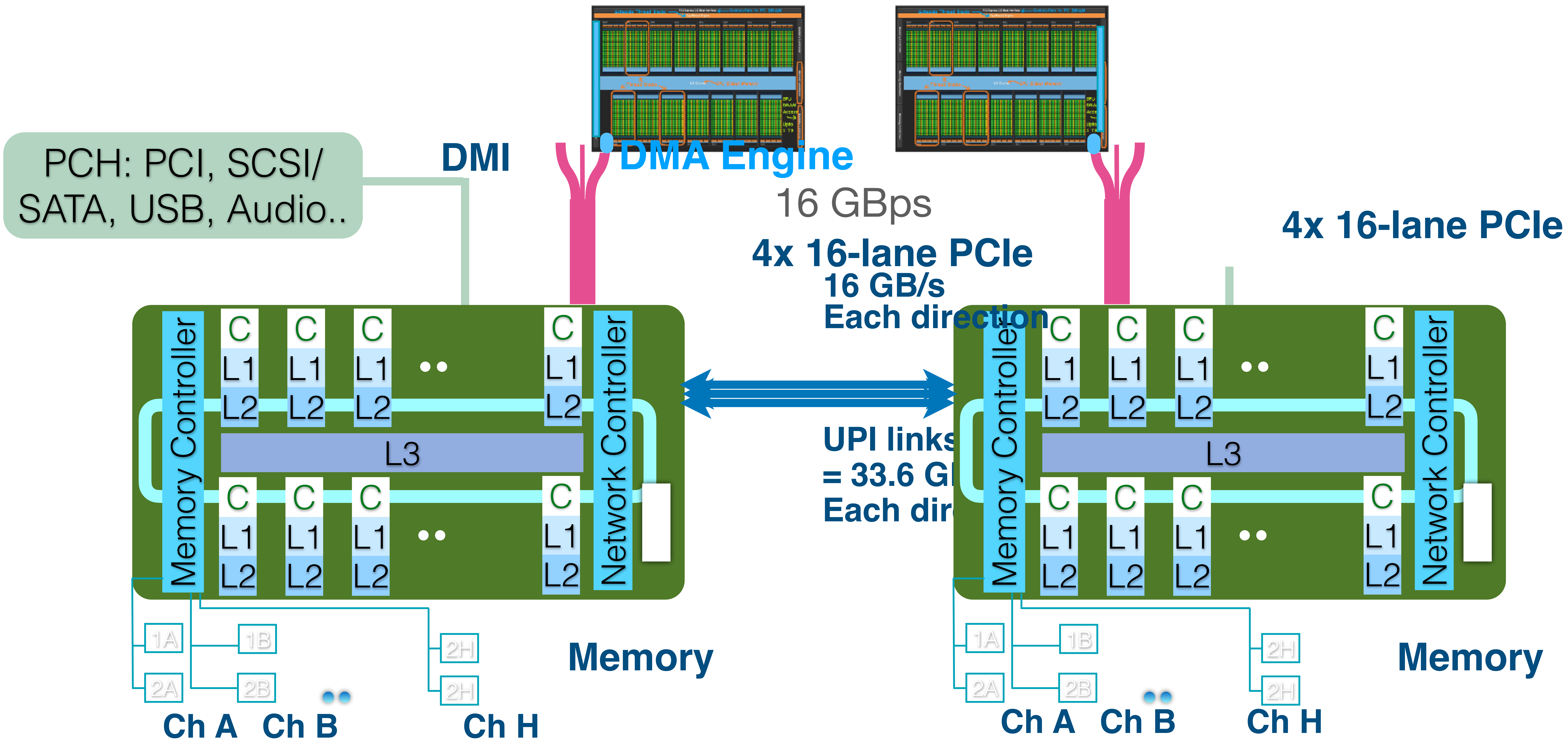
COL380

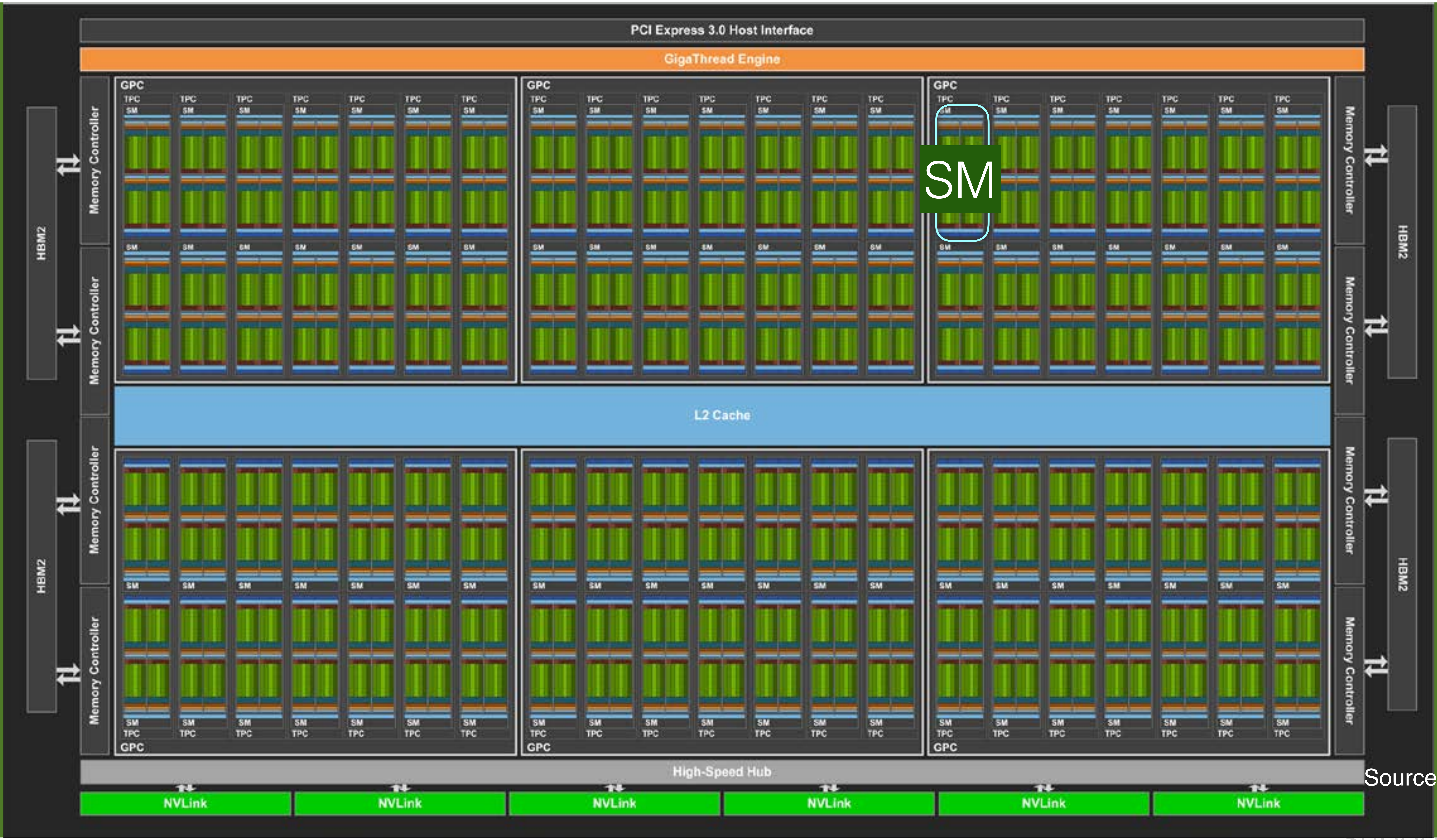
Introduction to Parallel & Distributed Programming

GPU and manycore

CPU+GPU

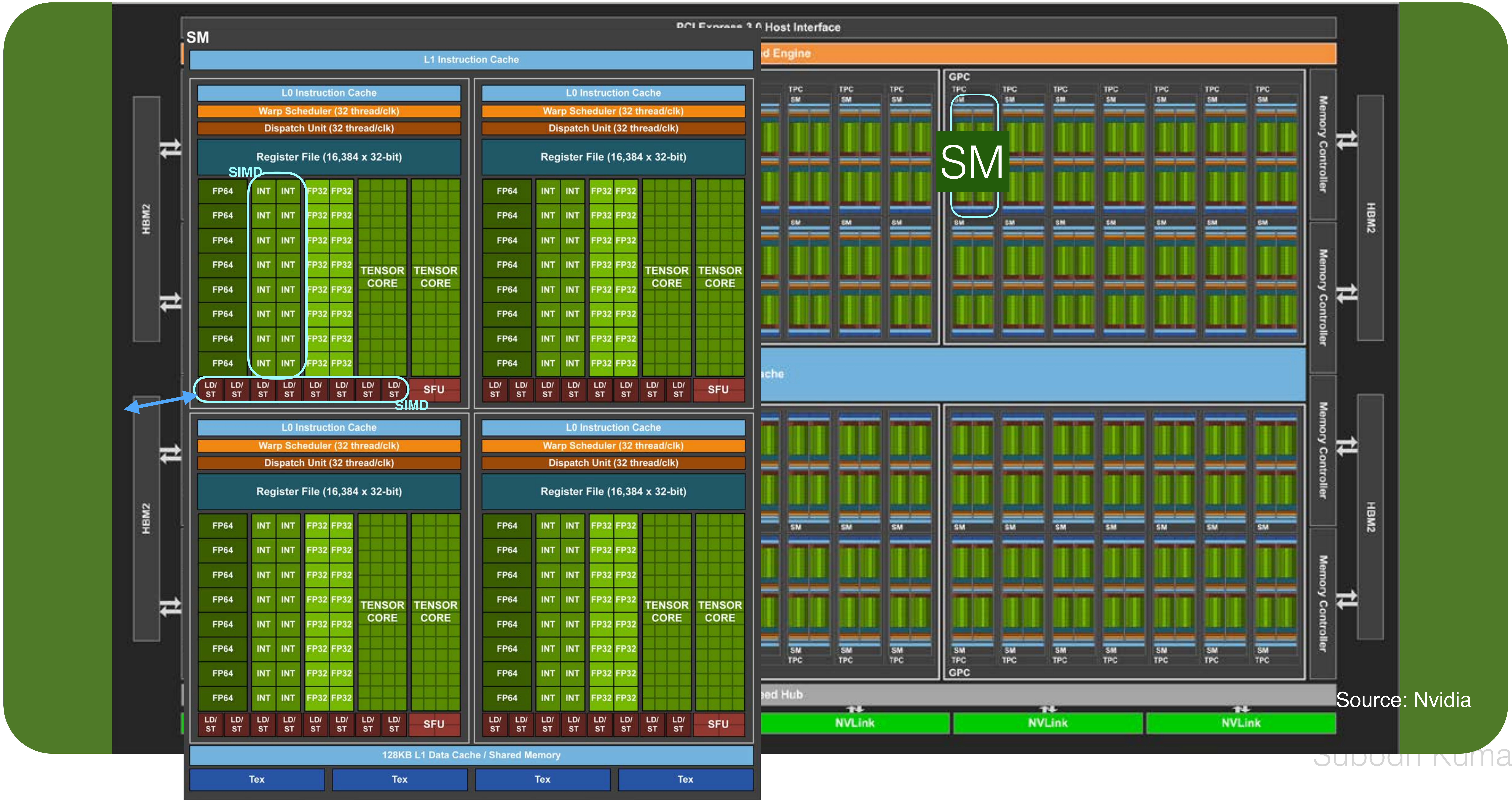


Subodh Kumar



Source: Nvidia

GPU



PCI Express 3.0 Host Interface
SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
SIMD
FP64
INT
FP32
TENSOR CORE
LD/ST
SFU
SIMD
128KB L1 Data Cache / Shared Memory
Tex
HBM2
GPC
TPC
SM
Memory Controller
NVLink

Source: Nvidia

Offload Model?

```
#pragma omp target teams num_teams(n) distribute
#pragma omp parallel for
  for(int i=0; i<N; i++)
       vec2[i] += vec1[i];
```

```
accelerate(sum, size, vec, vec2);
```

```
sum (const int size, __global float * vec1, __global float * vec2)
{
       int ii = get_global_id(0);
       if (ii < size) vec2[ii] += vec1[ii];
}
```

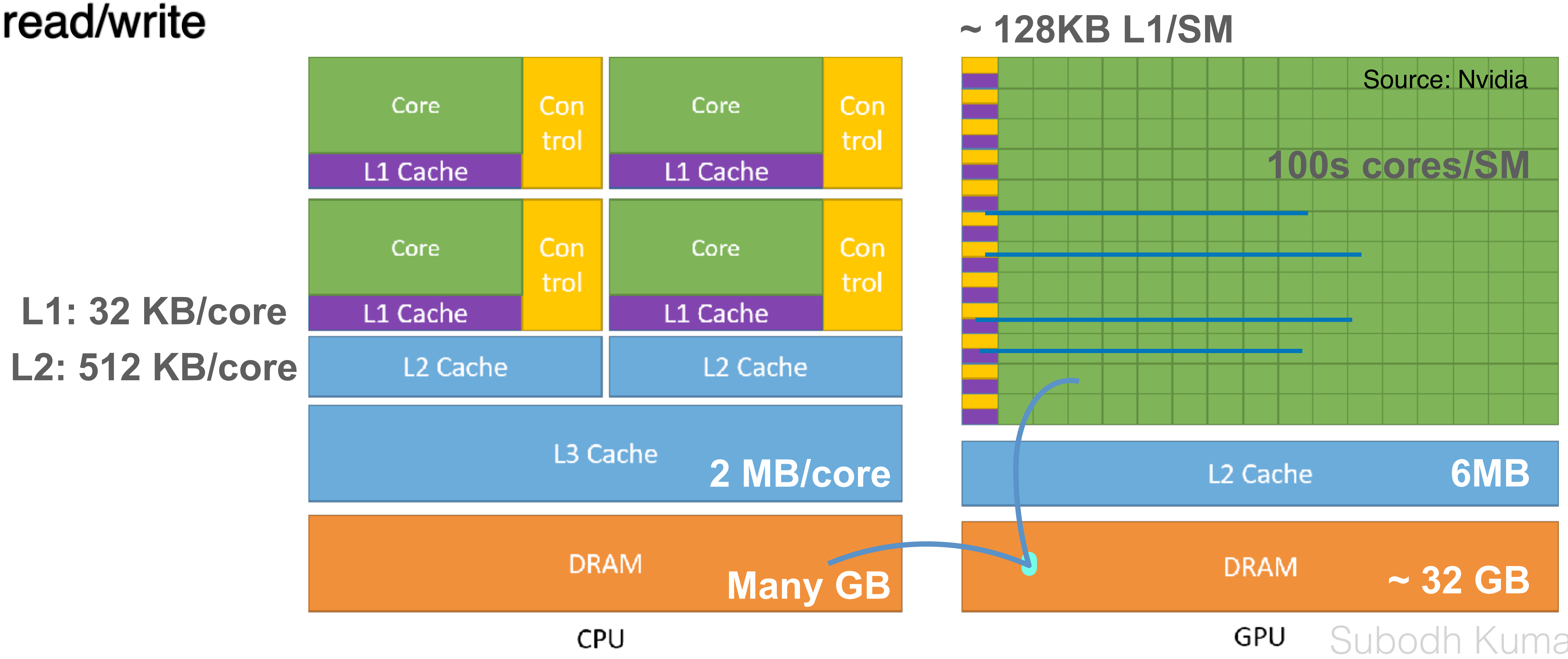
Matrix Multiplication

// Matrix multiplication kernel – per thread code

```
__global__ void
MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width
{
    // Pvalue stores the matrix element computed by the thread
    float Pvalue = 0;
    for (int k = 0; k < Width; ++k) {
        float Melement = Md[threadIdx.y*Width+k];
        float Nelement = Nd[k*Width+threadIdx.x];
        Pvalue += Melement * Nelement;
    }

    Pd[threadIdx.y*Width+threadIdx.x] = Pvalue;
}
```

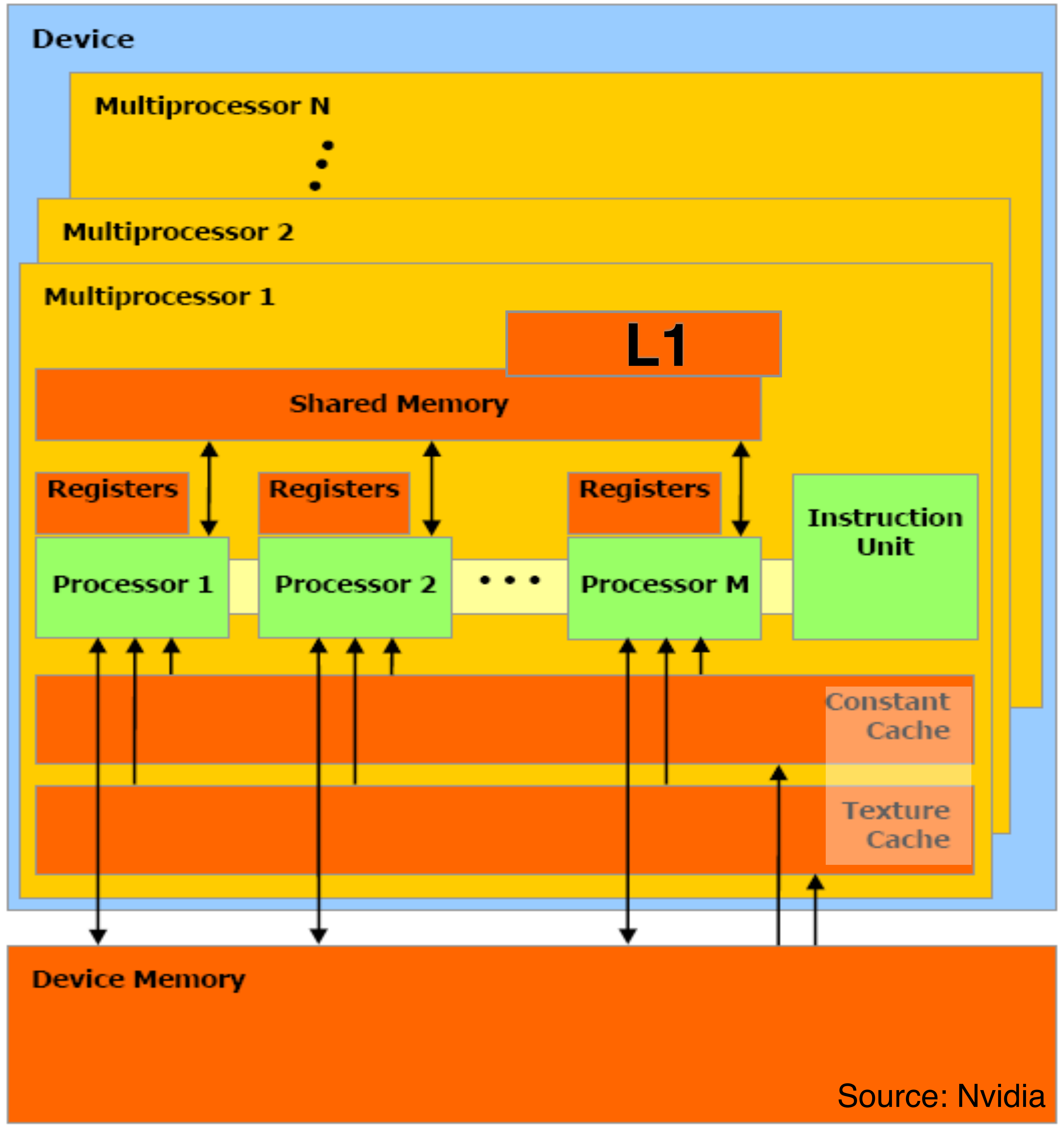
CPU vs GPU architecture

- Run many threads
  - Memory latency needs to be hidden
  - SIMD => Explicit parallel read/write



Subodh Kumar

Architecture Model



Subodh Kumar

CUDA Programming Model

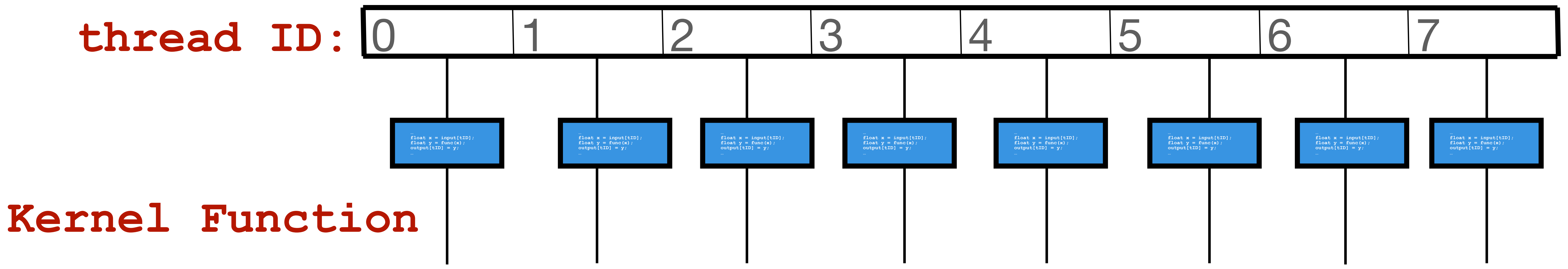
- Co-processor
  - Many cores
  - CPU code offloads multi-threaded tasks
    - Message passing and shared memory models
- GPU Threads are organized hierarchically
  - Grids, Blocks, Warps
  - Data-parallelism focus
- Shared memory
  - Memory hierarchy exposed
  - Parallel read/write

Device & Threads

- Each GPU is also called a device
  - CPU is host
- GPU memory is device memory
- Device executes GPU code (kernel)
  - Executed by many threads in parallel
- GPU is massively parallel
  - Thousands of cores
- GPU threads are lightweight
  - Very little creation and scheduling overhead
  - Context remains live
  - Many more threads then the number if processors

Thread Organization

- CPU thread ‘forks’ a grid of threads, each executes the kernel
  - Grid executes asynchronously, CPU thread continues
  - Grid has a number of blocks, each block has a number of threads
    - Threads execute in groups of 32 called warps
    - Execute as SIMT: Single instruction multiple threads
      - But a warp can diverge



```
…
float x = input[tID];
float y = func(x);
output[tID] = y;
…
```

```
…
float x = input[tID];
float y = func(x);
output[tID] = y;
…
```

```
…
float x = input[tID];
float y = func(x);
output[tID] = y;
…
```

```
…
float x = input[tID];
float y = func(x);
output[tID] = y;
…
```

```
…
float x = input[tID];
float y = func(x);
output[tID] = y;
…
```

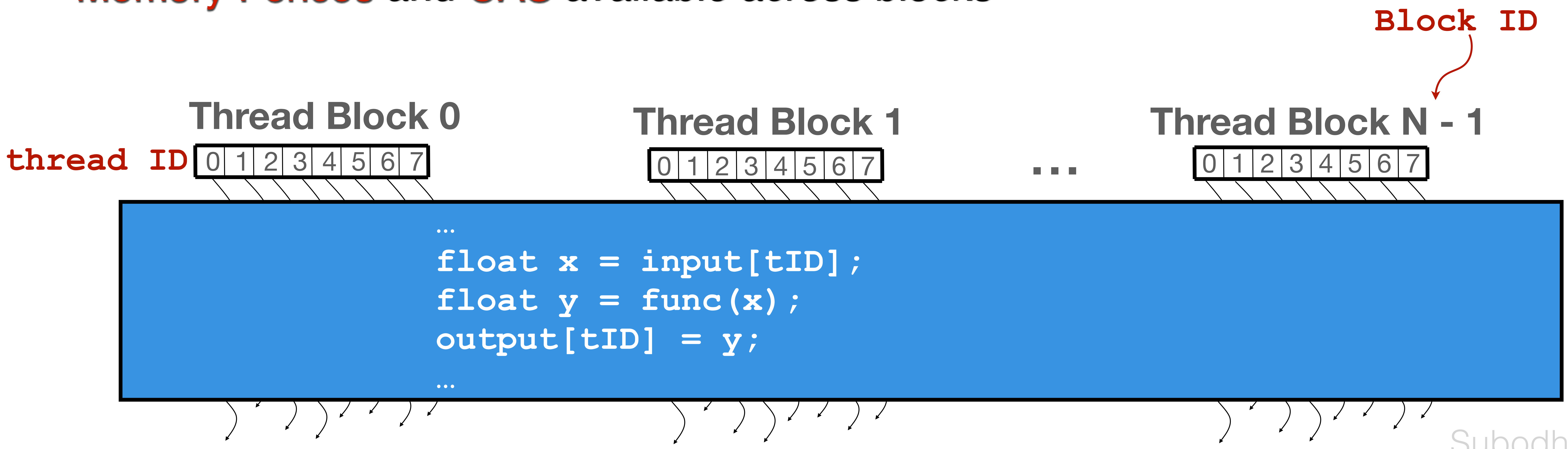
```
…
float x = input[tID];
float y = func(x);
output[tID] = y;
…
```

```
…
float x = input[tID];
float y = func(x);
output[tID] = y;
…
```

```
…
float x = input[tID];
float y = func(x);
output[tID] = y;
…
```

Thread Blocks

- Threads within a block share shared memory and global memory
  - Threads in different blocks only access a common global memory
  - Intra-warp register-based consensus
- Barrier and finer-grained synchronization only within warps and blocks
  - Memory Fences and CAS available across blocks

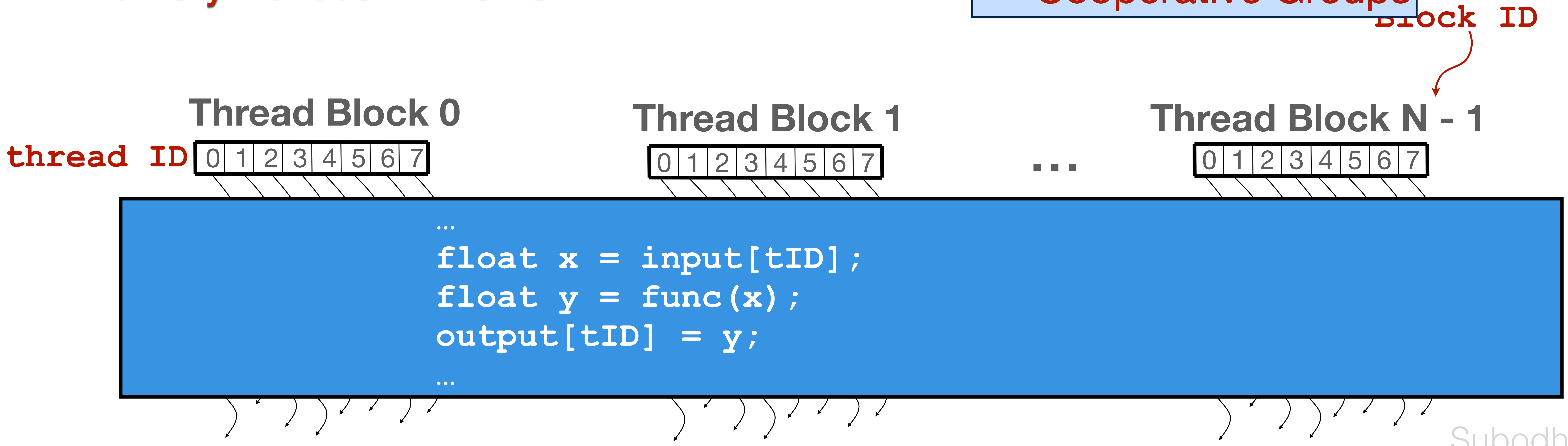


```
…
float x = input[tID];
float y = func(x);
output[tID] = y;
…
```

Thread Blocks

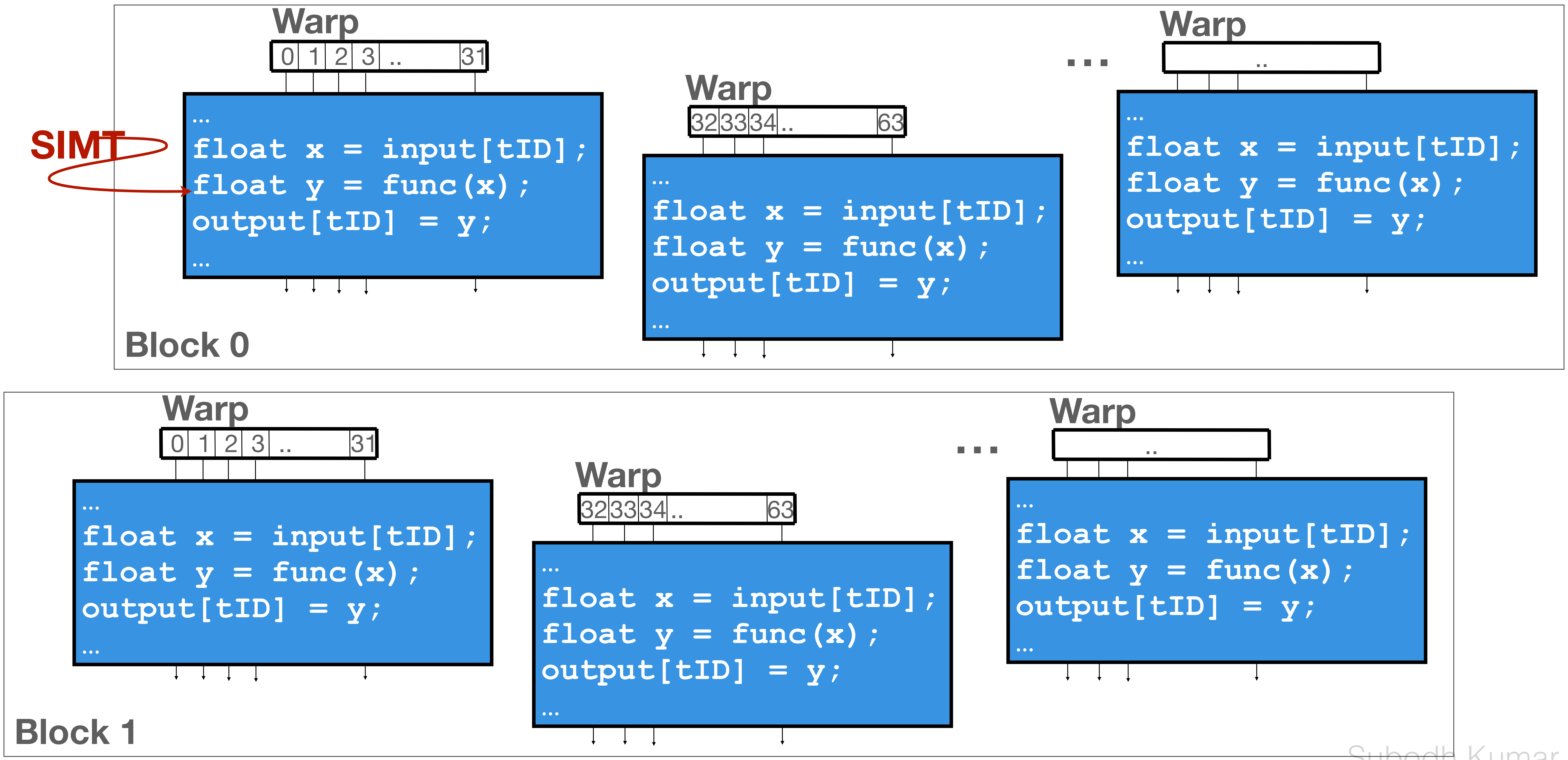
- Threads within a block share shared memory and global memory
  - Threads in different blocks only access a common global memory
  - Intra-warp register-based consensus
- Barrier and finer-grained synchronization only within warps and blocks
  - Memory Fences and CAS available across blocks

Also see:
Cooperative Groups

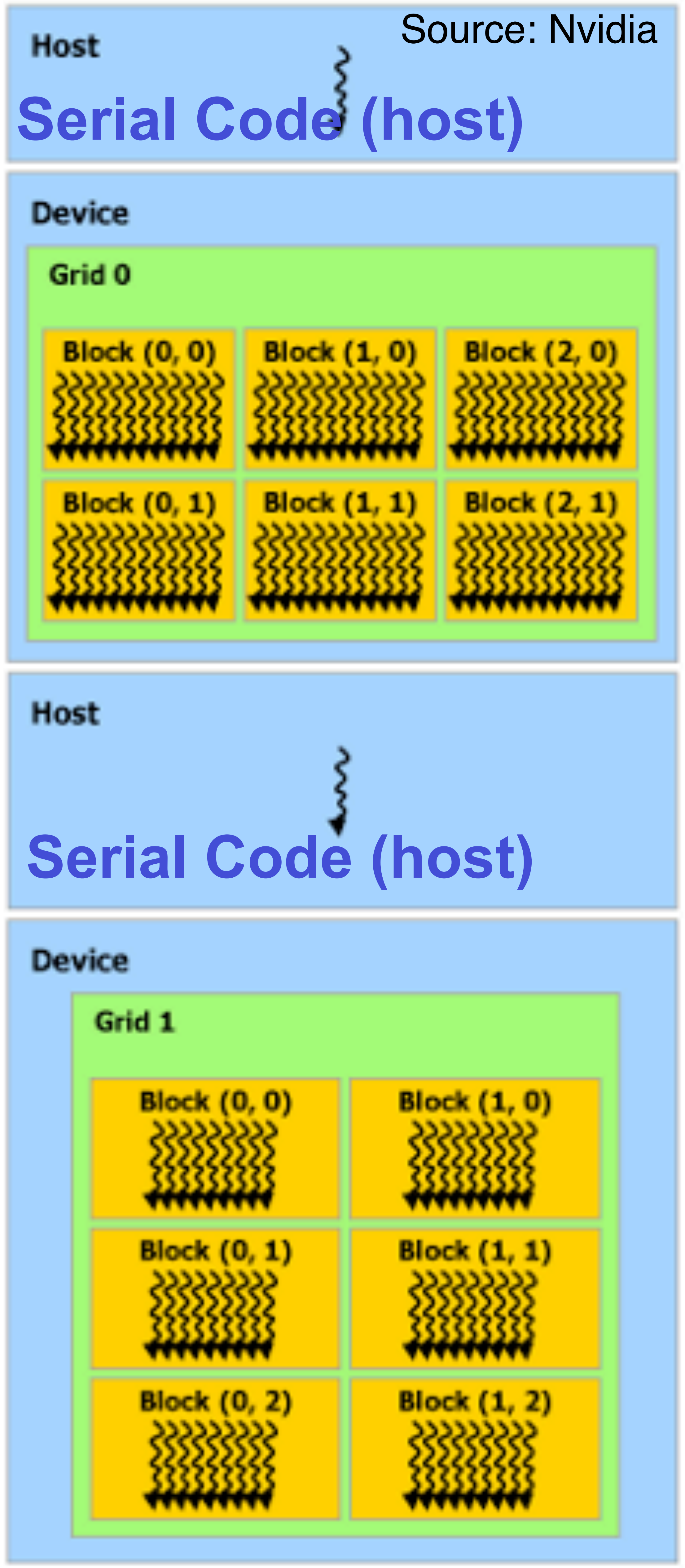


```
…
float x = input[tID];
float y = func(x);
output[tID] = y;
…
```

Warps


Warp
0 1 2 3 .. 31
SIMT
...
float x = input[tID];
float y = func(x);
output[tID] = y;
...
Warp
32 33 34 .. 63
...
float x = input[tID];
float y = func(x);
output[tID] = y;
...
...
Warp
..
...
float x = input[tID];
float y = func(x);
output[tID] = y;
...
Block 0
Warp
0 1 2 3 .. 31
...
float x = input[tID];
float y = func(x);
output[tID] = y;
...
Warp
32 33 34 .. 63
...
float x = input[tID];
float y = func(x);
output[tID] = y;
...
...
Warp
..
...
float x = input[tID];
float y = func(x);
output[tID] = y;
...
Block 1

CUDA is C-like



Execute on GPU:

KernelA<<< nBlk, nTid >>>(args);

Execute on GPU

KernelB<<< nBlk, nTid >>>(args);

- Integrated host+device app Cuda program
  - Serial or modestly parallel parts in host C code
  - Highly parallel parts in device Cuda code

Extended C

- Declspecs
  - global, device, managed, shared, local, constant
- Built-in variables
  - threadIdx, blockIdx, blockDim
- Intrinsics
  - __syncthreads
- Runtime API
  - Memory, symbol, execution management
- Kernel launch

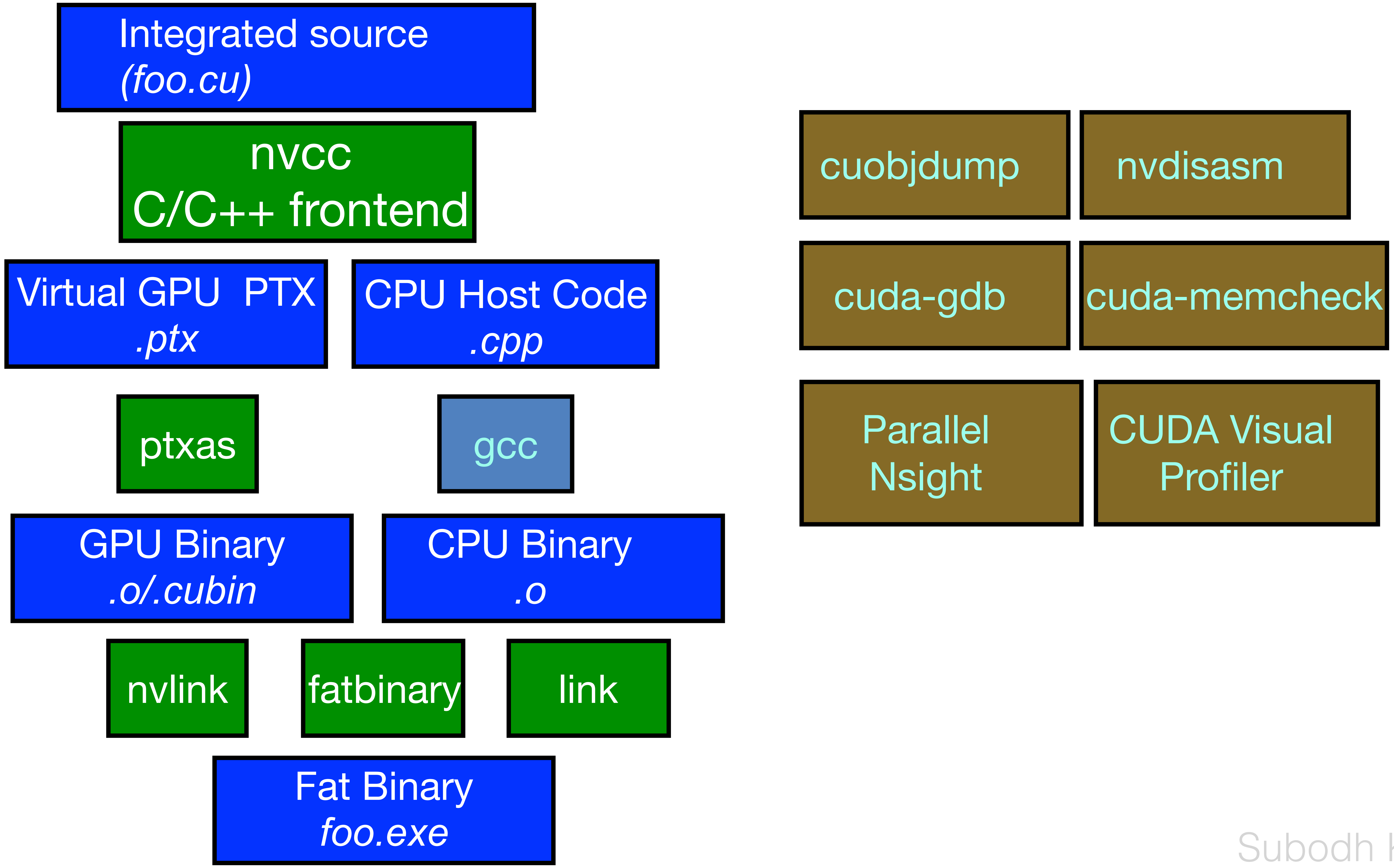
Built-in Types:
char2, int4,
__half2, dim3

```
__device__ float filter[N];

__global__ void convolve (float *image){

    __shared__ float region[M];
    ...
   region[threadIdx.x] = image[i];
   __syncthreads()
   ...
   image[j] = result;
}
...
// Allocate GPU memory
void *myimage = cudaMalloc(bytes)

// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

CUDA Tool-chain

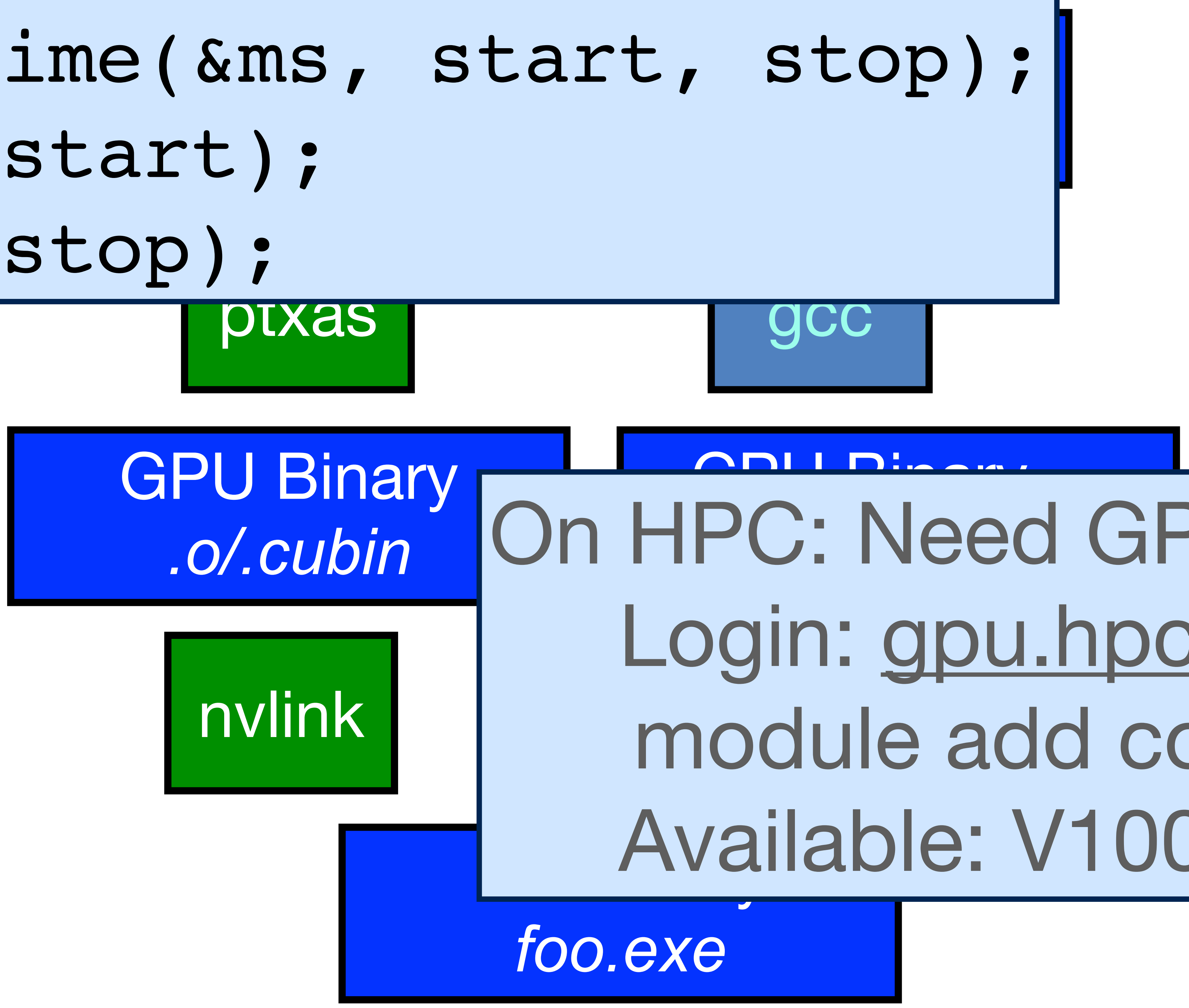
- Source files have a mix of host and device code
- nvcc separates device code from host code
  - compiles device code into PTX/cubin
  - host code is output as C source (and invoke compiler)
- Applications link to the generated host code
  - host code includes PTX/cubin code as a global initialized data array
  - and cudart (CUDA C runtime) function calls to load and launch kernels
- Possible to load and execute the PTX/cubin using the CUDA driver API

Cuda Dev Tools



Cuda Dev Tools

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);
GPUThings();
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float ms;
cudaEventElapsedTime(&ms, start, stop);
cudaEventDestroy(start);
cudaEventDestroy(stop);
```



cuobjdump
nvdisasm
cuda-gdb
cuda-memcheck
Parallel Nsight
CUDA Visual Profiler

On HPC: Need GPU nodes

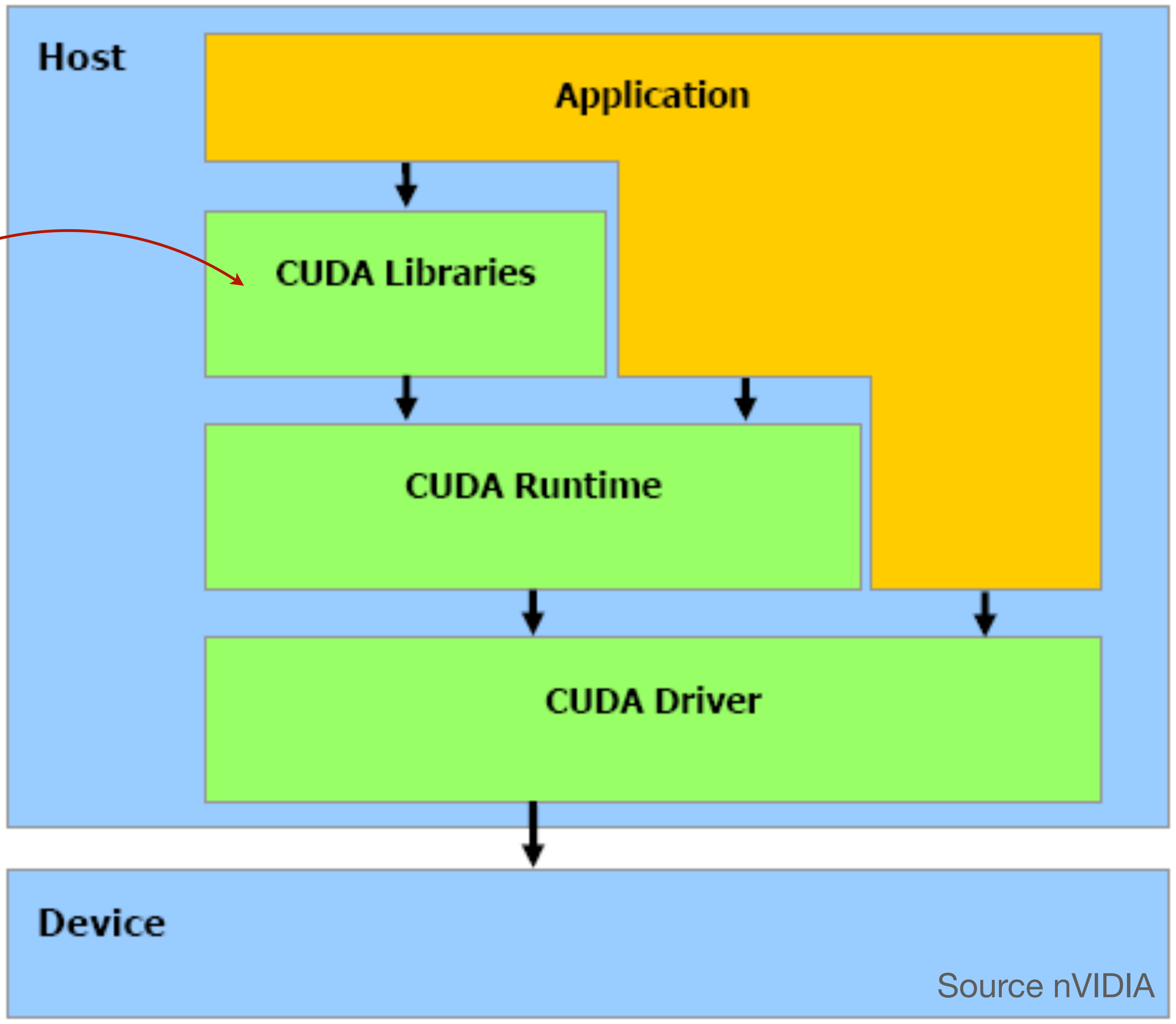
Login: gpu.hpc.iitd.ac.in (K40, CC3.5)

module add compiler/cuda/11.0/compilervars

Available: V100 (CC7.0)

CUDA runtime tools

Provides library functions for host as well as device
Implement subset of stdlib



Invoke GPU Kernel

- Run *k* instances of kernel function *f*
  - Each instance in a thread
  - Kernel 'pushed' to GPU
    - ➡ Declared with `__global__` specifier
- *k* threads are organized into blocks

```
int main()
{
    cpuCode();

    f<<<1, N>>>(F); // GPU Kernel
}
```

Invoke GPU Kernel

- Run *k* instances of kernel function *f*
  - Each instance in a thread
  - Kernel 'pushed' to GPU
    - ➡ Declared with `__global__` specifier
- *k* threads are organized into blocks

```
int main()
{
    cpuCode();

    f<<<1, N>>>(F); // GPU Kernel
}
```

```
// Kernel definition
__global__ void f(float* F)
{
    int id = threadIdx.x;
    …
}
```

Invoke GPU Kernel

- Run *k* instances of kernel function *f*
  - Each instance in a thread
  - Kernel ‘pushed’ to GPU
    - ➡ Declared with `__global__` specifier
- *k* threads are organized into blocks

```
int main()
{
    cpuCode();

    f<<<1, N>>>(F); // GPU Kernel
}
```

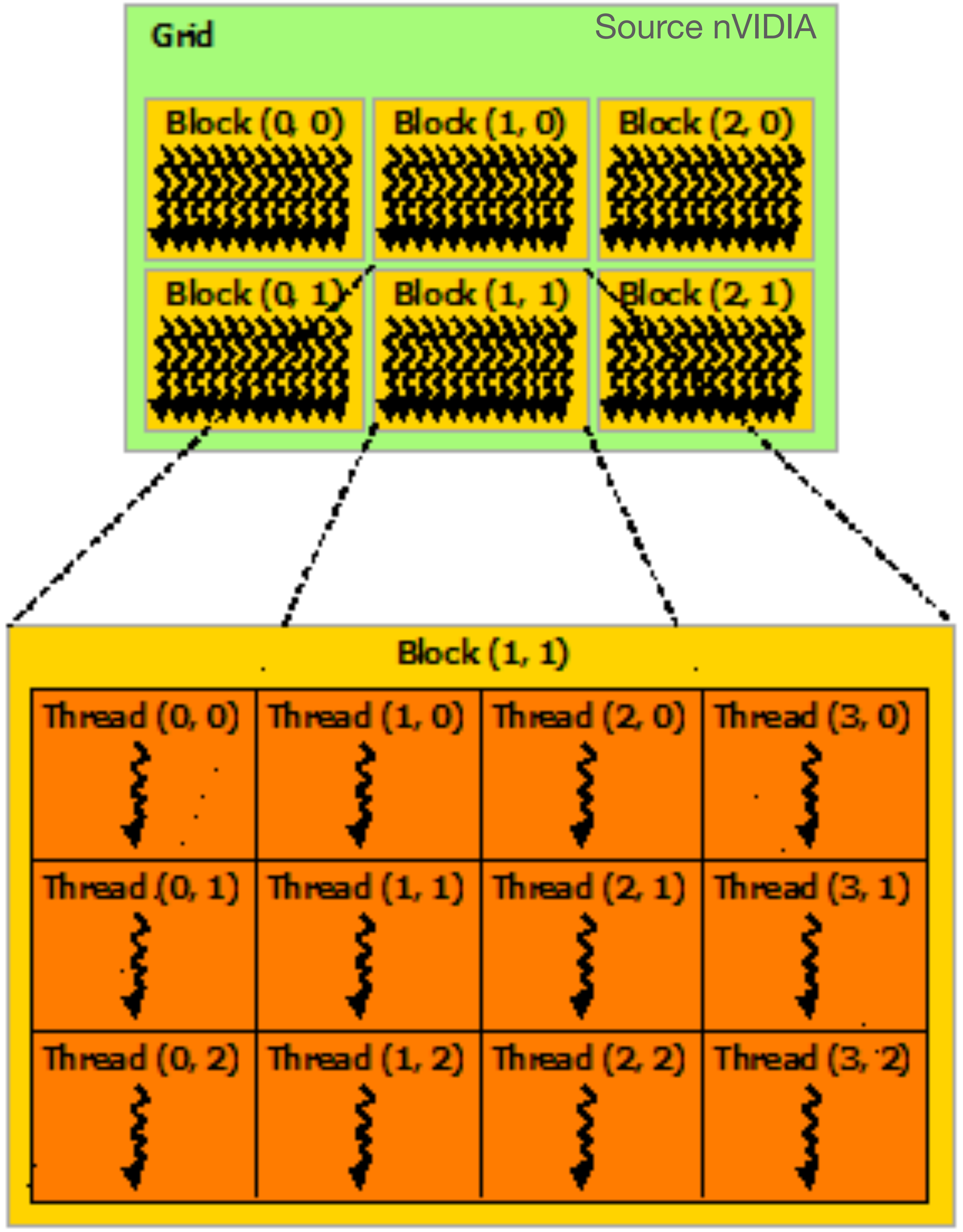
```
// Kernel definition
__global__ void f(float* F)
{
    int id = threadIdx.x;
    …
}
```

Also see: cudaGraphcudaGraphCreate(..)

Specify task-graph of Kernels, CPU functions, memory operations, Synchronization

Grid/Block Dimension

- **Invocation: <<<A,B>>>**
- **A** and **B** can be (up to) 3-D vectors
  - **dim3 B**(a, b, c); // a,b,c are ints
    - ➡ a x b x c <= 1024 for blocks
    - c is the most significant dimension, a the least
    - Dereference: **B**.x, **B**.y and **B**.z
    - Thread ID = (**B**.x + **B**.y * a + **B**.z * a*b)
    - Inbuilt: **dim3** threadIdx, blockIdx, blockDim;



Thread Execution

- All threads of a block need not execute in SIMD fashion
- Threads executed in groups of $\overset{\text{upto}}{\wedge}$ 32 parallel threads (There need not be 32 physical cores)
  - called ***warps***
- All threads of a warp start together
  - But may diverge due to conditional branching
  - Different sub-warps are serialized until they converge back See __syncwarp()
  - Important efficiency consideration