CUDA Code process:

Kernel:

1. Make kernel function. Example:

global void kernel(int *a, int *b, int *c, int width, int n)

2. Specify index in function OR rowIndex and colIndex

3. Write the operation to be parallelized using the indices written in Step 2

Main:

1. Make host copies of variables to be used. Example:

int *a, *b, *c;

2. Make device copies to then copy the values of the host copies. Example:

int *d_a, *d_b, *d_c;

3. Specify int nb (total dimensions). Example:

int nb = N * N;

4. Get size using "nb". Example:

int size = nb * sizeof(int)

5. Allocate space for device copies. Example:

cudaMalloc((void **)&d_a, size)

6. Allocate space for host copies and setup input values. Example:

```
a = (int *)malloc(size); random_ints(a, nb);
```

7. Copy host values to device values. Example:

```
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
```

8. Call kernel function. Example:

```
dim3 grid(N, N);
kernel<<<grid, 1>>>(d_a, d_b, d_c);
```

9. Copy result to host. Example:

```
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
```

10. Cleanup using free() of both host and device copies. Example:

```
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
```

Addition Operation:

```
int rowIndex = threadIdx.y;
int colIndex = threadIdx.x;

c[rowIndex][colIndex] = a[rowIndex][colIndex] + b[rowIndex][colIndex];
```

Addition Operation with width:

```
rowIndex = (blockIdx.y * blockDim.y + threadIdx.y) * width;
colIndex = (blockIdx.x * blockDim.x + threadIdx.x) * width;

#Traversing every thread which is in itself a matrix of threads
#1st "for" = rows
#2nd "for" = columns

for (int i=;0 i < width; i++)
for (int j=0; j < width; j++)
c[rowIndex + i][colIndex + j] =
a[rowIndex + i][colIndex + j] +
b[rowIndex + i][colIndex + j];
```

Matrix Product operation:

```
for (int k=0; k < n; k++)
c[rowIndex][colIndex] += a[rowIndex][k] * b[k][colIndex];
```

Matrix Product operation with width:

```
rowIndex = (blockIdx.y * blockDim.y + threadIdx.y) * width;
colIndex = (blockIdx.x * blockDim.x + threadIdx.x) * width;

#Traversing every thread which is in itself a matrix of threads
#1st "for" = rows
#2nd "for" = columns
#3rd "for" = matrix product

for (int i=rowIndex; i < rowIndex + width; i++)
for (int j=colIndex; j < colIndex + width; j++)
for (int k=0; k < n; k++)
c[i][j] += a[i][k] * b[k][j];
```

Case 1:

(N,1)

Index: blockIdx.x

Case 2:

(1,N)

Index: threadIdx.x

Case 3:

(N,N)

Index = threadIdx.x + blockIdx.x * blockDim.x

Case 4:

(N,N) but with decimals

Index = threadIdx.x + blockIdx.x * blockDim.x

int num_blocks = (num_of_all_threads + THREADS_PER_BLOCK-1) / THREADS_PER_BLOCK

^ Note: remember that int automatically rounds down

Main call = kernel<<<int_num_blocks, THREADS_PER_BLOCK>>>(N)

Must type if statement before statement to be parallelized:

if(index < N):

  statement

Case 5:

dim3 grid (3,2);

• kernel<<<grid, 1>>>(…);

• int index = blockIdx.x + blockIdx.y * gridDim.x;

Case 6:

• dim3 threads(5,3);

• kernel<<1, threads>>(…);

• int index = threadIdx.x + threadIdx.y * blockDim.x;

Case 7:

• dim3 grid(3,2);

• dim3 block(5,3);

• kernel<<<grid, block>>>(…);

• int index =

(blockIdx.x + blockIdx.y * gridDim.x)

* (blockDim.x * blockDim.y)

+ (threadIdx.x + threadIdx.y * blockDim.x);

Case 8:

Matrix Addition

```
int nb = N*N;
dim3 block(N, N);
kernel<<<1, block>>>(…);

int rowIndex = threadIdx.y;
int colIndex = threadIdx.x;
```

Operation:

```
c[rowIndex][colIndex] = a[rowIndex][colIndex] + b[rowIndex][colIndex];
```

Case 9:

Matrix Addition

```
int nb = N*N;
dim3 grid(N, N);
kernel<<<grid, 1>>>(…);

int rowIndex = blockIdx.y;
int colIndex = blockIdx.x;
```

Operation:

```
c[rowIndex][colIndex] = a[rowIndex][colIndex] + b[rowIndex][colIndex];
```

Case 10:

Matrix Addition

```
int nb = N*N*N*N;

dim3 grid(N, N);

dim3 block(N, N);
```

```
add<<<grid,block>>>(d_a, d_b, d_c);
```

```
rowIndex = blockIdx.y * blockDim.y + threadIdx.y

colIndex = blockIdx.x * blockDim.x + threadIdx.x
```

Operation:

```
c[rowIndex][colIndex] = a[rowIndex][colIndex] + b[rowIndex][colIndex];
```

Case 11:

Matrix Addition (with width)

```
int nb = N*N*N*N*width*width;

dim3 grid(N, N);

dim3 block(N, N);
```

```
add<<<grid,block>>>(d_a, d_b, d_c, width);
```

```
rowIndex = (blockIdx.y * blockDim.y + threadIdx.y) * width;

colIndex = (blockIdx.x * blockDim.x + threadIdx.x) * width;
```

Operation:

```
#Traversing every thread which is in itself a matrix of threads

#1st "for" = rows

#2nd "for" = columns
```

```
for (int i=;0 i < width; i++)

for (int j=0; j < width; j++)

c[rowIndex + i][colIndex + j] =

a[rowIndex + i][colIndex + j] +

b[rowIndex + i][colIndex + j];
```

Matrix Product (2 cases):

Case 12:

Matrix Product (without width)

```
int nb = N*N*N*N;

dim3 grid(N, N);

dim3 block(N, N);

product<<<grid,block>>>(d_a, d_b, d_c, N * N);

rowIndex = blockIdx.y * blockDim.y + threadIdx.y;

colIndex = blockIdx.x * blockDim.x + threadIdx.x;
```

Operation:

```
#Matrix product

for (int k=0; k < n; k++)

c[rowIndex][colIndex] += a[rowIndex][k] * b[k][colIndex];
```

Case 13:

Matrix Product (with width)

```
int nb = N*N*N*N*width*width;

dim3 grid(N, N);

dim3 block(N, N);

product<<<grid,block>>>(d_a, d_b, d_c, width, N * N * width );

rowIndex = (blockIdx.y * blockDim.y + threadIdx.y) * width;

colIndex = (blockIdx.x * blockDim.x + threadIdx.x) * width;
```

Operation:

```
#Traversing every thread which is in itself a matrix of threads

#1st "for" = rows

#2nd "for" = columns

#3rd "for" = matrix product

for (int i=rowIndex; i < rowIndex + width; i++)

for (int j=colIndex; j < colIndex + width; j++)

for (int k=0; k < n; k++)

c[i][j] += a[i][k] * b[k][j];
```
