BLOCKS

BLOCKS

- **Blocks** are groups of threads that are executed together on a single Streaming Multiprocessor (SM).
- They are organized to efficiently utilize the GPU's resources and facilitate parallel processing.
- When you launch a kernel (a function that runs on the GPU) in CUDA, you specify the number of blocks and threads per block.
- The threads within a block can communicate and synchronize with each other through shared memory, making it possible to divide a complex task into smaller units and process them in parallel

BLOCKS

kernel_function<<<num_blocks, num_threads>>>(param1, param2,...)

- If you change this from one to two, you double the number of threads you are asking the GPU to invoke on the hardware. Thus, the same call,

some_ kernel_ func<<< 2, 128 >>>(a, b, c);

- will call the GPU function named **some_kernel**_func 2 x 128 times, each with a different thread. This, however, complicates the calculation of the ***thread_id x parameter,*** effectively the array index position.

```
__global__ void some_kernel_func (int * const a, const int * const b, const int * const c)
{
const unsigned int thread_idx = (blockIdx.x * blockDim.x) + threadIdx.x;
a[thread_idx] = b[thread_idx] * c[thread_idx];
}
```

BLOCKS

- To calculate the ***thread_idx*** parameter, we have to consider number of blocks.
- 1024 threads per block on the Fermi hardware
- 65,536 blocks would translate into around 64 million threads.
- At 1024 threads, you only get one thread block per SM.
- you need some 65,536 SMs in a single GPU

BLOCKS

- With 64 million threads, assuming one thread per array element, you can process up to 64 million elements.
- Assuming each element is a **single-precision floating-point** number, requiring **4 bytes** of data, need around **256 million** bytes, or 256 MB, of data storage space.
- Almost all GPU cards support at least this amount of memory space.
- working with threads and blocks alone you can achieve quite a large amount of parallelism.

BLOCKS

Block 0 Warp 0 (Thread 0 to 31)	Block 0 Warp 1 (Thread 32 to 63)	Block 1 Warp 0 (Thread 64 to 95)	Block 1 Warp 1 (Thread 96 to 127)

Address 0 to 31	Address 32 to 63	Address 64 to 95	Address 96 to 127

Fig : Block mapping to address

Block arrangement

- A kernel program to print the block, thread, warp, and thread index to the screen.
- Unless you have at least version 3.2 of the SDK, the printf statement is not supported in kernels.
- So we'll ship the data back to the CPU and print it to the console window.
- The kernel program is thus as follows:

Block arrangement

```
__global__ void what_is_my_id(unsigned int * const block, unsigned int * const thread, unsigned int * const warp, unsigned int * const calc_thread)
		{
/* Thread id is block index * block size +  thread offset into the block */
		const unsigned int thread_idx =(blockIdx.x * blockDim.x) + threadIdx.x;
		block[thread_idx] =blockIdx.x;
		thread[thread_idx] = threadIdx.x;

 /* Calculate warp using built in variable warpSize */
		warp[thread_idx] = threadIdx.x / warpSize;
		calc_thread[thread_idx] = thread_idx;
		}
```

Block arrangement

- On the CPU you have to run a section of code, to allocate memory for the arrays on the GPU and then transfer the arrays back from the GPU and display them on the CPU.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
__global__ void what_is_my_id(unsigned int * const block,
unsigned int * const thread,
unsigned int * const warp,
unsigned int * const calc_thread)
{
```

Block arrangement

```
/* Thread id is block index * block size +thread offset into the block */
        const unsigned int thread_idx = (blockIdx.x * blockDim.x) + threadIdx.x;
        block[thread_idx] = blockIdx.x;
        thread[thread_idx] = threadIdx.x;

/* Calculate warp using built in variable warpSize */
        warp[thread_idx] = threadIdx.x / warpSize;
        calc_thread[thread_idx] = thread_idx;
        }
```

Block arrangement

```
#define ARRAY_SIZE 128
#define ARRAY_SIZE_IN_BYTES (sizeof(unsigned int) * (ARRAY_SIZE))
/* Declare statically four arrays of ARRAY_SIZE each */
unsigned int cpu_block[ARRAY_SIZE];
unsigned int cpu_thread[ARRAY_SIZE];
unsigned int cpu_warp[ARRAY_SIZE];
unsigned int cpu_calc_thread[ARRAY_SIZE];
int main(void)
{
/* Total thread count = 2 * 64 = 128 */
const unsigned int num_blocks = 2;
const unsigned int num_threads = 64;
char ch;
```

Block arrangement

```
/* Declare pointers for GPU based parameters */
        unsigned int * gpu_block;
        unsigned int * gpu_thread;
        unsigned int * gpu_warp;
        unsigned int * gpu_calc_thread;
/* Declare loop counter for use later */
        unsigned int i;
/* Allocate four arrays on the GPU */
        cudaMalloc((void **)&gpu_block, ARRAY_SIZE_IN_BYTES);
        cudaMalloc((void **)&gpu_thread, ARRAY_SIZE_IN_BYTES);
        cudaMalloc((void **)&gpu_warp, ARRAY_SIZE_IN_BYTES);
        cudaMalloc((void **)&gpu_calc_thread, ARRAY_SIZE_IN_BYTES);
```

Block arrangement

```
/* Execute our kernel */
	what_is_my_id<<<num_blocks, num_threads>>>(gpu_block, gpu_thread,
gpu_warp,gpu_calc_thread);

/* Copy back the gpu results to the CPU  /
	cudaMemcpy(cpu_block, gpu_block, ARRAY_SIZE_IN_BYTES,
	cudaMemcpyDeviceToHost);
	cudaMemcpy(cpu_thread, gpu_thread, ARRAY_SIZE_IN_BYTES,
	cudaMemcpyDeviceToHost);
	cudaMemcpy(cpu_warp, gpu_warp, ARRAY_SIZE_IN_BYTES,
	cudaMemcpyDeviceToHost);
	cudaMemcpy(cpu_calc_thread, gpu_calc_thread, ARRAY_SIZE_IN_BYTES,
	cudaMemcpyDeviceToHost);
```

Block arrangement

```
/* Free the arrays on the GPU as now we're done with them */
        cudaFree(gpu_block);
        cudaFree(gpu_thread);
        cudaFree(gpu_warp);
        cudaFree(gpu_calc_thread);

/* Iterate through the arrays and print */
        for (i=0; i < ARRAY_SIZE; i++)
        {
        printf("Calculated Thread: %3u - Block: %2u - Warp %2u - Thread %3u\n",
        cpu_calc_thread[i], cpu_block[i], cpu_warp[i], cpu_thread[i]);
        }
        ch =getch();
        }
```

Block arrangement

- The output of the previous program is as follows:

Calculated Thread: 0 - Block: 0 - Warp 0 - Thread 0
Calculated Thread: 1 - Block: 0 - Warp 0 - Thread 1
Calculated Thread: 2 - Block: 0 - Warp 0 - Thread 2
Calculated Thread: 3 - Block: 0 - Warp 0 - Thread 3
Calculated Thread: 4 - Block: 0 - Warp 0 - Thread 4

Calculated Thread: 30 - Block: 0 - Warp 0 - Thread 30
Calculated Thread: 31 - Block: 0 - Warp 0 - Thread 31
Calculated Thread: 32 - Block: 0 - Warp 1 - Thread 32
Calculated Thread: 33 - Block: 0 - Warp 1 - Thread 33
Calculated Thread: 34 - Block: 0 - Warp 1 - Thread 34

Calculated Thread: 62 - Block: 0 - Warp 1 - Thread 62
Calculated Thread: 63 - Block: 0 - Warp 1 - Thread 63
Calculated Thread: 64 - Block: 1 - Warp 0 - Thread 0
Calculated Thread: 65 - Block: 1 - Warp 0 - Thread 1
Calculated Thread: 66 - Block: 1 - Warp 0 - Thread 2
Calculated Thread: 67 - Block: 1 - Warp 0 - Thread 3