1. A matrix addition takes two input matrices B and C and produces one output matrix A. Each element of the output matrix A is the sum of the corresponding elements of the input matrices B and C, that is, A[i][j] == B[i][j] + C[i][j]. For simplicity, we will only handle square matrices of which the elements are single-precision floating-point numbers. Write a matrix addition kernel and the host stub function that can be called with four parameters: pointer to the output matrix, pointer to the first input matrix, pointer to the second input matrix, and the number of elements in each dimension. Use the following instructions.
2. Write the host stub function by allocating memory for the input and output matrices, transferring input data to the device, launch the kernel, transferring the output data to host, and freeing the device memory for the input and output data. Leave the execution configuration parameters open for this step.

size\_t size = amount\*amount\*sizeof(float);

double \*h\_a = (double\*)malloc(size);

double \*h\_b = (double\*)malloc(size);

double \*h\_c = (double\*)malloc(size);

double \*d\_a, d\_b, d\_c;

cudaMalloc((void \*\*)&d\_a, size);

cudaMalloc((void \*\*)&d\_b, size);

cudaMalloc((void \*\*)&d\_c, size);

cudaMemcpy(d\_a, h\_a, size, cudaMemcpyHostToDevice);

cudaMemcpy(d\_b, h\_b, size, cudaMemcpyHostToDevice);

VectorAddKernel<<<uhhhhhhh>>>(d\_a, d\_b, d\_c, amount);

cudaMemcpy(h\_c, d\_c, size, cudaMemcpyDeviceToHost);

cudaFree(d\_a);

cudaFree(d\_b);

cudaFree(d\_c);

1. Write a kernel that has each thread producing one output matrix element. Fill in the execution configuration parameters for the design.

<<<ceil(amount/1024), 1024>>>

\_\_global\_\_ void VectorAddKernel(double\* a, double\* b, double\* c, size\_t amount){

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

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

    if(col < amount && row < amount)

        a[row\*amount+col] = b[row\*amount+col] + c[row\*amount+col];

}

1. Write a kernel that has each thread producing one output matrix row. Fill in the execution configuration parameters for the design.

<<<ceil(amount/1024), 1024>>>

\_\_global\_\_ void VectorAddKernel(double\* a, double\* b, double\* c, size\_t amount){

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

    for(int col=0; col < amount; col++){

        int index = i\*amount + j;

        c[index] = a[index] + b[index];

    }

}

1. Write a kernel that has each thread producing one output matrix column. Fill in the execution configuration parameters for the design.

<<<ceil(amount/1024), 1024>>>

\_\_global\_\_ void VectorAddKernel(double\* a, double\* b, double\* c, int amount){

    int row = blockIdx.y \* blockDim.x + threadIdx.y;

    if(row < amount)

        for(int col=0; col<amount; col++)

            a[col][row] = b[col][row] + c[col][row];

}

1. Analyze the pros and cons of each preceding kernel design.

When the kernel produces the columns and rows, it is more effective than when the kernel has each thread produce the individual elements. This is due to the threads working on the rows and columns working with adjacent data in their warp

1. A matrix-vector multiplication takes an input matrix B and a vector C and produces one output vector A. Each element of the output vector A is the dot product of one row of the input matrix B and C, that is, A[i] = ∑j B[i][j] + C[j]. For simplicity, we will only handle square matrices of which the elements are single-precision floating-point numbers. Write a matrix-vector multiplication kernel and the host stub function that can be called with four parameters: pointer to the output vector, pointer to the input matrix, pointer to the input vector, and the number of elements in each dimension.

\_\_global\_\_ void VectorMulKernel(double\* a, double\* b, double\* c, int amount){

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

    long sum = 0.0f;

    if(col < amount){

        for(int row=0; row<amount; row++)

            sum += b[col\*amount+row] \* c[col];

        a[col] = sum[col]

    }

}

void host\_stub(double a, double b, double c, int amount){

    double \*d\_a;

    double \*d\_b;

    double \*d\_c;

    size\_t size = amount\*sizeof(float);

    cudaMalloc((void \*\*)&d\_a, size);

    cudaMalloc((void \*\*)&d\_b, size);

    cudaMalloc((void \*\*)&d\_c, size);

    cudaMemcpy(d\_a, a, size, cudaMemcpyHostToDevice);

    cudaMemcpy(d\_b, b, size, cudaMemcpyHostToDevice);

    VectorAddKernel<<<uhhhhhhh>>>(d\_a, d\_b, d\_c, amount);

    cudaMemcpy(a, d\_a, size, cudaMemcpyDeviceToHost);

    cudaFree(d\_a);

    cudaFree(d\_b);

    cudaFree(d\_c);

    return 0;

}

1. Let’s assume that a CUDA device’s SM (streaming multiprocessor) can take up to 1536 threads and up to 4 thread blocks. Which of the following block configuration would result in the most number of threads in the SM?

(A) 128 threads per block

(B) 256 threads per block

(C) 512 threads per block

(D) 1024 threads per block

1. For a vector addition, assume that the vector length is 2000, each thread calculates one output element, and the thread block size is 512 threads. How many threads will be in the grid?

(A) 2000

(B) 2024

(C) 2048

(D) 2096

1. With respect to the previous question, how many warps do you expect to have divergence due to the boundary check on vector length?

(A) 1

(B) 2

(C) 3

(D) 6

1. You need to write a kernel that operates on an image of size 400x900 pixels. You would like to assign one thread to each pixel. You would like your thread blocks to be square and to use the maximum number of threads per block possible on the device (your device has compute capability 3.0). How would you select the grid dimensions and block dimensions of your kernel?

Total Pixels/threads = 360,000

Max # of threads/block = 1024 = 32x32

Grid x = 400/32 = 13

Grid y = 900/32 = 29

To maximize the number of threads per block on the device you would find the maximum number of threads the block can handle and find the grid size (powers of 2) that closely matches that.

For example if the SM can only handle a total of 1300 threads, then a block size of 32x32 will only lead to 1024 threads being used as a block can not be smaller than its set grid size. You can’t mix and match grid sizes. Additionally, you can’t have too small of a grid size as blocks consume registers, which is limited.

If we were given a mystical GPU of unlimited registers, then the GPU can utilize the maximum grid size of 13x29 of block size 32x32

1. With respect to the previous question, how many idle threads do you expect to have?

Given the mystical GPU, the number of idle threads are

Pixels Computed = 1024\*13\*28 = 372,736

Idle Threads = 372,736 – 360,000 = 12,736 idle threads