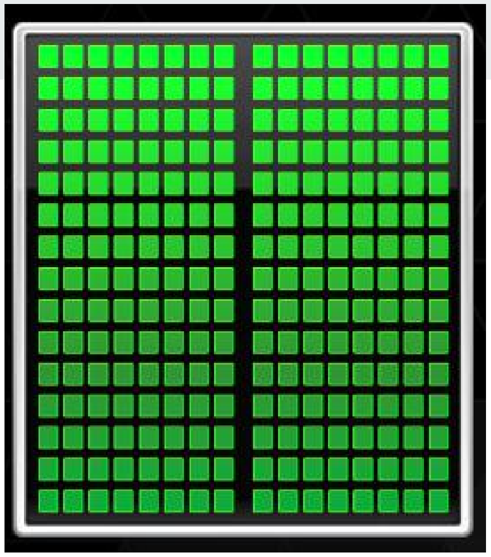
Introduction to CUDA Programming

Dept. of MACS NITK

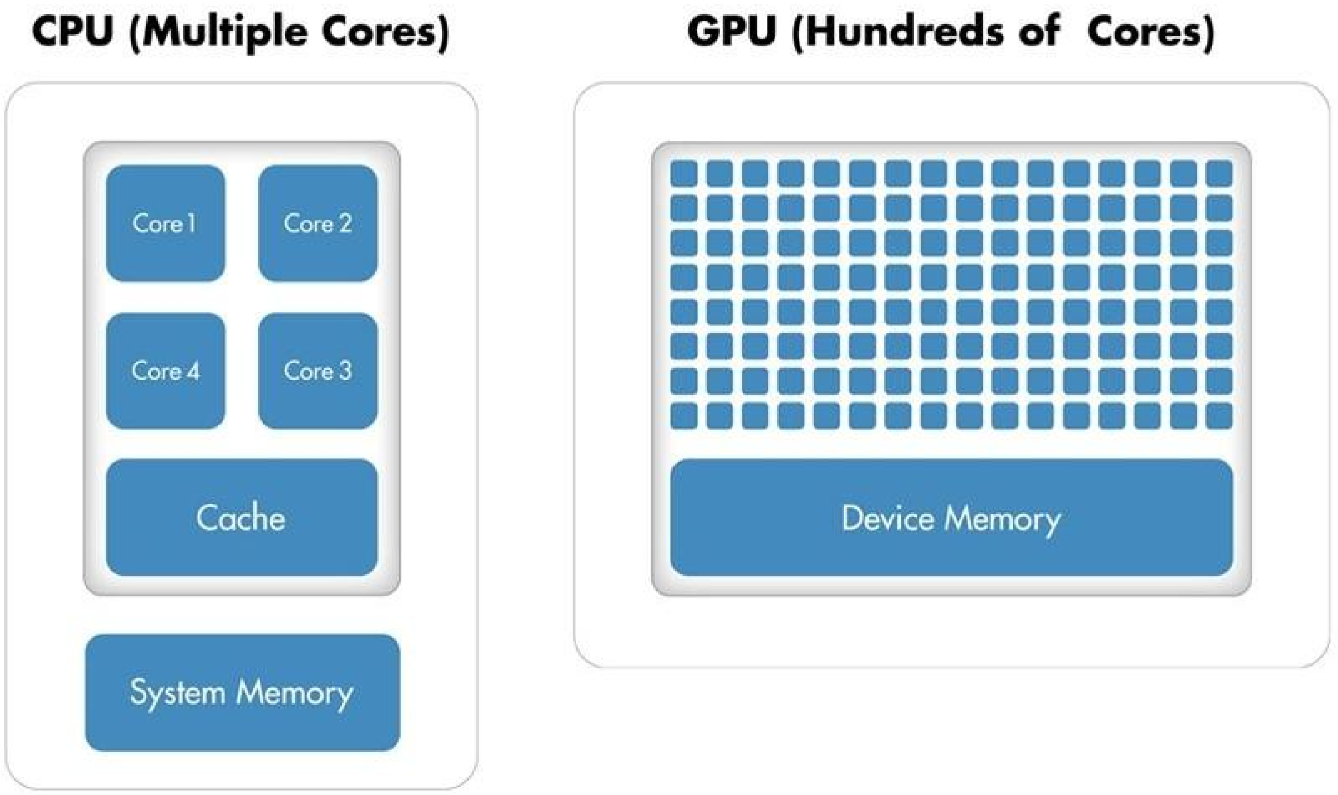
Let's learn about GPU..

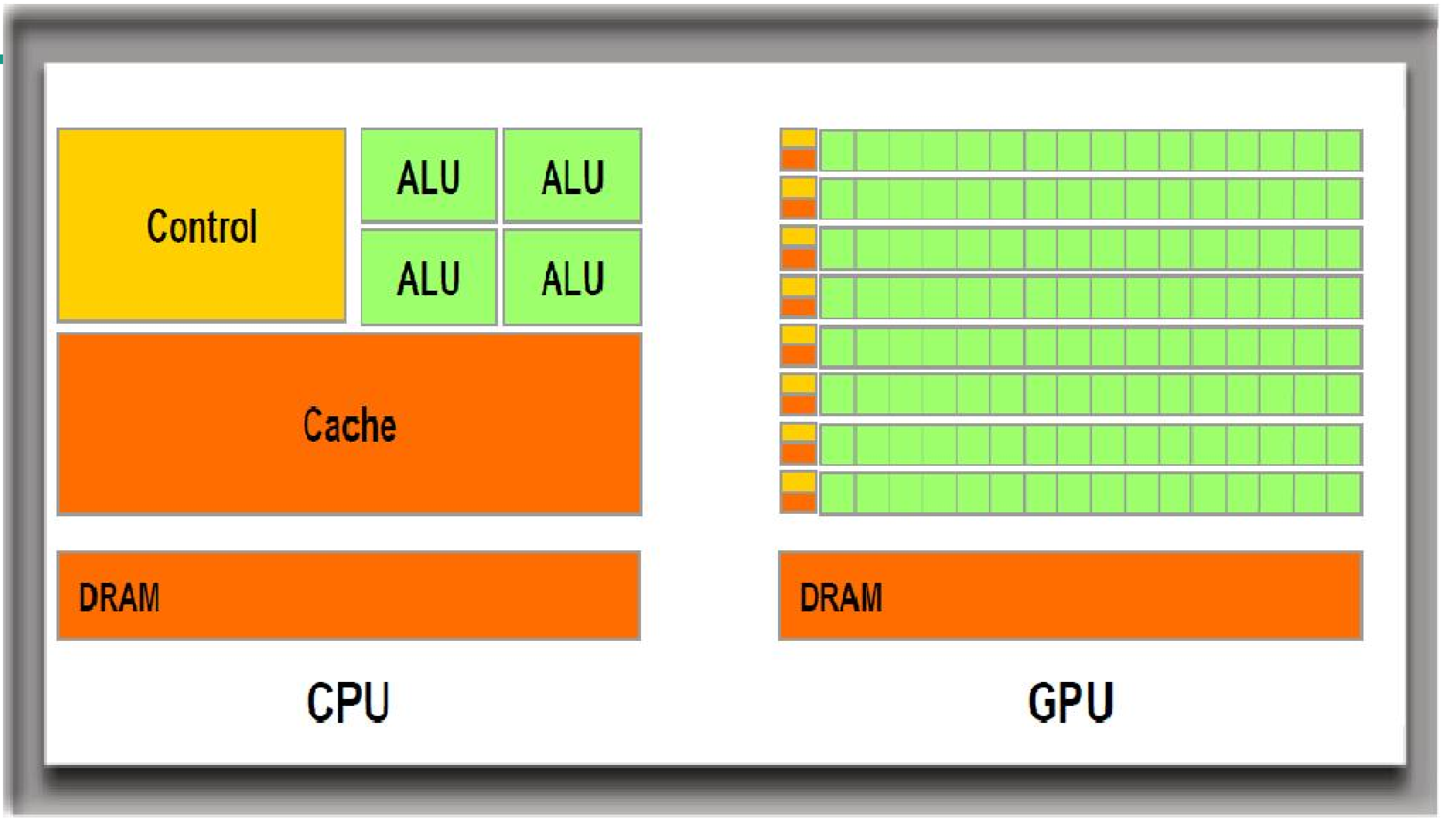
- A little history:
  - The first GPUs were designed as graphics accelerators
    - supported only specific fixed-function pipelines.
  - In the late 1990s, the hardware became increasingly programmable
    - Culminating in NVIDIA's first GPU in 1999.

Graphics Processing Unit



- Has thousands of cores and ALUs.
- They can handle billions of repetitive low level tasks.
- GPU is specialized for compute-intensive, highly parallel computation.
- They are devoted to data processing rather than data caching and flow control.
- Follows SPMD processing model.


CPU (Multiple Cores)
Core 1
Core 2
Core 4
Core 3
Cache
System Memory
GPU (Hundreds of Cores)
Device Memory


Control
ALU
ALU
ALU
ALU
Cache
DRAM
CPU
DRAM
GPU

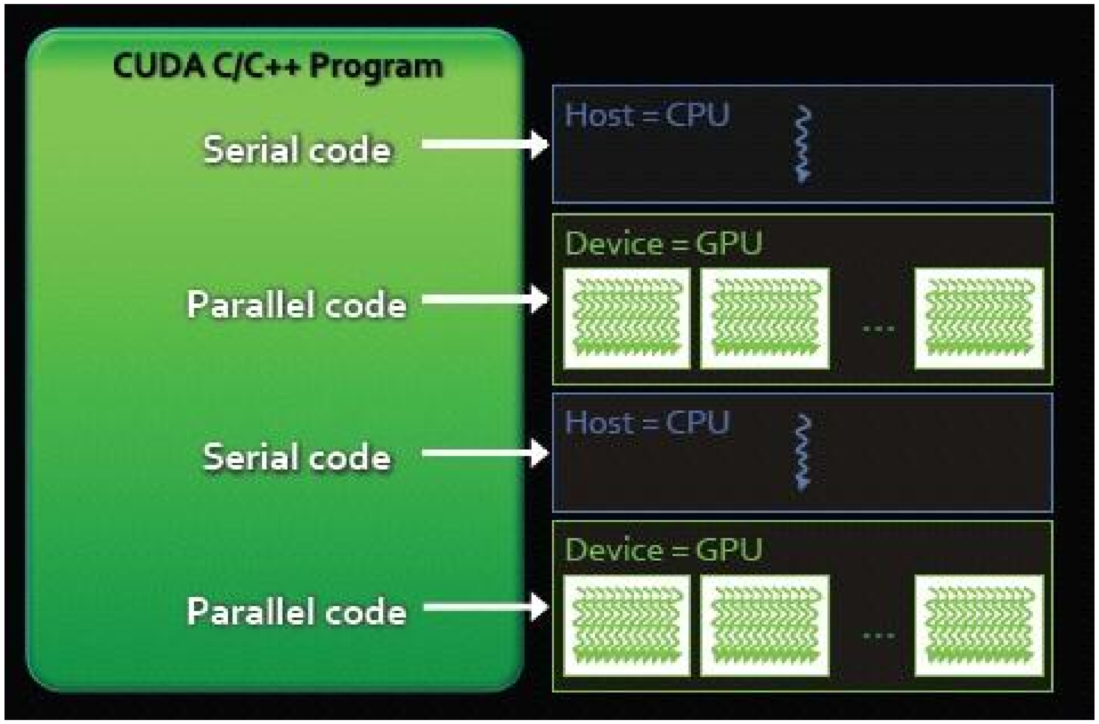
CUDA – Compute Unified Device Architecture

- NVIDIA invited Ian Buck to join the company.
  - Started evolving a solution to seamlessly run C on the GPU.
  - Putting the software and hardware together, NVIDIA unveiled CUDA in 2006
  - CUDA was launched in 2007.
  - The world's first solution for general-computing on GPUs

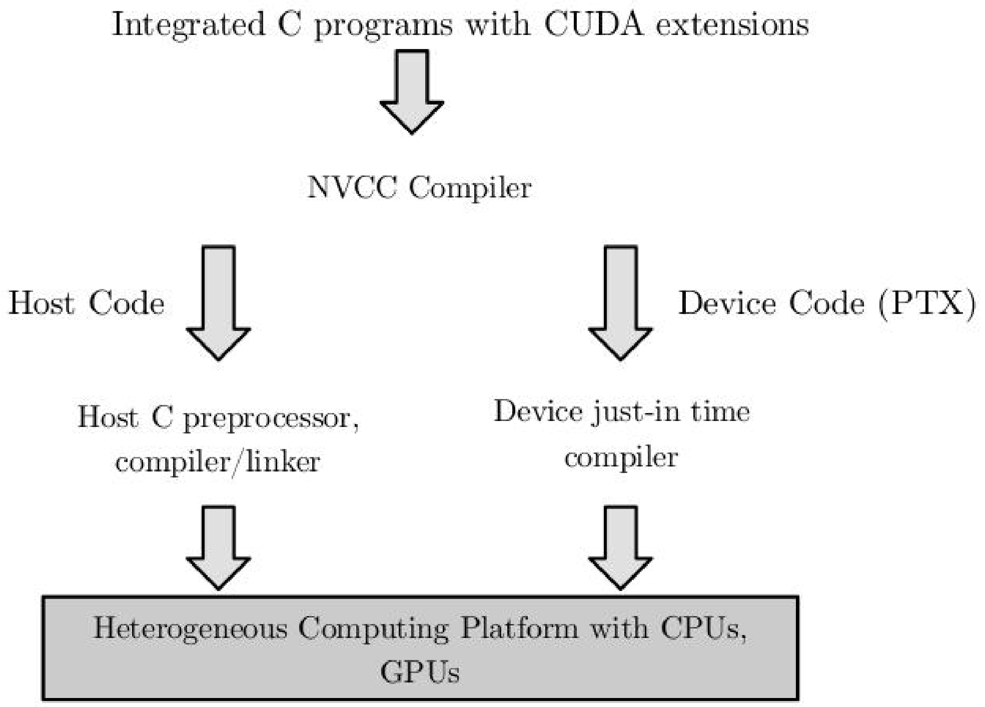
General Structure of the GPU Program in CUDA

- Host Program – Executed by the CPU.
  - This is a serial code.
  - Sets up the parameters for GPU (kernel) execution.
- Kernel Program – Executed in Parallel by the SIMD cores (Streaming Processors) in the GPU.

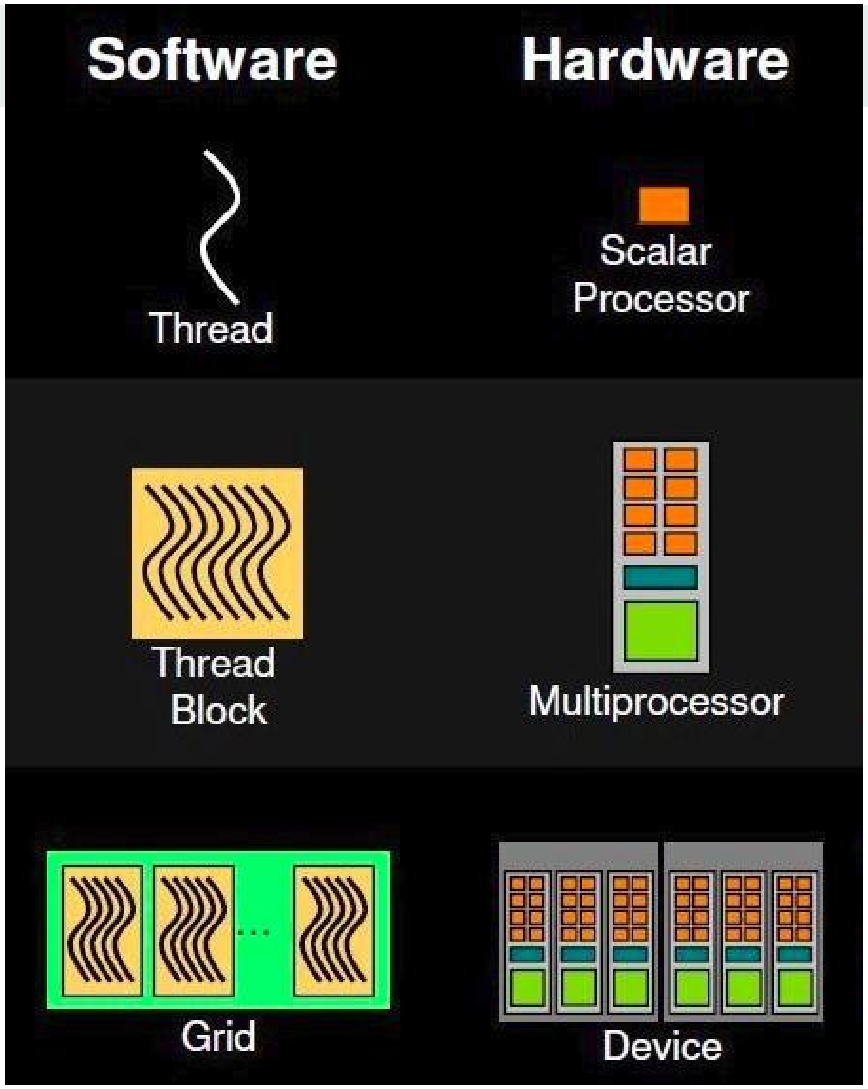
CUDA Program Execution

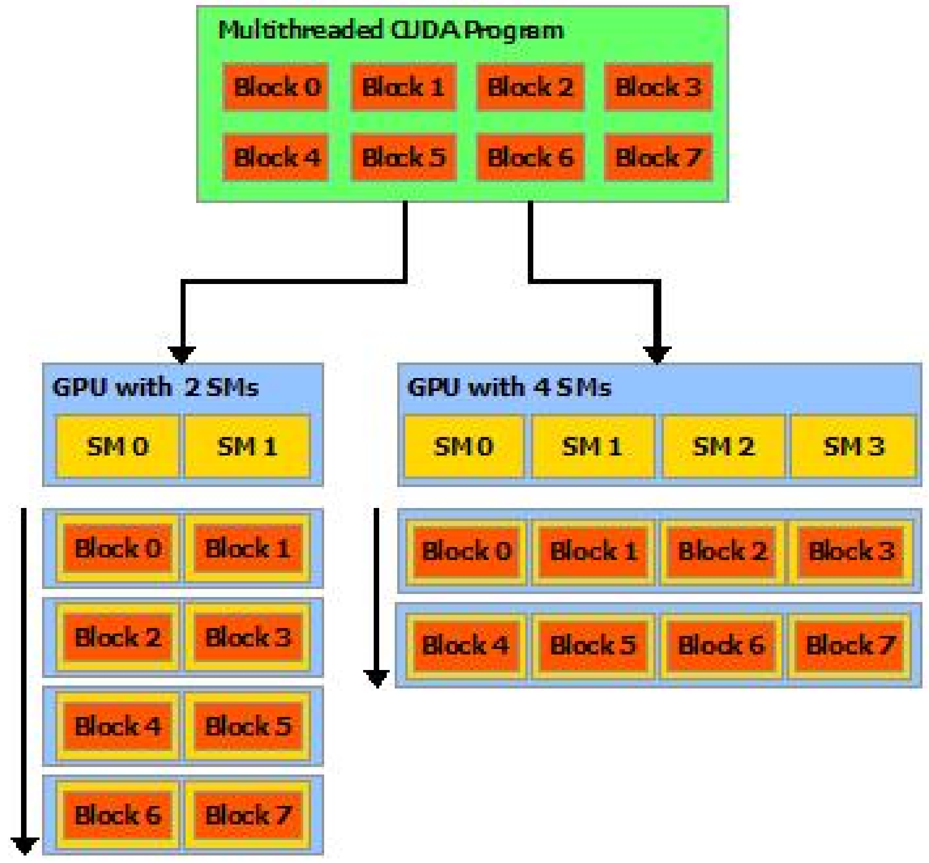


CUDA Program Execution

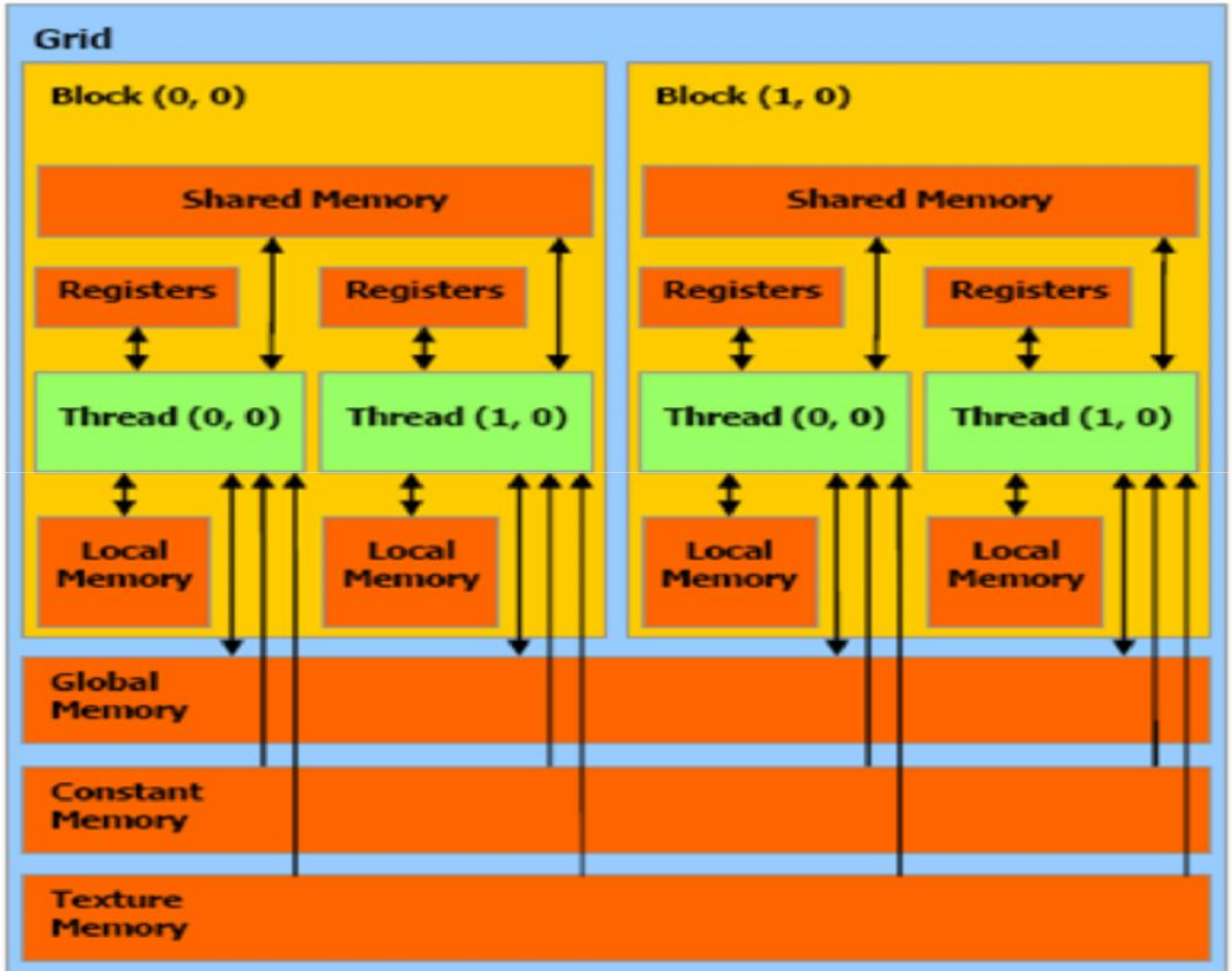

Integrated C programs with CUDA extensions
NVCC Compiler
Host Code
Device Code (PTX)
Host C preprocessor, compiler/linker
Device just-in time compiler
Heterogeneous Computing Platform with CPUs, GPUs

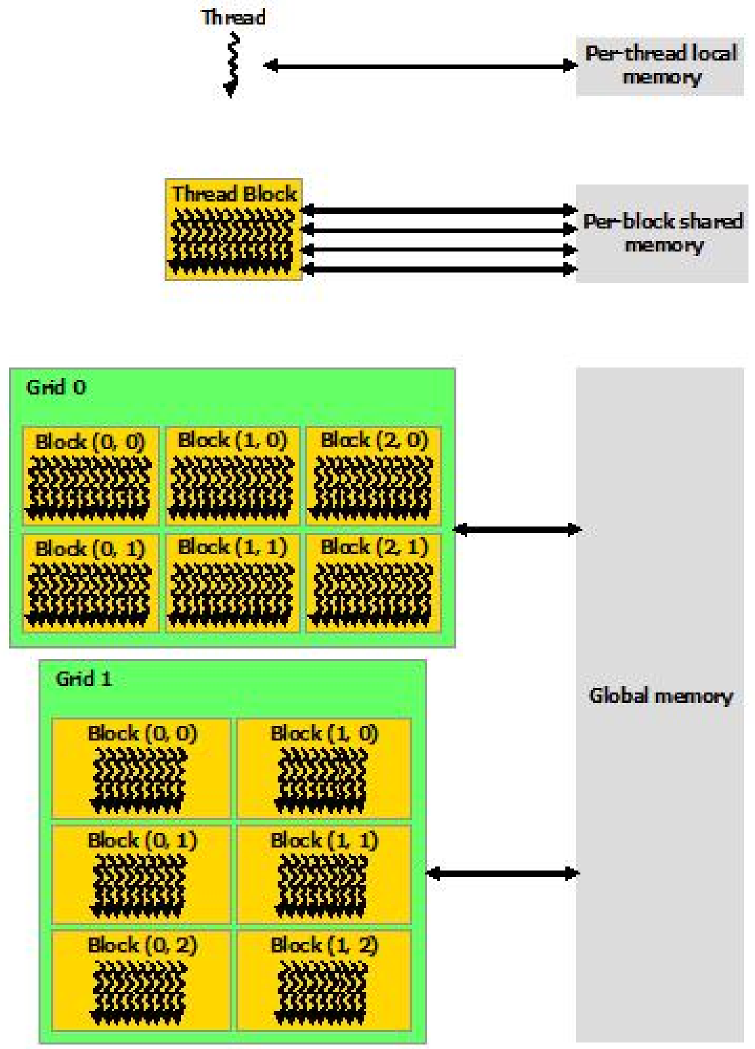
CUDA Execution Model




Multithreaded CUDA Program
Block 0
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7
GPU with 2 SMs
SM 0
SM 1
Block 0
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7
GPU with 4 SMs
SM 0
SM 1
SM 2
SM 3
Block 0
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7

CUDA Memory Model:




Thread
Per-thread local memory
Thread Block
Per-block shared memory
Grid 0
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Grid 1
Block (0, 0)
Block (1, 0)
Block (0, 1)
Block (1, 1)
Block (0, 2)
Block (1, 2)
Global memory

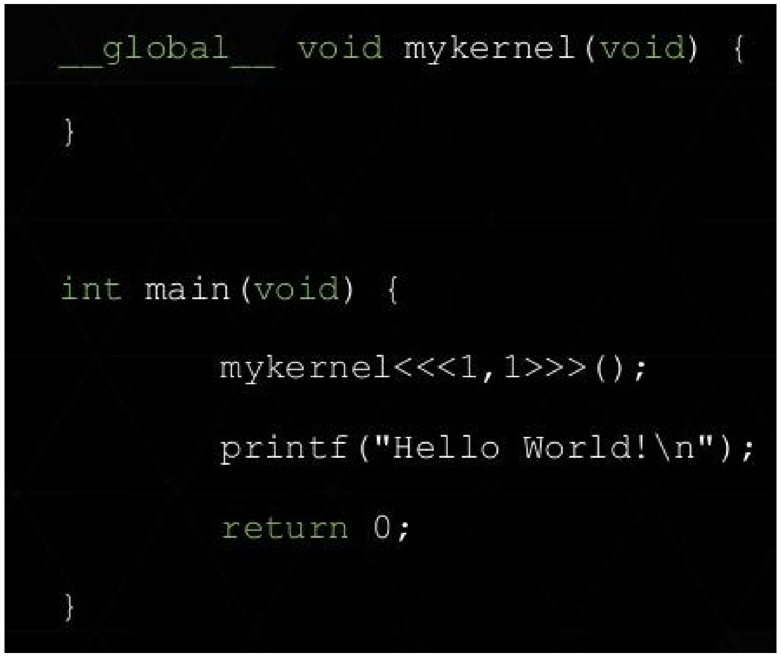
- Local Memory – Accessible only by a single thread in a thread block.
- Shared Memory – All threads in a single thread block have access to the shared memory.
- Global Memory – All thread blocks have access to global memory. Can be **read or write**.
- Constant Memory – All thread blocks have access to Constant memory. It is **read only memory**.
- Texture Memory – It is a type of Global memory which is **read only cache memory**. Accessible by all the thread blocks. High speed memory.

Steps in GPU execution

- Initialization of host data on CPU
- Allocate memory on GPU
- The data has to be transferred from CPU to GPU : Copy the data from host memory to the allocated GPU memory.
- Set the kernel parameters.
- Perform the computation in parallel.
- Transfer back the computed result from GPU to CPU.
- Free the allocated memory on GPU.

Every programming language introduction starts with Hello World!

```
int main(void)
{
printf("Hello World!\n");
return  0;
}
```



```
__global__ void mykernel(void) {
}

int main(void) {
        mykernel<<<1,1>>>();
        printf("Hello World!\n");
        return 0;
}
```

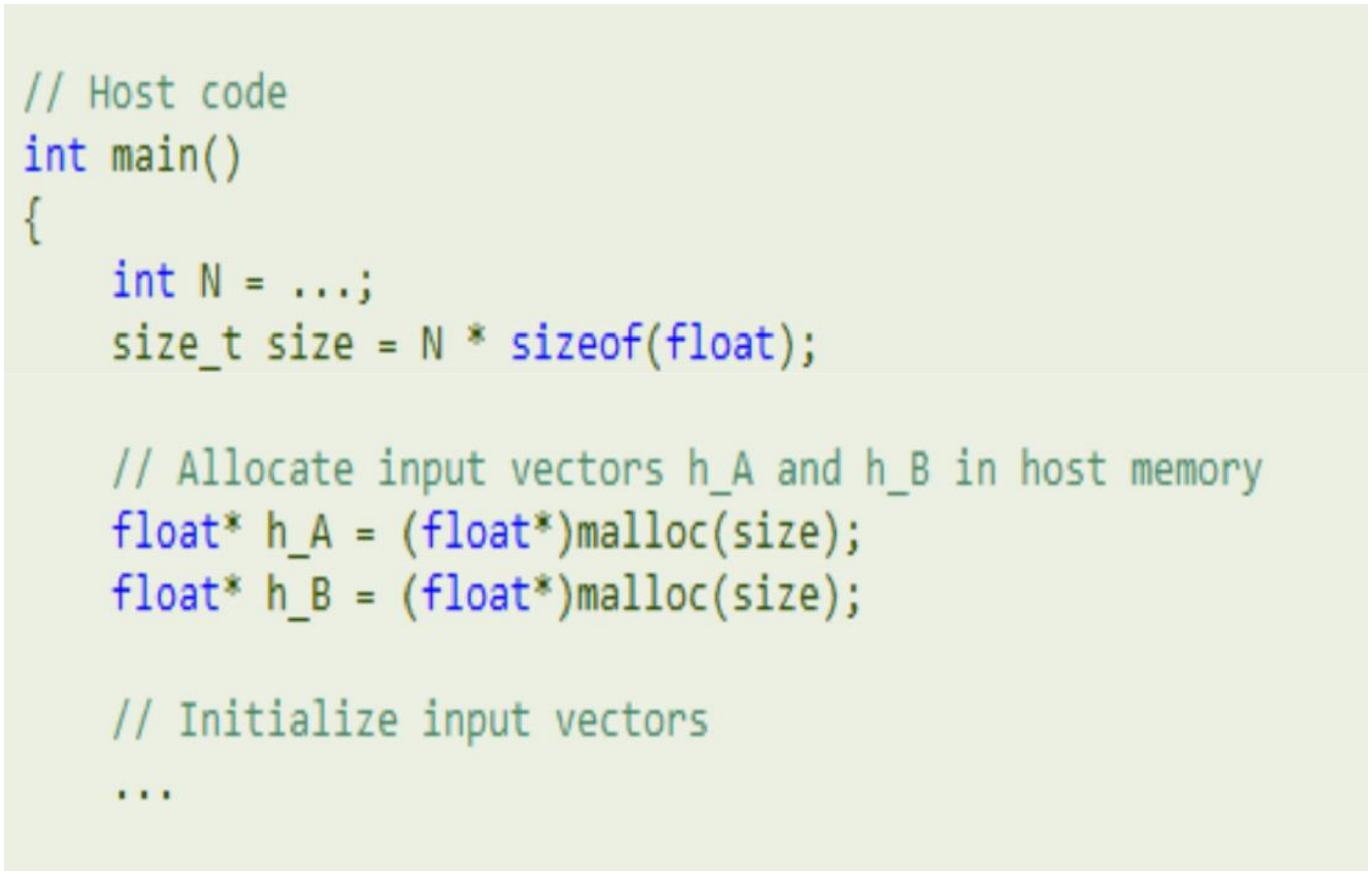
Learn about:

- Keyword __global__ indicates a function that runs on device.
- It is called from host.
- mykernel() is a device function processed by NVIDIA compiler.
- Host functions are processed by host compiler.

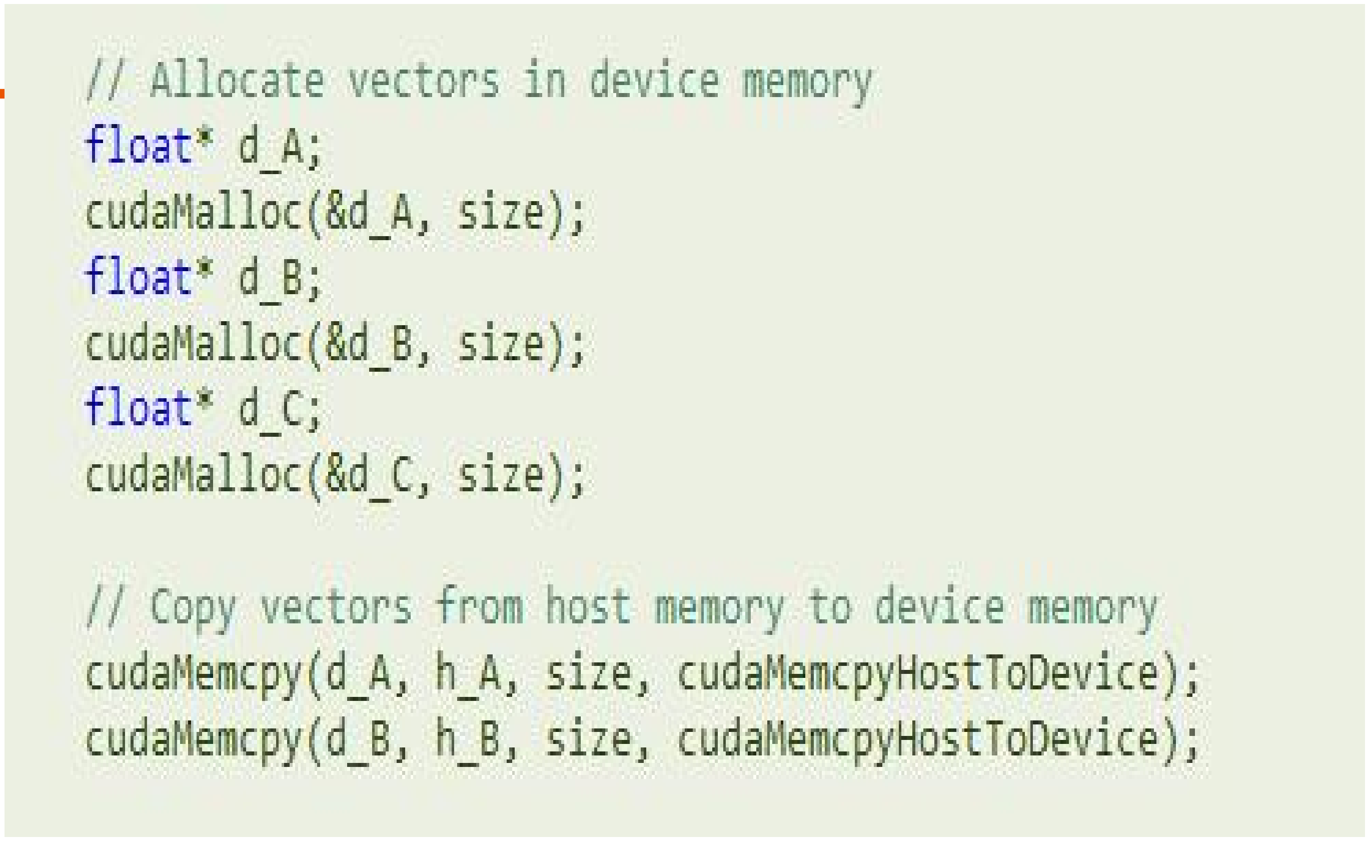
- mykernel<<<1,1>>>() launches the kernel
- Two parameters indicate the number of blocks and number of threads per block.

Vector Addition

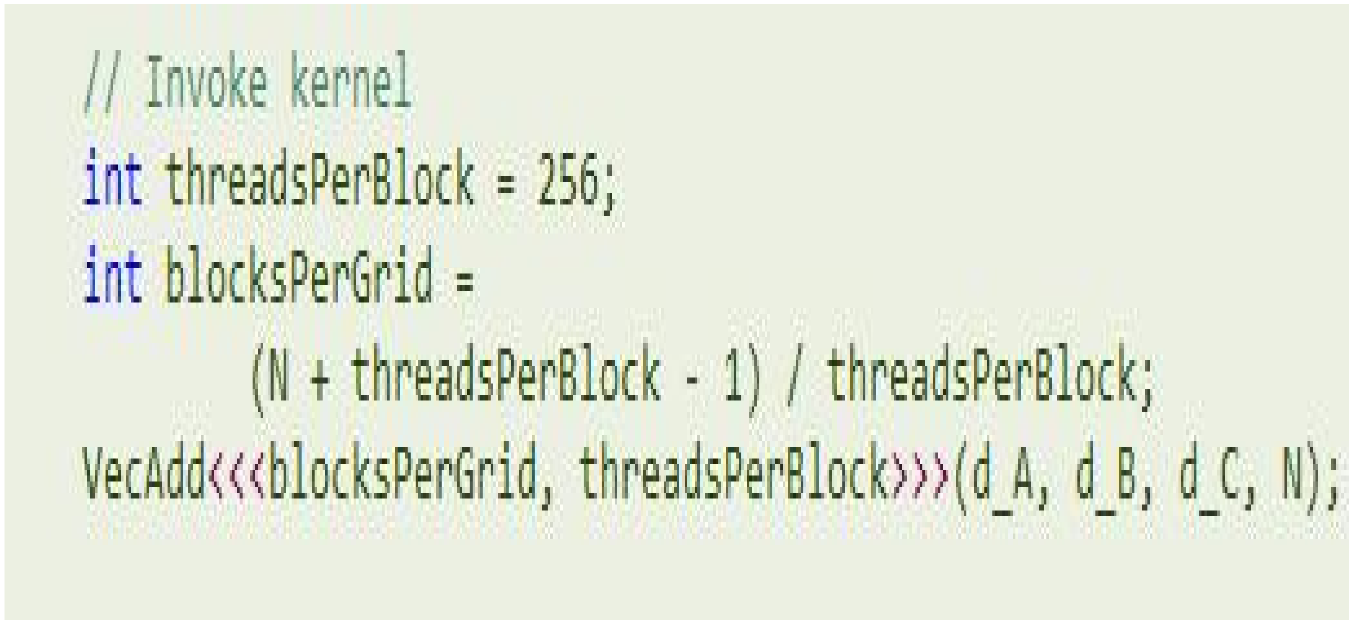
Vector Addition



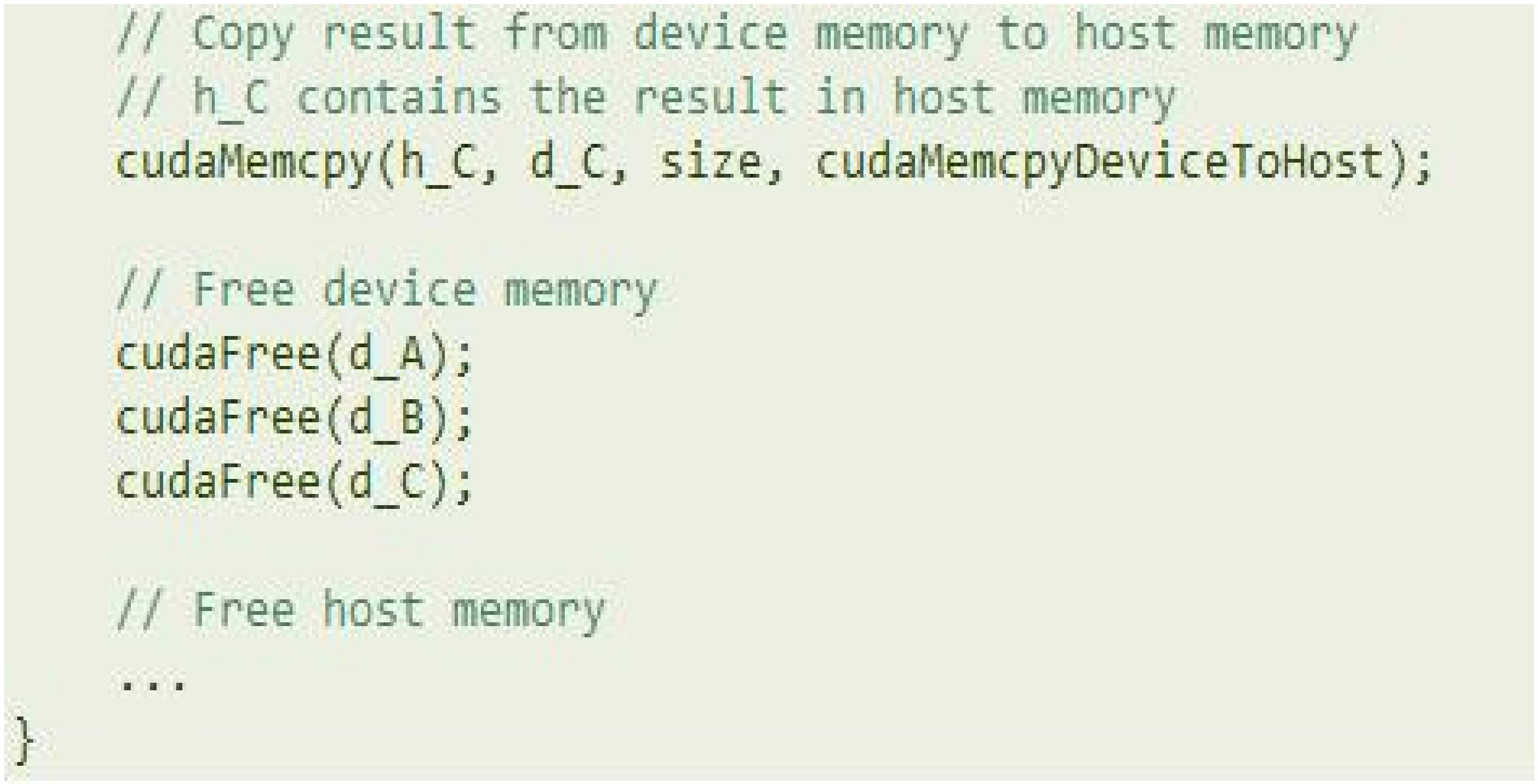
```
// Host code
int main()
{
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);

    // Initialize input vectors
    ...
```



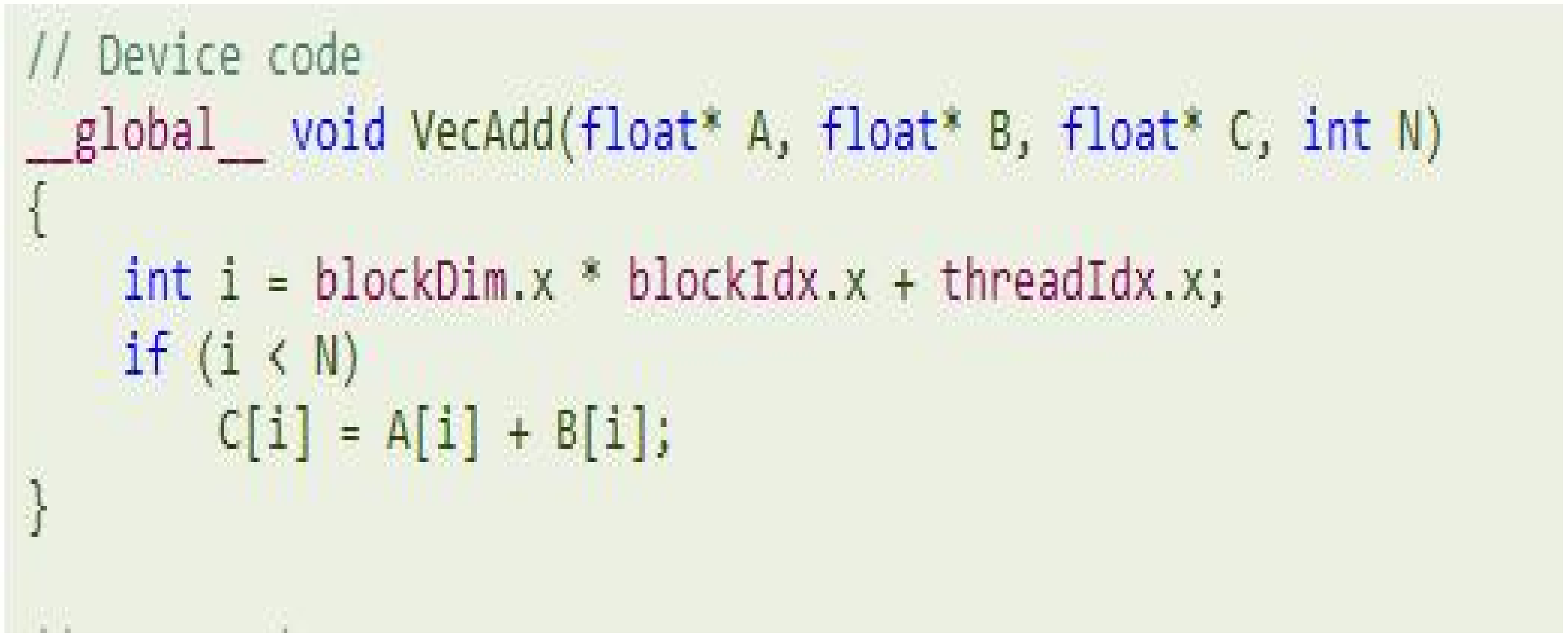
```c
// Allocate vectors in device memory
float* d_A;
cudaMalloc(&d_A, size);
float* d_B;
cudaMalloc(&d_B, size);
float* d_C;
cudaMalloc(&d_C, size);

// Copy vectors from host memory to device memory
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```



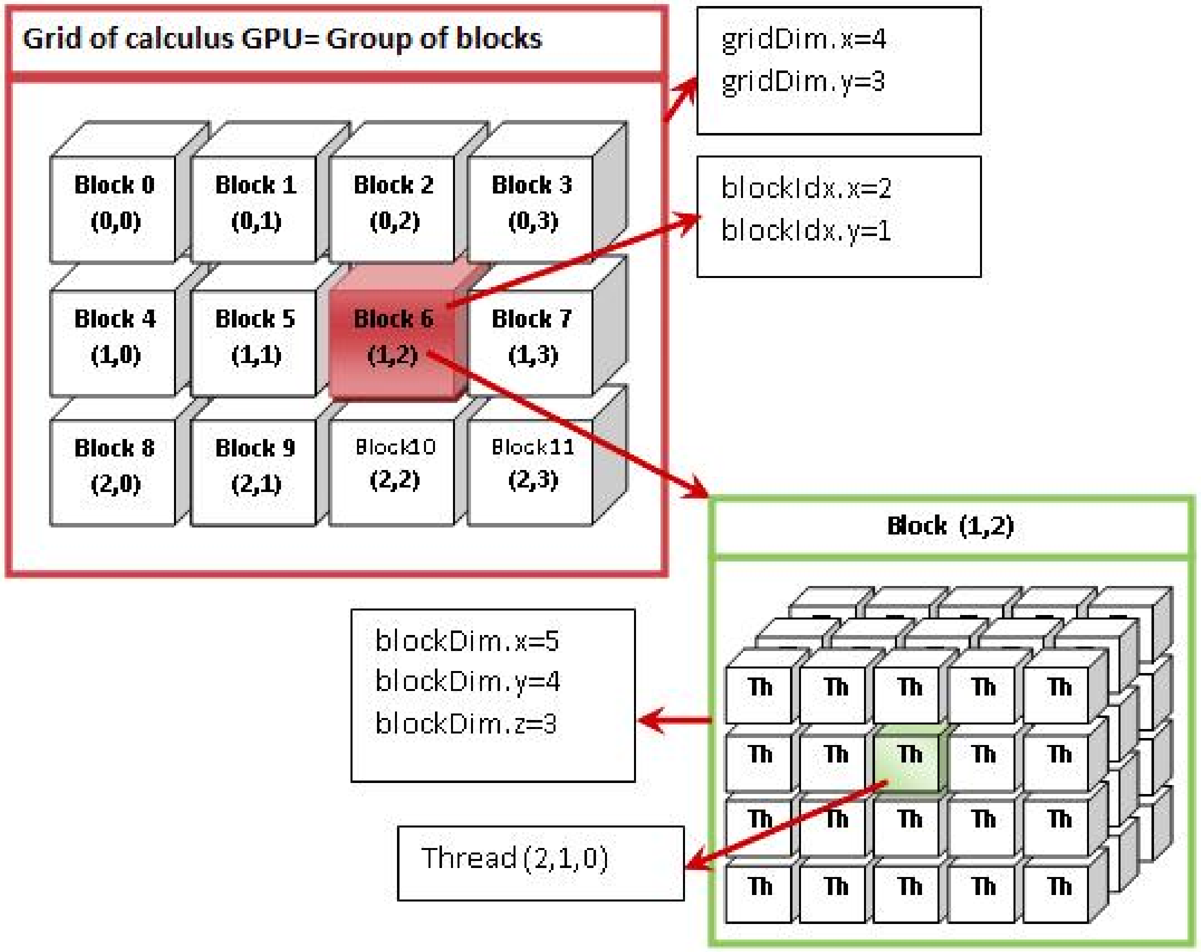
```
// Invoke kernel
int threadsPerBlock = 256;
int blocksPerGrid =
        (N + threadsPerBlock - 1) / threadsPerBlock;
VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
```



```
    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    ...
}
```



```
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```


Grid of calculus GPU= Group of blocks
gridDim.x=4
gridDim.y=3
Block 0 (0,0)
Block 1 (0,1)
Block 2 (0,2)
Block 3 (0,3)
blockIdx.x=2
blockIdx.y=1
Block 4 (1,0)
Block 5 (1,1)
Block 6 (1,2)
Block 7 (1,3)
Block 8 (2,0)
Block 9 (2,1)
Block10 (2,2)
Block11 (2,3)
Block (1,2)
blockDim.x=5
blockDim.y=4
blockDim.z=3
Thread (2,1,0)
Th