD.dimension1 = (argc > 1) ? atoi(argv[1]) : 100;
	A_MD.dimension2 = (argc > 2) ? atoi(argv[2]) : A_MD.dimension1;
	B_MD.dimension1 = (argc > 3) ? atoi(argv[3]) : A_MD.dimension2;
	B_MD.dimension2 = (argc > 4) ? atoi(argv[4]) : B_MD.dimension1;
	C_MD.dimension1 = A_MD.dimension1;
	C_MD.dimension2 = B_MD.dimension2;

	printf("Matrix A is %d-by-%d\n", A_MD.dimension1, A_MD.dimension2);
	printf("Matrix B is %d-by-%d\n", B_MD.dimension1, B_MD.dimension2);
	printf("Matrix C is %d-by-%d\n", C_MD.dimension1, C_MD.dimension2);

	allocateAndInitializeAB();

	// matrix matrix multiplication in the CPU
	double elapsed;
/**	clock_t start = clock();
	computeCpuMMM();
	clock_t end = clock();
      elapsed = (end - start) / (double) CLOCKS_PER_SEC;
      printf("Computation time in the CPU: %f seconds\n", elapsed);
	fflush(stdout);
**/
```

```
	copyMatricesToGPU();

	int threads_per_block = tileDim*tileDim;
	int num_blocks = ((C_MD.dimension1 / tileDim) + 1) * ((C_MD.dimension1 / tileDim) +
1);
	clock_t gstart = clock();
	matrix_mult<<<num_blocks, threads_per_block>>>(A_GPU, B_GPU, C_GPU,
C_MD.dimension1);
	cudaThreadSynchronize();
	clock_t gend = clock();
	elapsed = (gend - gstart) / (double) CLOCKS_PER_SEC;
      printf("Computation time in the GPU: %f seconds\n", elapsed);

	copyResultFromGPU();
//	compareHostAndGpuOutput();
	return 0;
}

// allocate and initialize A and B using a random number generator
void allocateAndInitializeAB() {

	size_t sizeofA = A_MD.dimension1 * A_MD.dimension2 * sizeof(float);
	A = (float*) malloc(sizeofA);

	srand(time(NULL));
	for (int i = 0; i < A_MD.dimension1; i++) {
		for (int j = 0; j < A_MD.dimension2; j++) {
			int index = i * A_MD.dimension2 + j;
			A[index] = (rand() % 1000) * 0.001;
		}
	}

	size_t sizeofB = B_MD.dimension1 * B_MD.dimension2 * sizeof(float);
	B = (float*) malloc(sizeofB);
	for (int i = 0; i < B_MD.dimension1; i++) {
		for (int j = 0; j < B_MD.dimension2; j++) {
			int index = i * B_MD.dimension2 + j;
			B[index] = (rand() % 1000) * 0.001;
		}
	}
}

// allocate memory in the GPU for all matrices, and copy A and B content from the host CPU
memory to the GPU memory
void copyMatricesToGPU() {

	size_t sizeofA = A_MD.dimension1 * A_MD.dimension2 * sizeof(float);
	check_error(cudaMalloc((void **) &A_GPU, sizeofA));
	check_error(cudaMemcpy(A_GPU, A, sizeofA, cudaMemcpyHostToDevice));
```

```
	size_t sizeofB = B_MD.dimension1 * B_MD.dimension2 * sizeof(float);
	check_error(cudaMalloc((void **) &B_GPU, sizeofB));
	check_error(cudaMemcpy(B_GPU, B, sizeofB, cudaMemcpyHostToDevice));

	size_t sizeofC = C_MD.dimension1 * C_MD.dimension2 * sizeof(float);
	check_error(cudaMalloc((void **) &C_GPU, sizeofC));
}

// copy results from C_GPU which is in GPU card memory to C_CPU which is in the host CPU
for result comparison
void copyResultFromGPU() {
	size_t sizeofC = C_MD.dimension1 * C_MD.dimension2 * sizeof(float);
	C_CPU = (float*) malloc(sizeofC);
	check_error(cudaMemcpy(C_CPU, C_GPU, sizeofC, cudaMemcpyDeviceToHost));
}

// do a straightforward matrix-matrix multiplication in the CPU
// notice that this implementation can be massively improved in the CPU by doing proper
cache blocking but we are
// not providing you the efficient CPU implementation as that reveals too much about the ideal
GPU implementation
void computeCpuMMM() {

	// allocate the result matrix for the CPU computation
	size_t sizeofC = C_MD.dimension1 * C_MD.dimension2 * sizeof(float);
	C = (float*) malloc(sizeofC);

	// compute C[i][j] as the sum of A[i][k] * B[k][j] for all columns k of A
	for (int i = 0; i < A_MD.dimension1; i++) {
		int a_i = i * A_MD.dimension2;
		int c_i = i * C_MD.dimension2;
		for (int j = 0; j < B_MD.dimension2; j++) {
			int c_index = c_i + j;
			C[c_index] = 0;
			for (int k = 0; k < B_MD.dimension1; k++) {
				int a_index = a_i + k;
				int b_index = k * B_MD.dimension2 + j;
				C[c_index] += A[a_index] * B[b_index];
			}
		}
	}
}

// function to determine if the GPU computation is done correctly by comparing the output
from the GPU with that
// from the CPU
void compareHostAndGpuOutput() {
	int totalElements = C_MD.dimension1 * C_MD.dimension2;
```

```
	int missmatchCount = 0;
	for (int i = 0; i < totalElements; i++) {
		if (fabs(C[i] - C_CPU[i]) > 0.01) {
			missmatchCount++;
			printf("mismatch at index %i: %f\t%f\n", i, C[i], C_CPU[i]);
		}
	}
	if (missmatchCount > 0) {
		printf("Computation is incorrect: outputs do not match in %d indexes\n",
missmatchCount);
	} else {
		printf("Computation is correct: CPU and GPU outputs match\n");
	}
}

// Prints the specified error message and then exits
void die(const char *error) {
	printf("%s", error);
	exit(1);
}

// If the specified error code refers to a real error, report it and quit the program
void check_error(cudaError e) {
	if (e != cudaSuccess) {
		printf("\nCUDA error: %s\n", cudaGetErrorString(e));
		exit(1);
	}
}
```

SUB.SLURM

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --output="outputtest"
#SBATCH --exclusive
#SBATCH --nodelist=artemis2
#SBATCH --gres=gpu
#SBATCH --time=00:30:00
#SBATCH --account=parallelcomputing

dim=10000
srun mmm $dim $dim $dim $dim
```