CUDA Thread Basics
Hello World v.4.0: Vector Addition

```c
#define N 256
#include <stdio.h>

global void vecAdd (int *a, int *b, int *c);

int main() {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

    // initialize a and b with real values (NOT SHOWN)
    size = N * sizeof(int);
    cudaMemcpy(dev_a, a, size,cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, size,cudaMemcpyHostToDevice);

    __global__ void vecAdd (int *a, int *b, int *c) {
        int i = threadIdx.x;
        c[i] = a[i] + b[i];
    }

    cudaMalloc((void**)&dev_a, size);
    cudaMemcpy(dev_b, b, size,cudaMemcpyHostToDevice);
    vecAdd<<<1,N>>>(dev_a,dev_b,dev_c);
    cudaMemcpy(c, dev_c, size,cudaMemcpyDeviceToHost);

    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);
    exit (0);
}
```

- CUDA gives each thread a unique ThreadID to distinguish between each other even though the kernel instructions are the same.
- In our example, in the kernel call the memory arguments specify 1 block and N threads.
NVIDIA GPU Memory Hierarchy

- Grids map to GPUs
- Blocks map to the MultiProcessors (MP)
- Threads map to Stream Processors (SP)
- Warps are groups of (32) threads that execute simultaneously
In a NVIDIA GTX 480:

- Maximum number of threads per block: 1024
- Maximum sizes of x-, y-, and z- dimensions of thread block: 1024 x 1024 x 64
- Maximum size of each dimension of grid of thread blocks: 65535 x 65535 x 65535
Defining Grid/Block Structure

• Need to provide each kernel call with values for two key structures:
  • Number of blocks in each dimension
  • Threads per block in each dimension

• myKernel<<< B, T >>>(arg1, ...);

• B – a structure that defines the number of blocks in grid in each dimension (1D or 2D).
• T – a structure that defines the number of threads in a block in each dimension (1D, 2D, or 3D).
1D Grids and/or 1D Blocks

• If want a 1-D structure, can use a integer for B and T in:
• myKernel<<< B, T >>>(arg1, ...);
• B – An integer would define a 1D grid of that size
• T – An integer would define a 1D block of that size

• Example: myKernel<<< 1, 100 >>>(arg1, ...);
• **blockIdx.x, blockIdx.y, blockIdx.z** are built-in variables that returns the block ID in the x-axis, y-axis, and z-axis of the block that is executing the given block of code.

• **threadIdx.x, threadIdx.y, threadIdx.z** are built-in variables that return the thread ID in the x-axis, y-axis, and z-axis of the thread that is being executed by this stream processor in this particular block.

• **blockDim.x, blockDim.y, blockDim.z** are built-in variables that return the “block dimension” (i.e., the number of threads in a block in the x-axis, y-axis, and z-axis).

• So, you can express your collection of blocks, and your collection of threads within a block, as a 1D array, a 2D array or a 3D array.

• These can be helpful when thinking of your data as 2D or 3D.

• The full global thread ID in x dimension can be computed by:
  • \[ x = blockIdx.x \times blockDim.x + threadIdx.x; \]
Thread Identification Example: x-direction

Assume a hypothetical 1D grid and 1D block architecture: 4 blocks, each with 8 threads.

For Global Thread ID 26:
- \( \text{gridDim.x} = 4 \times 1 \)
- \( \text{blockDim.x} = 8 \times 1 \)
- Global Thread ID = \( \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x} \)
- = \( 3 \times 8 + 2 = 26 \)
Vector Addition Revisited

```c
#define N 1618
#define T 1024 // max threads per block
#include <stdio.h>

__global__ void vecAdd (int *a, int *b, int *c);

int main() {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

    // initialize a and b with real values (NOT SHOWN)
    size = N * sizeof(int);
    cudaMalloc((void**)&dev_a, size);
    cudaMemcpy(dev_a, a, size,cudaMemcpyHostToDevice);
    cudaMalloc((void**)&dev_b, size);
    cudaMemcpy(dev_b, b, size,cudaMemcpyHostToDevice);
    vecAdd<<<(int)ceil(N/T),T>>>(dev_a,dev_b,dev_c);
    cudaMemcpy(c, dev_c, size,cudaMemcpyDeviceToHost);
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);
    exit (0);
}

__global__ void vecAdd (int *a, int *b, int *c) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) {
        c[i] = a[i] + b[i];
    }
}

• Since the maximum number of threads per dimension in a block is 1024, if you must use more than one block to access more threads.

• Divide the work between different blocks.

• Notice that each block is reserved completely; in this example, two blocks are reserved even though most of the second block is not utilized.

• WARNING: CUDA does not issue warnings or errors if your thread bookkeeping is incorrect -- Use small test cases to verify that everything is okay.
```
Higher Dimensional Grids/Blocks

- 1D grids/blocks are suitable for 1D data, but higher dimensional grids/blocks are necessary for:
  - higher dimensional data.
  - data set larger than the hardware dimensional limitations of blocks.

- CUDA has built-in variables and structures to define the number of blocks in a grid in each dimension and the number of threads in a block in each dimension.
CUDA Built-In Vector Types and Structures

- `uint3` and `dim3` are CUDA-defined structures of unsigned integers: `x`, `y`, and `z`.
  - `struct uint3 {x; y; z;};`
  - `struct dim3 {x; y; z;};`

- The unsigned structure components are automatically initialized to 1.

- These vector types are mostly used to define grid of blocks and threads.

- There are other CUDA vector types (discussed later).
CUDA Built-In Variables for Grid/Block Sizes

• **dim3 gridSize** -- Grid dimensions, x and y (z not used).

• Number of blocks in grid =
  gridSize.x * gridSize.y

• **dim3 blockDim** -- Size of block dimensions x, y, and z.

• Number of threads in a block =
  blockDim.x * blockDim.y * blockDim.z
Example Initializing Values

- To set dimensions:
  ```
  dim3 grid(16,16);  // grid = 16 x 16 blocks
  dim3 block(32,32);  // block = 32 x 32 threads
  myKernel<<<<<grid, block>>>(...);
  ```

- which sets:
  ```
  grid.x = 16;
  grid.y = 16;
  block.x = 32;
  block.y = 32;
  block.z = 1;
  ```
CUDA Built-In Variables for Grid/Block Indices

- `uint3 blockIdx` -- block index within grid:
  - `blockIdx.x, blockIdx.y` (z not used)
- `uint3 threadIdx` -- thread index within block:
  - `threadIdx.x, threadIdx.y, threadIdx.z`

- Full global thread ID in x and y dimensions can be computed by:
  - `x = blockIdx.x * blockDim.x + threadIdx.x;`
  - `y = blockIdx.y * blockDim.y + threadIdx.y;`
2D Grids and 2D Blocks

\[ \text{threadIdx.x} = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x} \]

\[ \text{threadIdx.y} = \text{blockIdx.y} \times \text{blockDim.y} + \text{threadIdx.y} \]

\[ \text{blockIdx.x} \]

\[ \text{blockIdx.y} \]

\[ \text{blockIdx.y} \times \text{blockDim.y} + \text{threadIdx.y} \]

\[ \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x} \]
Flatten Matrices into Linear Memory

• Generally memory allocated dynamically on device (GPU) and we cannot use two-dimensional indices (e.g. A[row][column]) to access matrices.

• We will need to know how the matrix is laid out in memory and then compute the distance from the beginning of the matrix.

• C uses **row-major** order --- rows are stored one after the other in memory, i.e. row 0 then row 1 etc.
Accessing Matrices in Linear Memory

In CUDA:
```
int col = blockIdx.x * blockDim.x + threadIdx.x;
int row = blockIdx.y * blockDim.y + threadIdx.y;
int index = col + row * N;
A[index] = ...
```
Matrix Addition: Add two 2D matrices

• Corresponding elements of each array \((a,b)\) added together to form element of third array \((c)\):

\[
c_{i,j} = a_{i,j} + b_{i,j}
\]

\((0 \leq i < n, \ 0 \leq j < m)\)
#define N 512
#define BLOCK_DIM 512
__global__ void matrixAdd (int *a, int *b, int *c);

int main() {
    int a[N][N], b[N][N], c[N][N];
    int *dev_a, *dev_b, *dev_c;

    int size = N * N * sizeof(int);
    // initialize a and b with real values (NOT SHOWN)
    cudaMalloc((void**)&dev_a, size);
    cudaMalloc((void**)&dev_b, size);
    cudaMalloc((void**)&dev_c, size);

    cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);

    dim3 dimBlock(BLOCK_DIM, BLOCK_DIM);
    dim3 dimGrid((int)ceil(N/dimBlock.x),(int)ceil(N/dimBlock.y));
    matrixAdd<<<dimGrid,dimBlock>>>(dev_a,dev_b,dev_c);
    cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);
    cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_c);
}

__global__ void matrixAdd (int *a, int *b, int *c) {
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int index = col + row * N;

    if (col < N && row < N) {
        c[index] = a[index] + b[index];
    }
}
Matrix Multiplication Review

\[ c_{i, j} = \sum_{k=0}^{l-1} a_{i,k}b_{k,j} \]

- To calculate the product of two matrices \( A \) and \( B \), we multiply the rows of \( A \) by the columns of \( B \) and add them up.

- Then place the sum in the appropriate position in the matrix \( C \).

\[
AB = \begin{bmatrix}
1 & 0 & -2 \\
0 & 3 & -1 \\
0 & 2 & 0 \\
\end{bmatrix}
\begin{bmatrix}
0 & 3 \\
-2 & -1 \\
0 & 4 \\
\end{bmatrix}
= \begin{bmatrix}
(1*0) + (0*-2) + (-2*0) \\
(0*0) + (3*-2) + (-1*0) \\
(0*0) + (2*3) + (0*4) \\
\end{bmatrix}
= \begin{bmatrix}
0 + 0 + 0 \\
0 + 0 + 0 \\
0 + 0 + 0 \\
\end{bmatrix}
= \begin{bmatrix}
0 & -5 \\
0 & -6 \\
\end{bmatrix}
= C
\]
Matrix Multiplication Review

\[ c_{i, j} = \sum_{k = 0}^{l - 1} a_{i,k} b_{k,j} \]

- To calculate the product of two matrices \( A \) and \( B \), we multiply the rows of \( A \) by the columns of \( B \) and add them up.

- Then place the sum in the appropriate position in the matrix \( C \).

\[
AB = \begin{bmatrix}
1 & 0 & -2 \\
0 & 3 & -1
\end{bmatrix}
\begin{bmatrix}
0 & 3 \\
-2 & -1 \\
0 & 4
\end{bmatrix}
\]

\[
= \begin{bmatrix}
(1*0) + (0*-2) + (-2*0) & (1*3) + (0*-1) + (-2*4) \\
(0*0) + (3*-2) + (-1*0) & (0*3) + (3*-1) + (-1*4)
\end{bmatrix}
\]

\[
= \begin{bmatrix}
0 + 0 + 0 & 3 + 0 + -8 \\
0 + -6 + 0 & 0 + -3 + -4
\end{bmatrix}
\]

\[
= \begin{bmatrix}
0 & -5 \\
-6 & -7
\end{bmatrix} = C
\]
Matrix Multiplication Review

To calculate the product of two matrices A and B, we multiply the rows of A by the columns of B and add them up.

Then place the sum in the appropriate position in the matrix C.

\[ c_{i, j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j} \]
Matrix Multiplication Review

- To calculate the product of two matrices $A$ and $B$, we multiply the rows of $A$ by the columns of $B$ and add them up.

- Then place the sum in the appropriate position in the matrix $C$.

\[
c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}
\]
Square Matrix Multiplication C Code

- Sequential algorithm consists of multiple nested for loops.
- Both multiplications and additions are in $O(N^3)$.
- Can it be parallelized?

```c
void matrixMult (int a[N][N], int b[N][N], int c[N][N], int width) {
    for (int i = 0; i < width; i++) {
        for (int j = 0; j < width; j++) {
            int sum = 0;
            for (int k = 0; k < width; k++) {
                int m = a[i][k];
                int n = b[k][j];
                sum += m * n;
            }
            c[i][j] = sum;
        }
    }
}
```
Motivation for Parallel Matrix Multiplication Algorithm

- To compute a single value of $C(i,j)$, only a single thread be necessary to traverse the $i$th row of $A$ and the $j$th column of $B$.

- Therefore, the number of threads needed to compute a square matrix multiply is $O(N^2)$. 
C to CUDA for Dummies
Step 1: Write the Kernel

C Function

```c
void matrixMult (int a[N][N], int b[N][N], int c[N][N], int width) {
    for (int i = 0; i < width; i++) {
        for (int j = 0; j < width; j++) {
            int sum = 0;
            for (int k = 0; k < width; k++) {
                int m = a[i][k];
                int n = b[k][j];
                sum += m * n;
            }
            c[i][j] = sum;
        }
    }
}
```

CUDA Kernel

```c
__global__ void matrixMult (int *a, int *b, int *c, int width) {
    int k, sum = 0;
    int col = threadIdx.x + blockDim.x * blockIdx.x;
    int row = threadIdx.y + blockDim.y * blockIdx.y;
    if(col < width && row < width) {
        for (k = 0; k < width; k++)
            sum += a[row * width + k] * b[k * width + col];
        c[row * width + col] = sum;
    }
}
```
```c
#define N 16
#include <stdio.h>
__global__ void matrixMult (int *a, int *b, int *c, int width);

int main() {
    int a[N][N], b[N][N], c[N][N];
    int *dev_a, *dev_b, *dev_c;

    // initialize matrices a and b with appropriate values
    int size = N * N * sizeof(int);
    cudaMalloc((void **) &dev_a, size);
    cudaMalloc((void **) &dev_b, size);
    cudaMalloc((void **) &dev_c, size);

    cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);

    dim3 dimGrid(1, 1);
    dim3 dimBlock(N, N);
    matrixMult<<<dimGrid, dimBlock>>>(dev_a, dev_b, dev_c, N);

    cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);
    cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_c);

    __global__ void matrixMult (int *a, int *b, int *c, int width) {
        int k, sum = 0;

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

        if(col < width && row < width) {
            for (k = 0; k < width; k++)
                sum += a[row * width + k] * b[k * width + col];
            c[row * width + col] = sum;
        }
    }
```