GPU Programming with CUDA

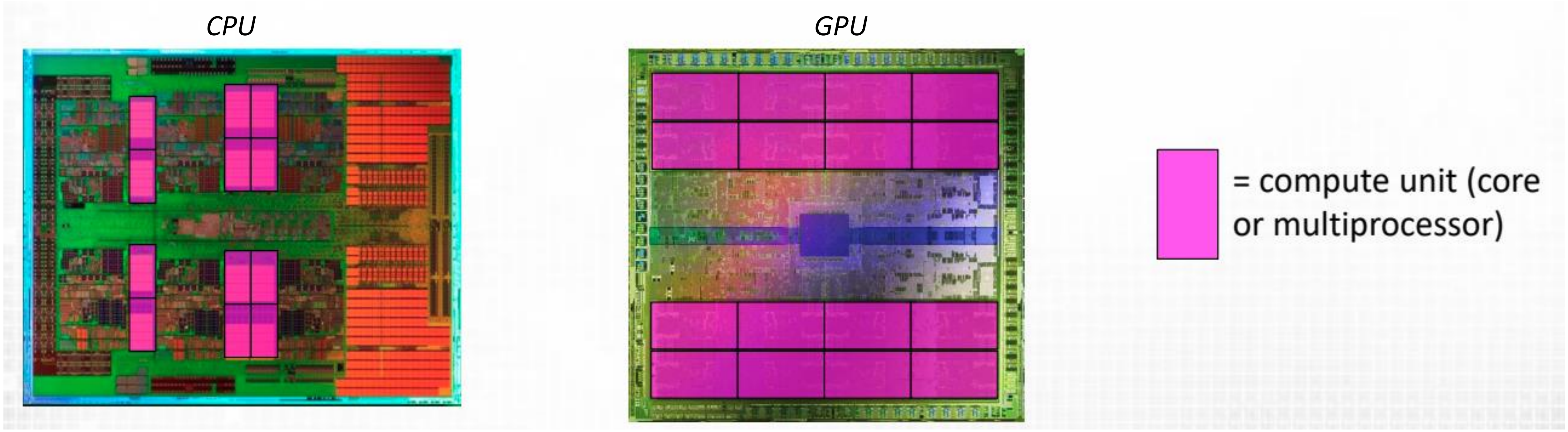
What is CUDA?

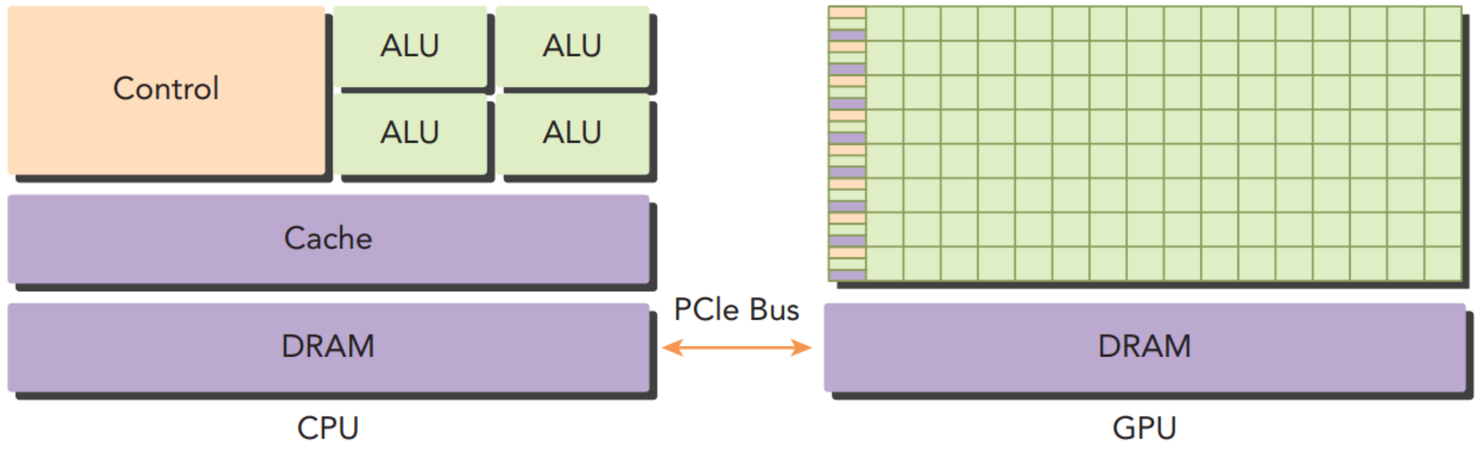
In November 2006, NVIDIA introduced CUDA, a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs.

Environment available in:

- C
- C++
- Fortran
- Python, Java wrappers

A closer look at CPU & GPU

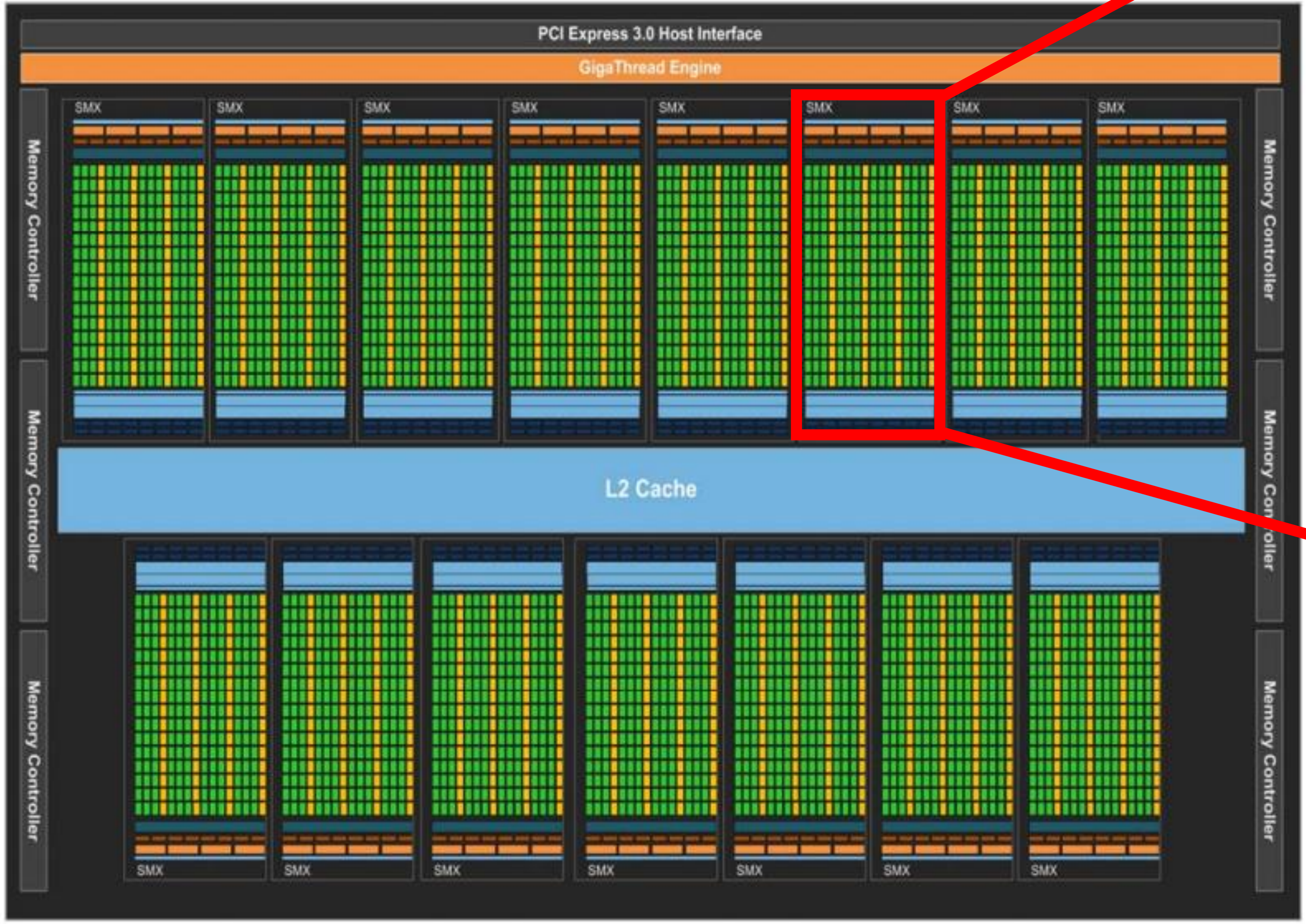




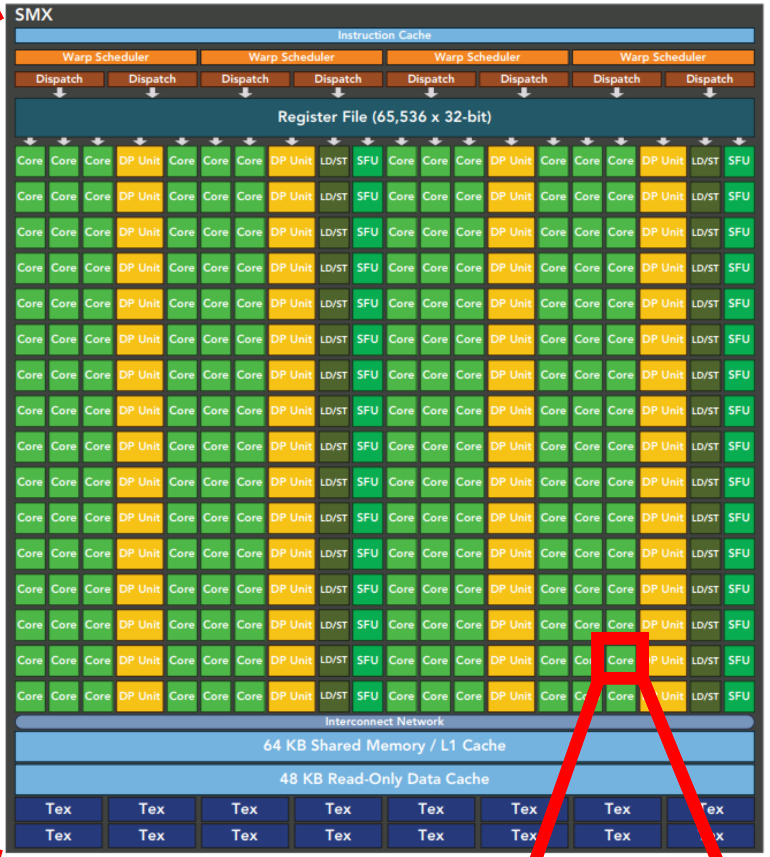
- **A CPU core**, relatively heavy-weight, is designed for very complex control logic, seeking to optimize the execution of sequential programs
- **A GPU core**, relatively light-weight, is optimized for data-parallel tasks with simpler control logic, focusing on the throughput of parallel programs

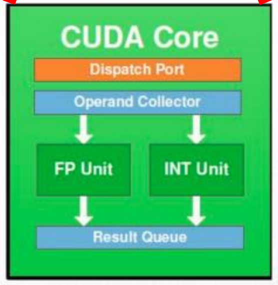
A closer look at GPU: a multi-multi-core machine

GPU: An array of *Streaming Multiprocessors* (SM)



Streaming Multiprocessor (SM)


SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit)
Core
DP Unit
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex

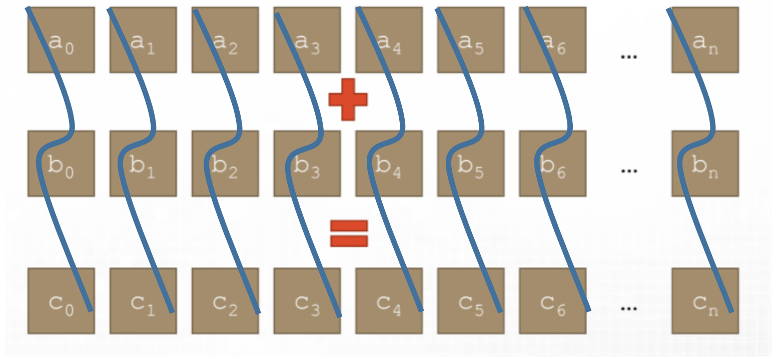


- *Less Scheduling units than cores*
- *Cores multiples of 32*
- *Scheduling of cores/ threads in groups of 32 (warps)*

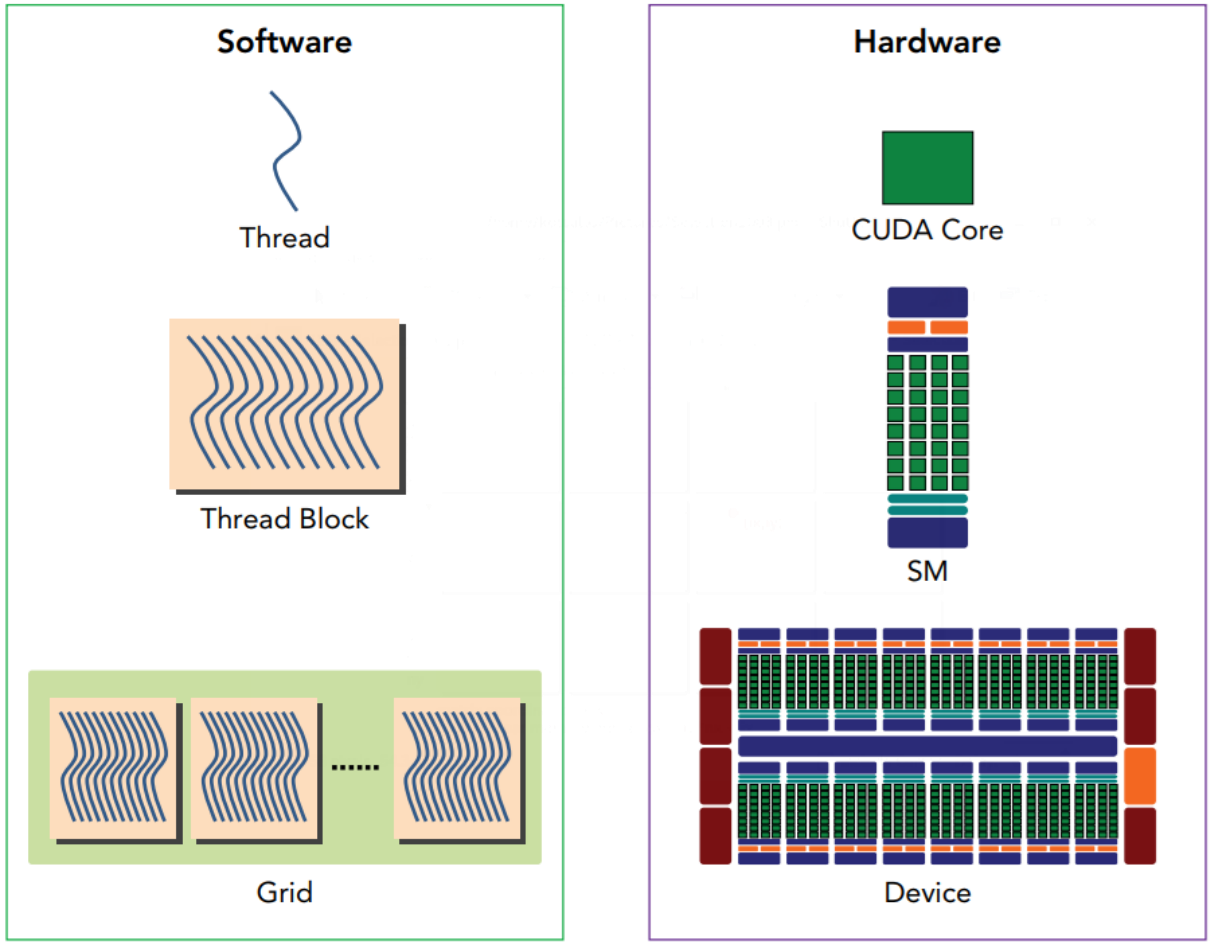
First GPU program

Kernel is the code that is called by the **host** (CPU) and executed by the **device** (GPU)

```
int main()
{
   ...

   //Kernel invocation (grid of blocks of threads)
   VecAdd<<<Blocks,ThreadsPerBlock>>>(A,B,C);

   ...
}
```



a_0 a_1 a_2 a_3 a_4 a_5 a_6 ... a_n
+
b_0 b_1 b_2 b_3 b_4 b_5 b_6 ... b_n
=
c_0 c_1 c_2 c_3 c_4 c_5 c_6 ... c_n

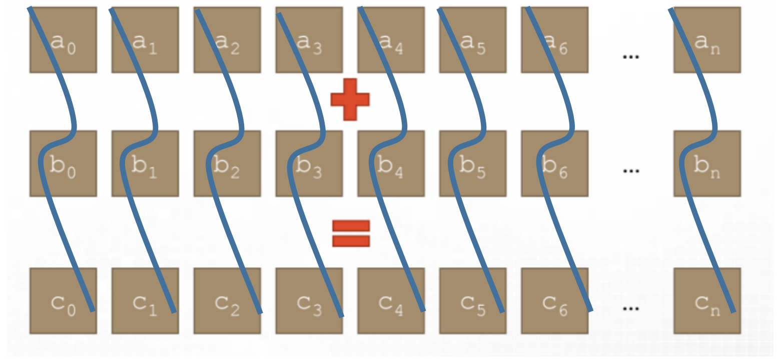


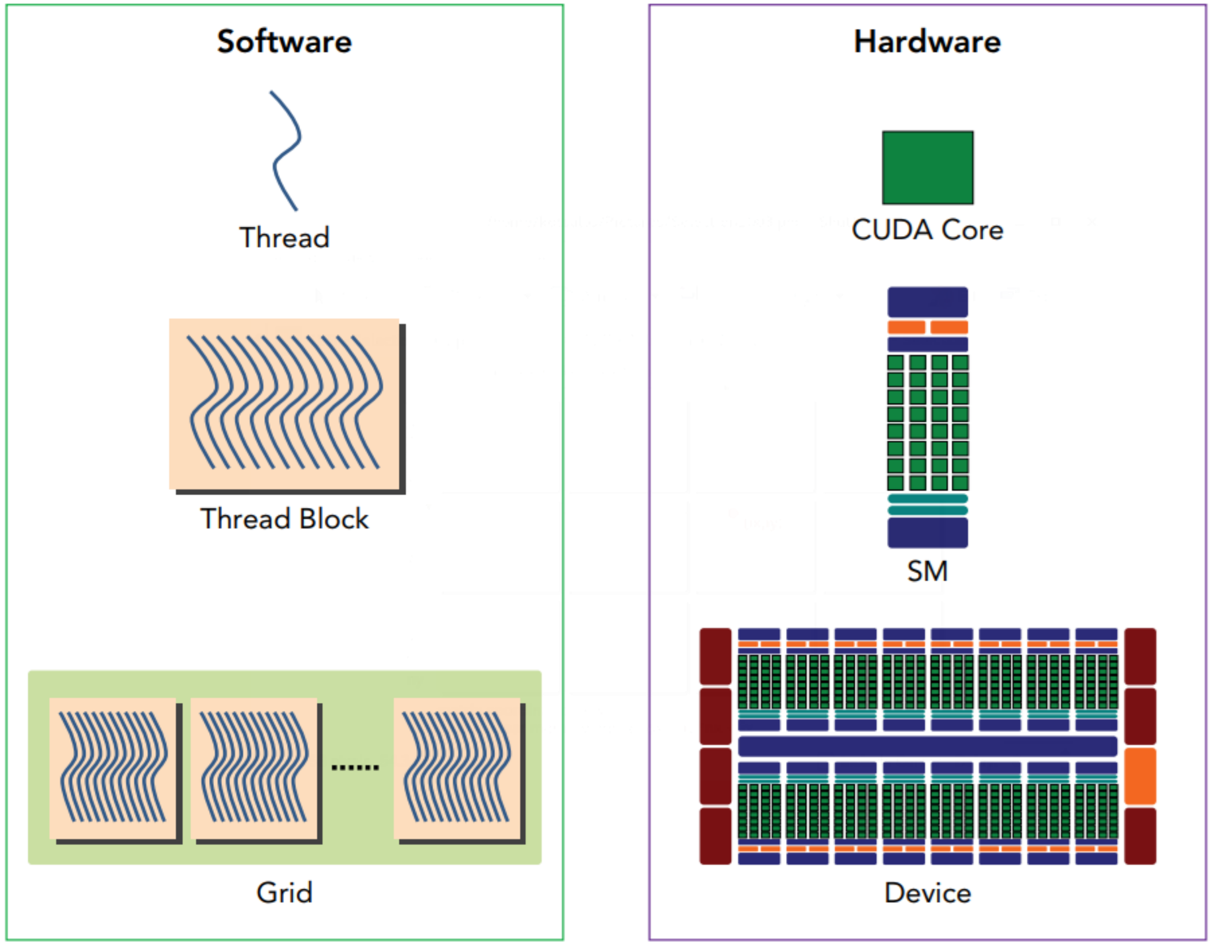
First GPU program

Kernel is the code that is called by the **host** (CPU) and executed by the **device** (GPU)

```
int main()
{
  ...

  //Kernel invocation (grid of blocks of threads)
  VecAdd<<<Blocks,ThreadsPerBlock>>>(A,B,C);

  ...
}
```

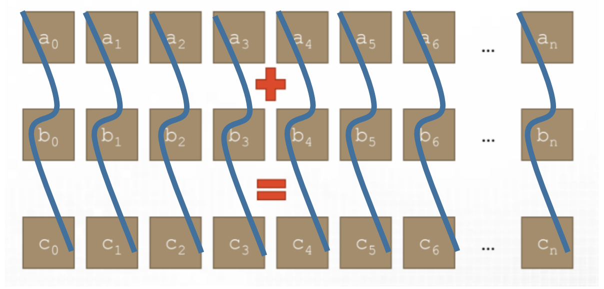
- To maximize performance, we need to fully utilize GPU resources
- Every CUDA core, every Streaming Multiprocessor should be in use
- The kernel acts like a for-loop, whose contents are executed by each thread (CUDA core)
- The goal is to create as many blocks/ threads as possible (even if we exceed the resources) so as to hide latency with computations. A GPU may have 192 CUDA cores but we should create more threads than the available so as, when one thread waits for a memory access, it is replaced by one that does actual computations

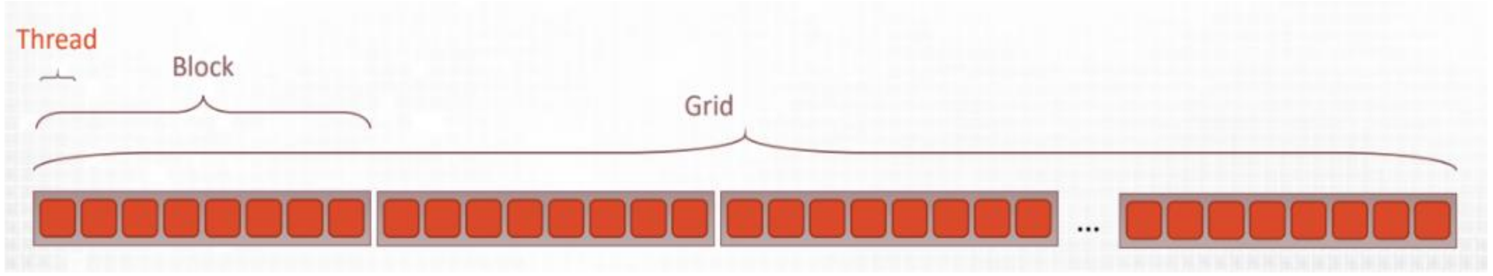

a_0 a_1 a_2 a_3 a_4 a_5 a_6 ... a_n
+
b_0 b_1 b_2 b_3 b_4 b_5 b_6 ... b_n
=
c_0 c_1 c_2 c_3 c_4 c_5 c_6 ... c_n

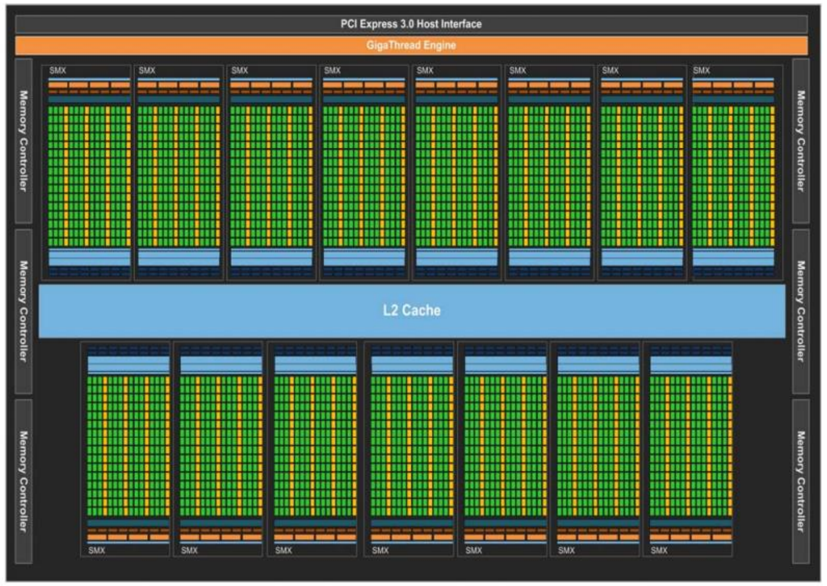


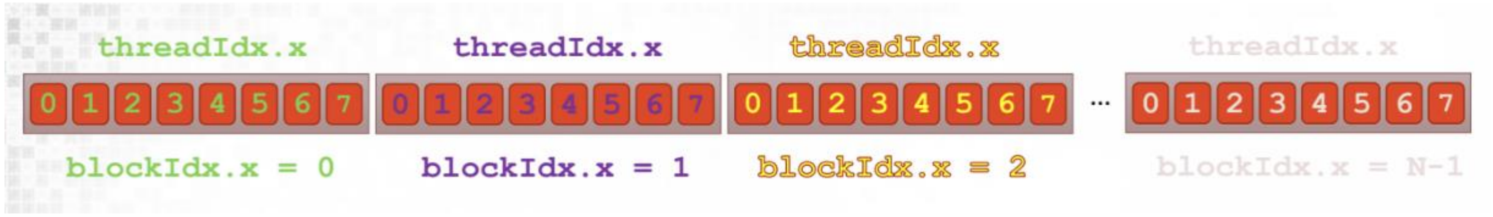
First GPU program

```
VecAdd<<<Blocks,ThreadsPerBlock>>>(A,B,C);

...

// Kernel definition - Executed by every thread
__global__ void VecAdd(float* A, float* B, float* C)
{
    // threadIdx, blockDim, blockIdx: CUDA built-in vars
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i < N) // Size of vectors
      C[i] = A[i] + B[i];
}
```

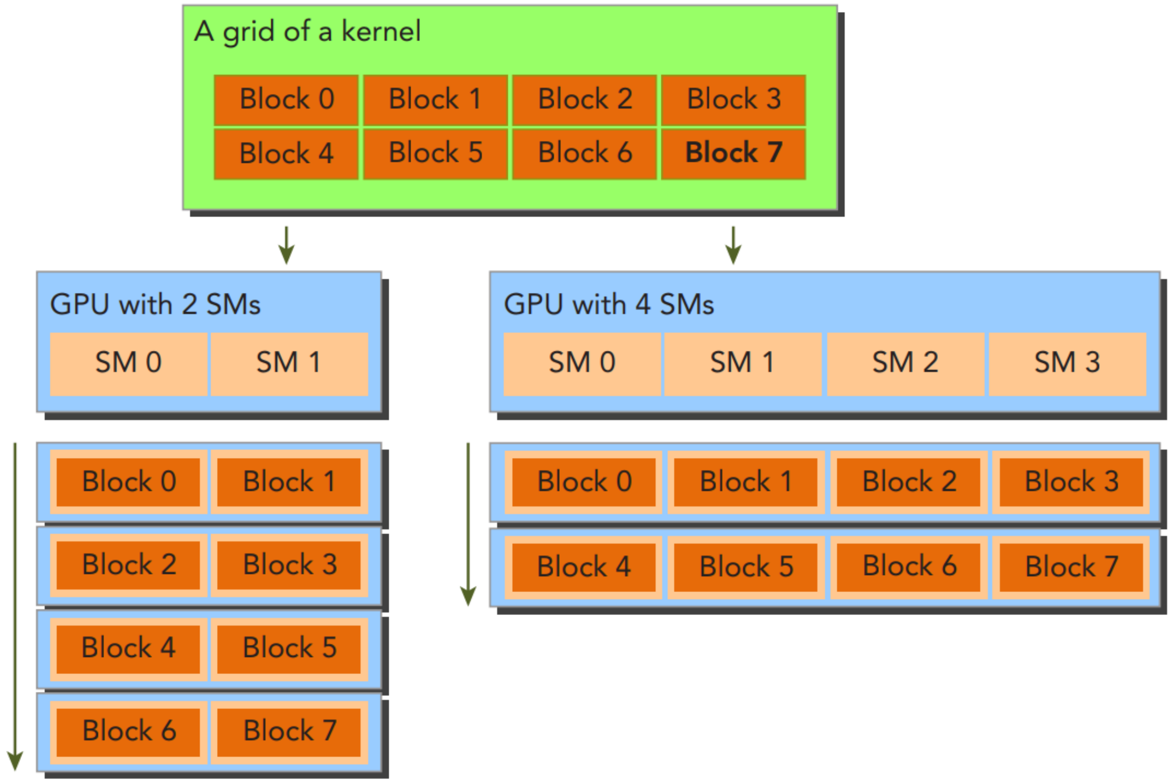

a_0 a_1 a_2 a_3 a_4 a_5 a_6 ... a_n
+
b_0 b_1 b_2 b_3 b_4 b_5 b_6 ... b_n
=
c_0 c_1 c_2 c_3 c_4 c_5 c_6 ... c_n







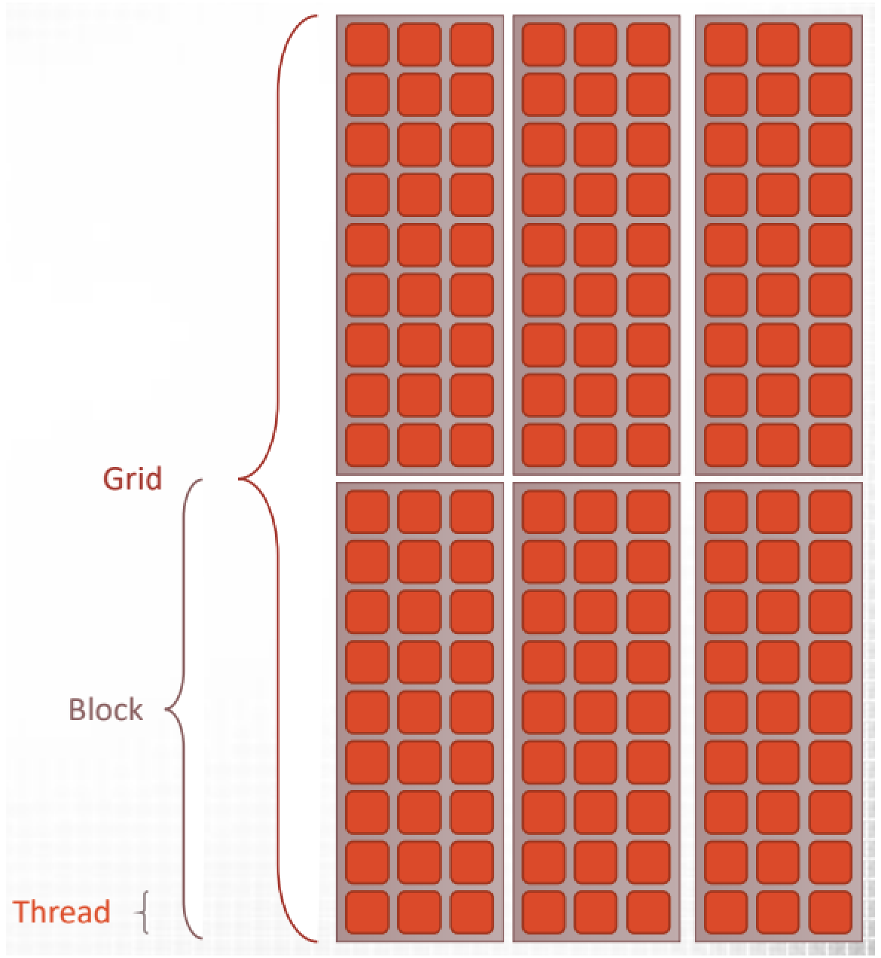
Scheduling

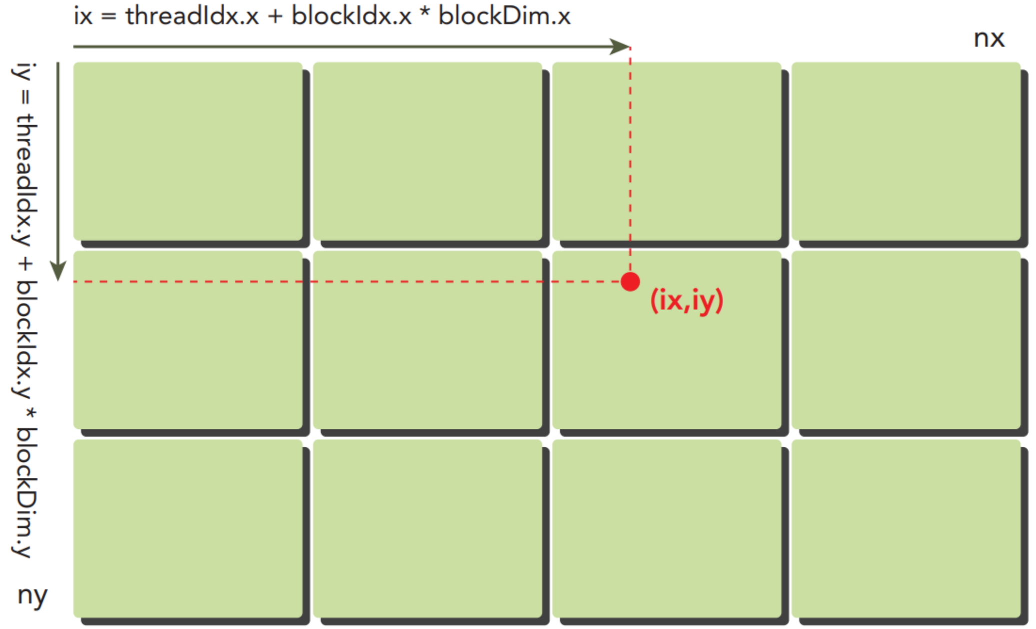


- CUDA will do the scheduling in the background given the available resources
- Remember to create more blocks/ threads than the available (SMs/CUDA cores) to hide latency

PCI Express 3.0 Host Interface
GigaThread Engine
SMX
Memory Controller
L2 Cache

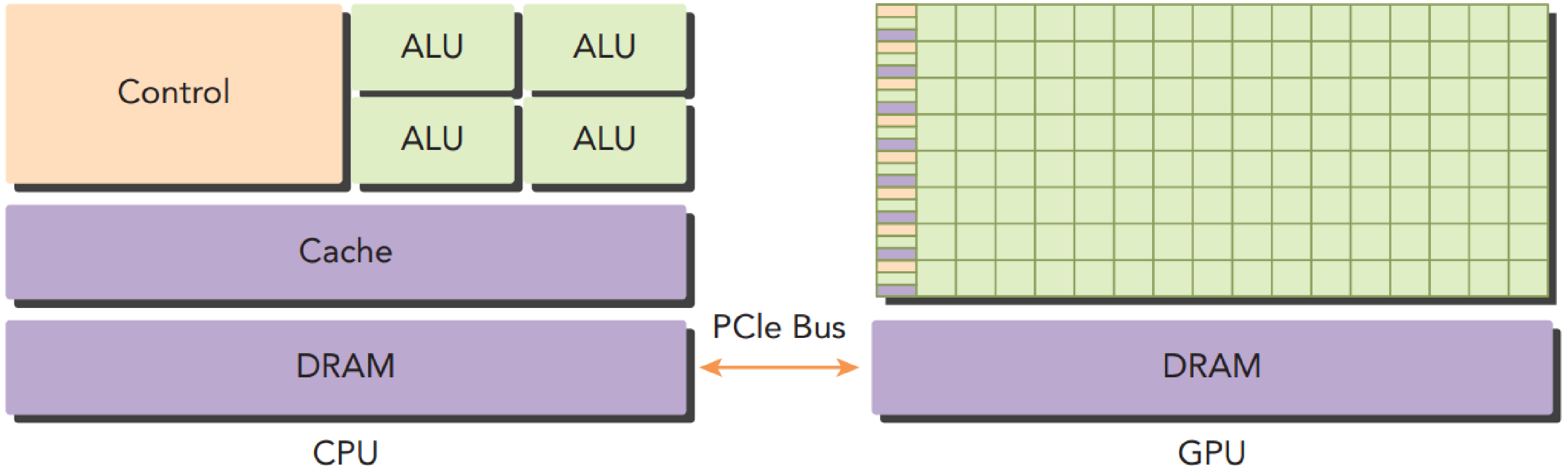
2D Blocks



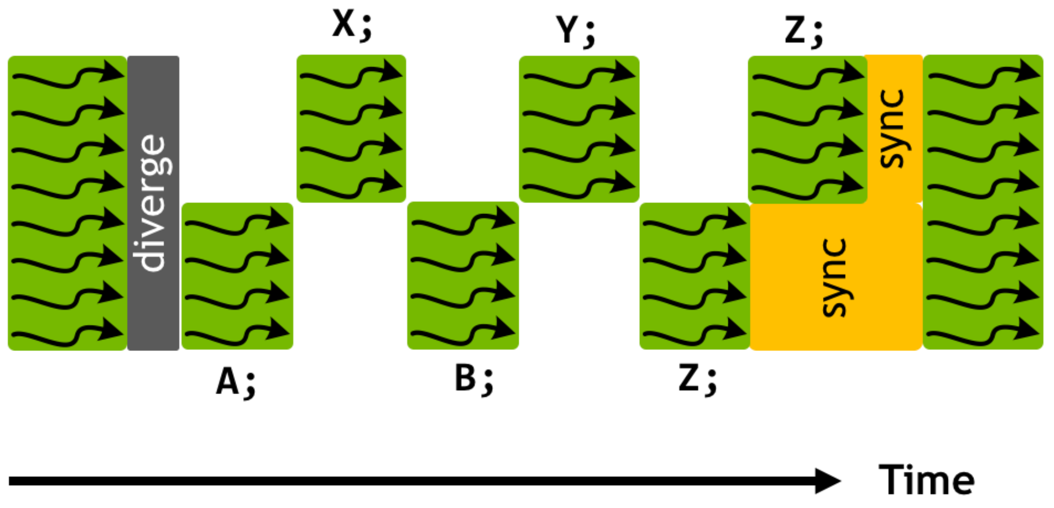

ix = threadIdx.x + blockIdx.x * blockDim.x
nx
iy = threadIdx.y + blockIdx.y * blockDim.y
(ix,iy)
ny

matrix coordinate: (ix,iy)
global linear memory index: idx = iy*nx + ix

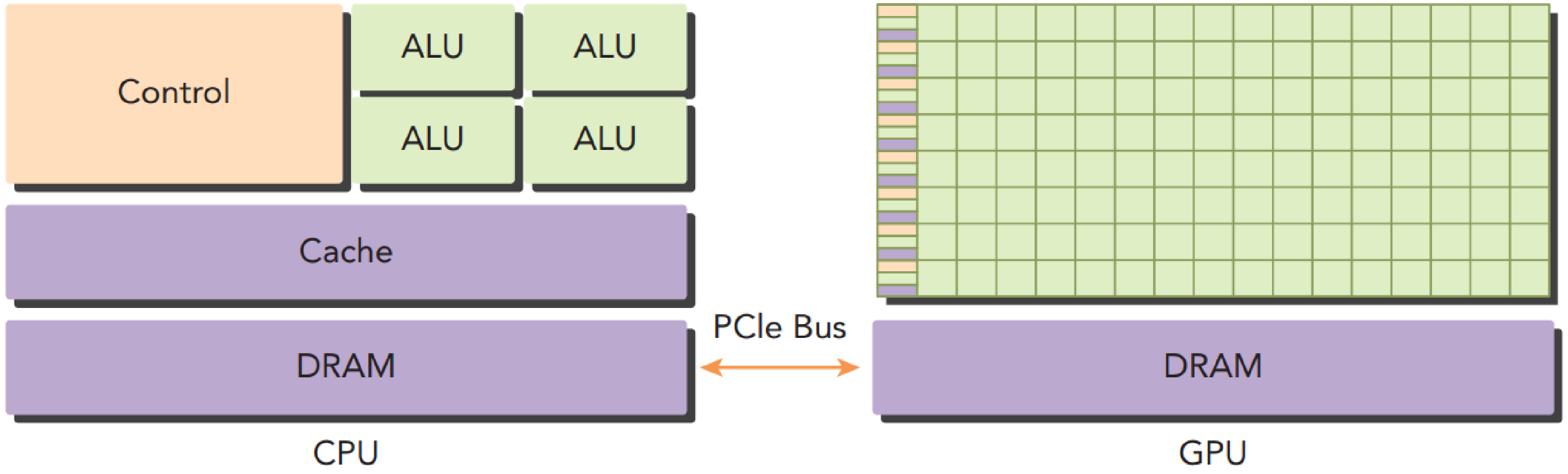
Branch Divergence in GPU



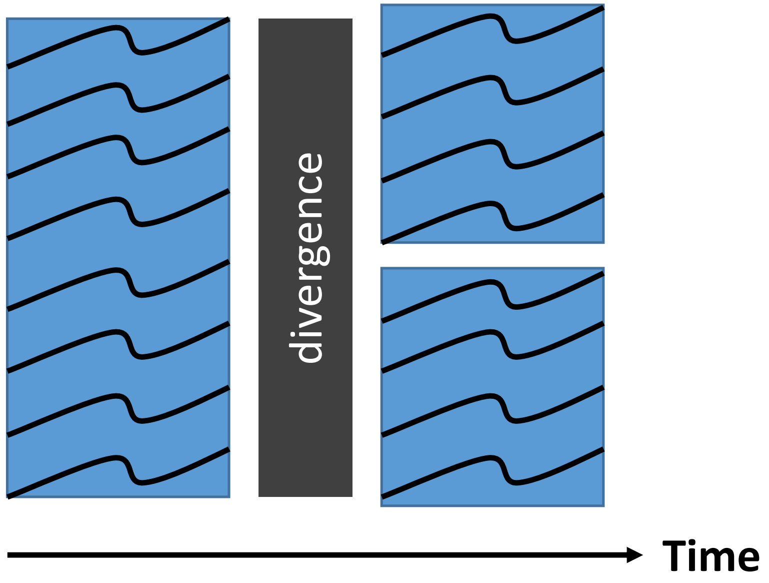
```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
__syncwarp()
```



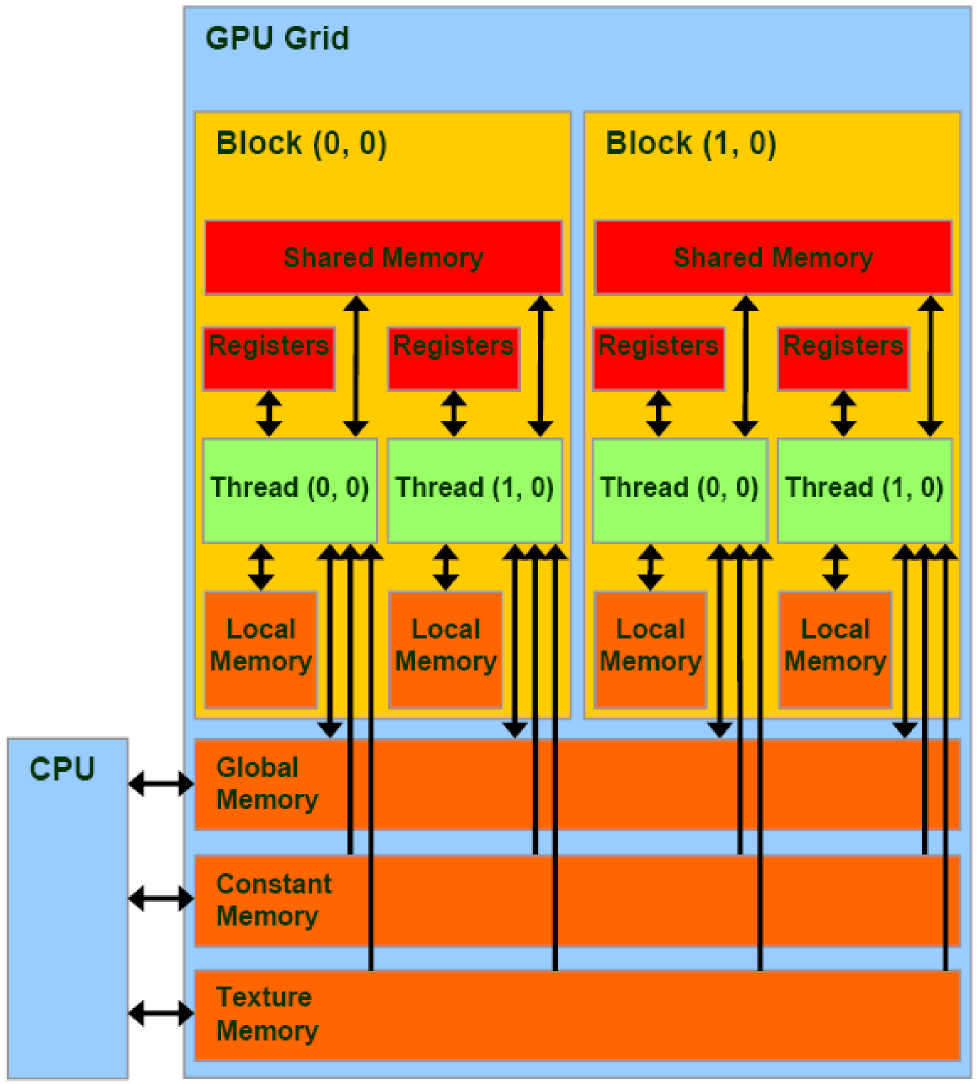
Branch Divergence in CPU



```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
__syncwarp()
```



CUDA Memory Model



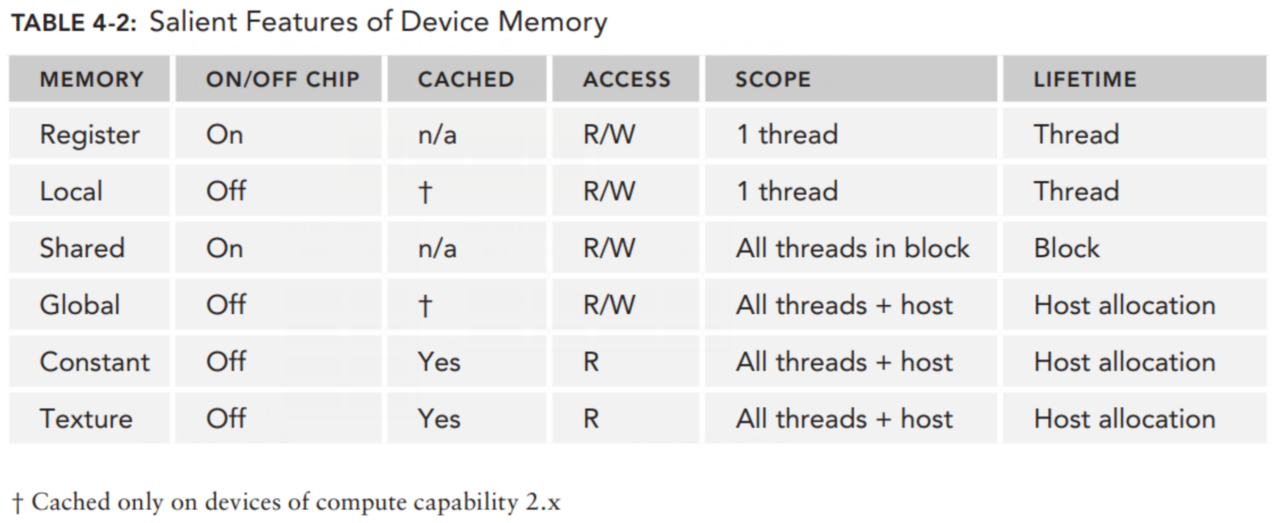


TABLE 4-2: Salient Features of Device Memory

MEMORY	ON/OFF CHIP	CACHED	ACCESS	SCOPE	LIFETIME
Register	On	n/a	R/W	1 thread	Thread
Local	Off	†	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

† Cached only on devices of compute capability 2.x

Off chip access is 100 times slower than on chip

Memory Allocation in action

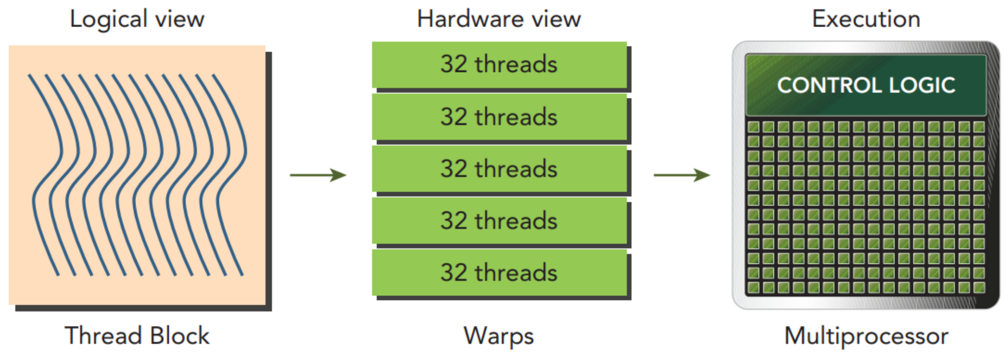
```
#define N 2048
#define THREADS_PER_BLOCK 128

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

int main(void) {
     float *a, *b, *c;                 // host copies of a, b, c
     float *d_a, *d_b, *d_c;           // device copies of a, b, c
     int size = N * sizeof(float);

     cudaMalloc((void **)&d_a, size);
     cudaMalloc((void **)&d_b, size);
     cudaMalloc((void **)&d_c, size);

     a = (float *)malloc(size); random_floats(a, N);
     b = (float *)malloc(size); random_floats(b, N);
     c = (float *)malloc(size);

     cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
     cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

     vectorAdd <<<N / THREADS_PER_BLOCK, THREADS_PER_BLOCK >>>(d_a, d_b, d_c);

     cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

     free(a); free(b); free(c);
     cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
     return 0;
}
```

Warps

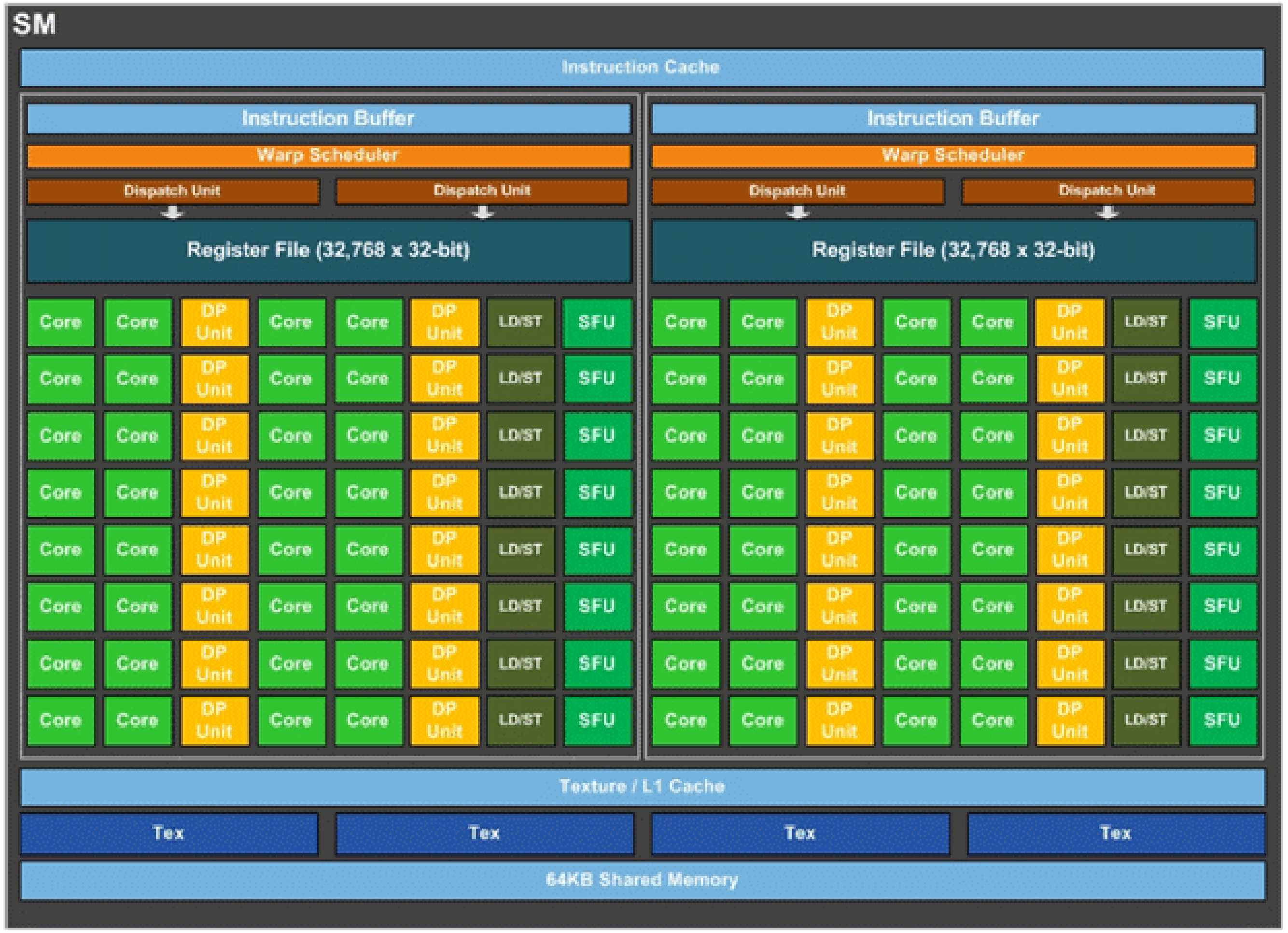
- CUDA Cores multiple of 32
- The blocks of threads are further subdivided into warps (groups of 32 threads)
- Transparent to the user, but can be critical for the performance

The number 32* is a magic number in CUDA programming. It comes from hardware and has a significant impact on the performance of software.
Conceptually, you can think of it as the granularity of work processed simultaneously in SIMD fashion by a SM. Optimizing your workloads to fit within the boundaries of a warp will generally lead to more efficient utilization of GPU compute resources.

* The "32" may change in the future



Streaming Multiprocessor (SM)

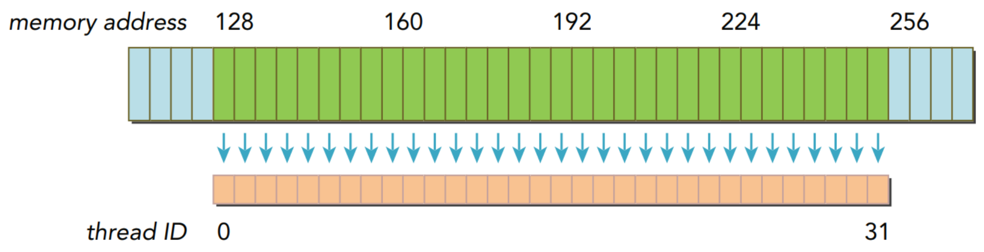


Depending on the number of warp schedulers and available CUDA cores, the warps can be executed concurrently

Optimized Performance

- The threads of the warps should access memory that is contiguous (close in the memory space)

- Threads of the warps should execute the same path (avoid if-branches per warp). Divergence of threads of a warp lead to serialization which impacts performance. Different warps can take whichever branch they want


memory address
128
160
192
224
256
thread ID
0
31