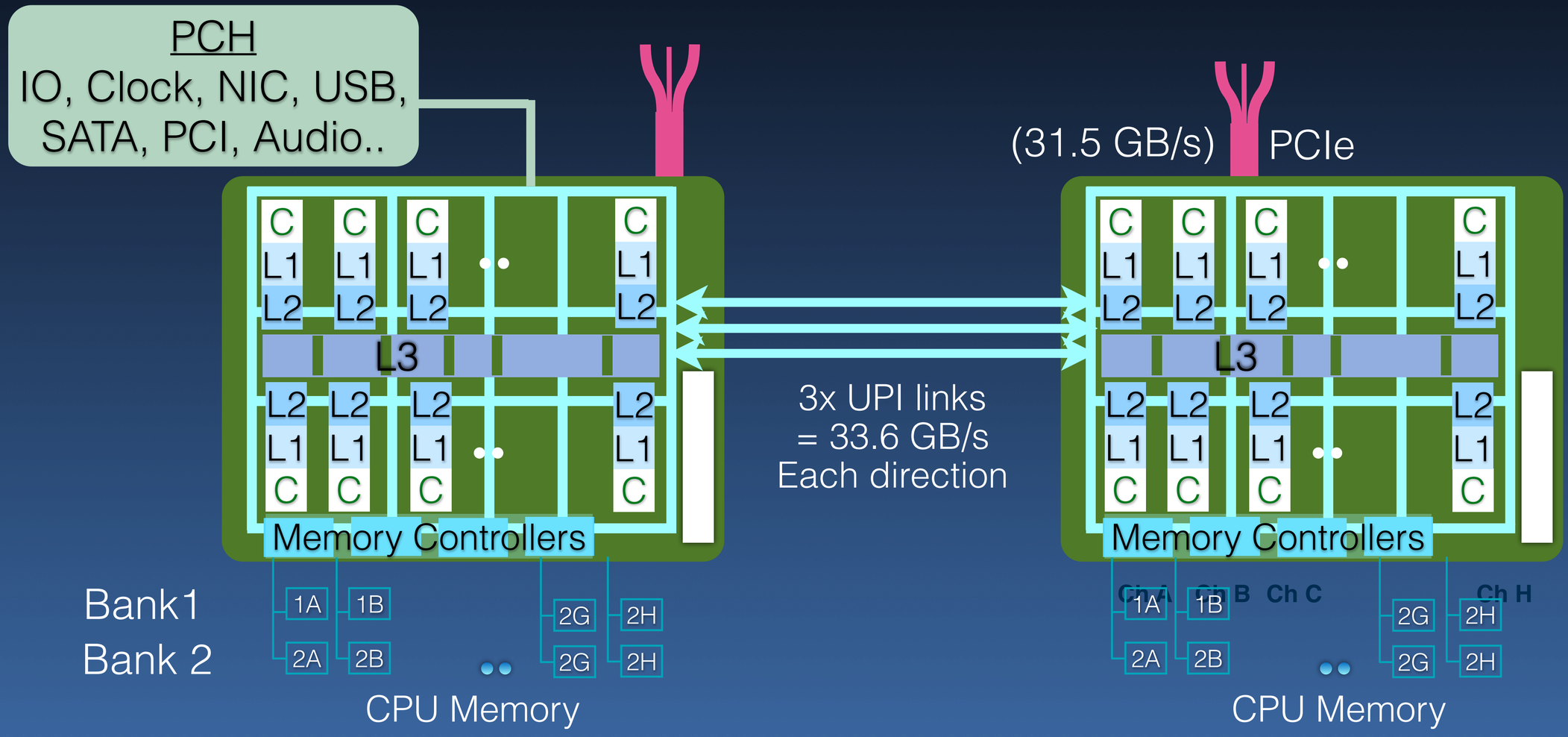
COL380

Introduction to
Parallel & Distributed Programming

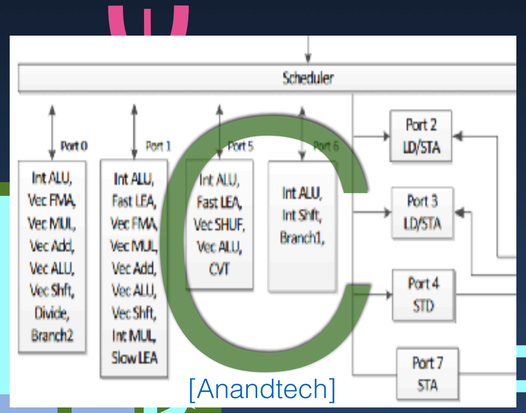
Modern Multi-Processor



Modern Multi-Processor

PCH
IO, Clock, NIC, USB, SATA, PCI, Audio..

(31.5 GB/s) PCIe



[Anandtech]

C L1 L2 L3 Memory Controllers

3x UPI links
= 33.6 GB/s
Each direction

Bank1
Bank 2

1A 1B 2A 2B 2G 2H 2G 2H

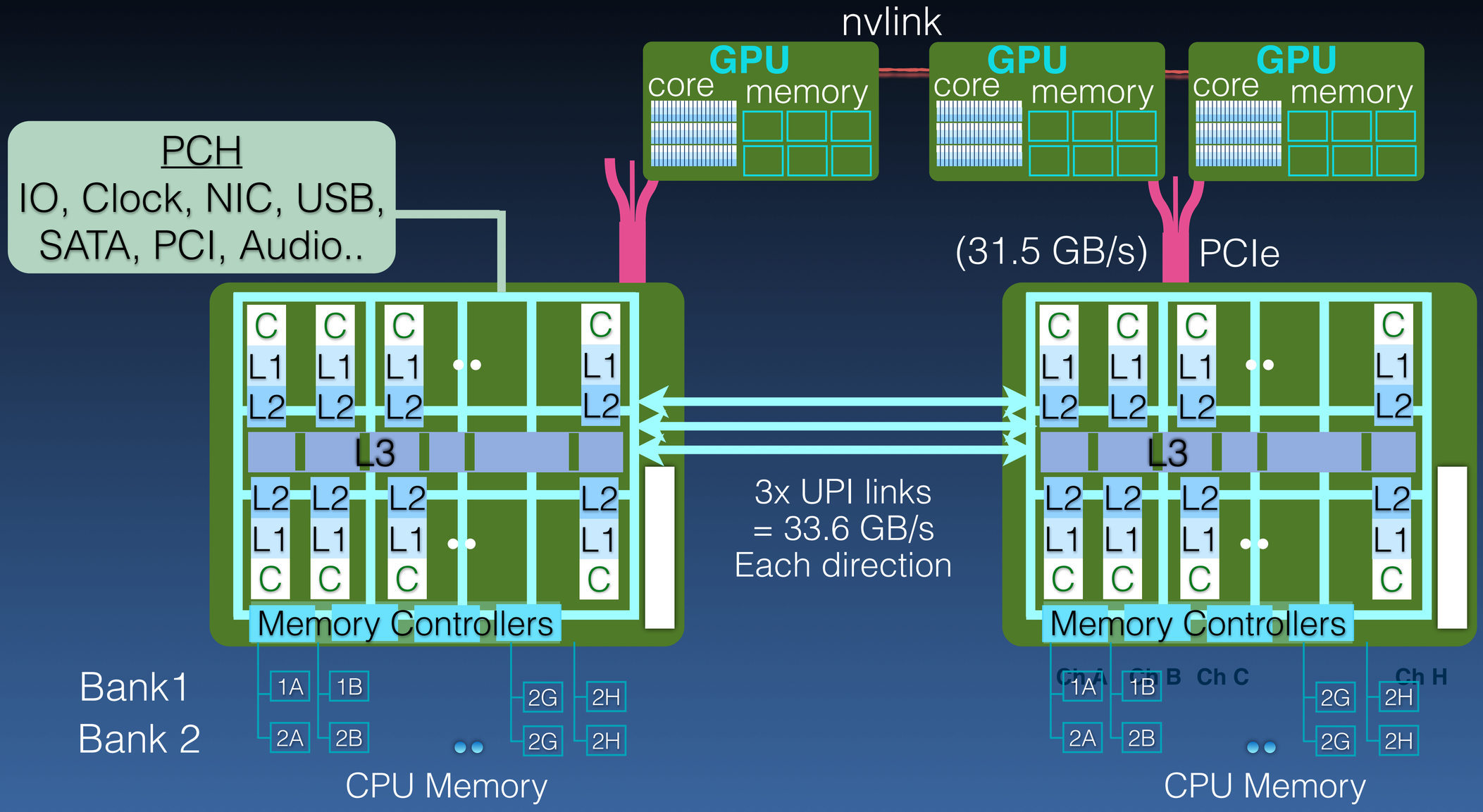
CPU Memory

Ch A Ch B Ch C Ch H

CPU Memory

Subodh Kumar

Modern Multi-Processor



GPU

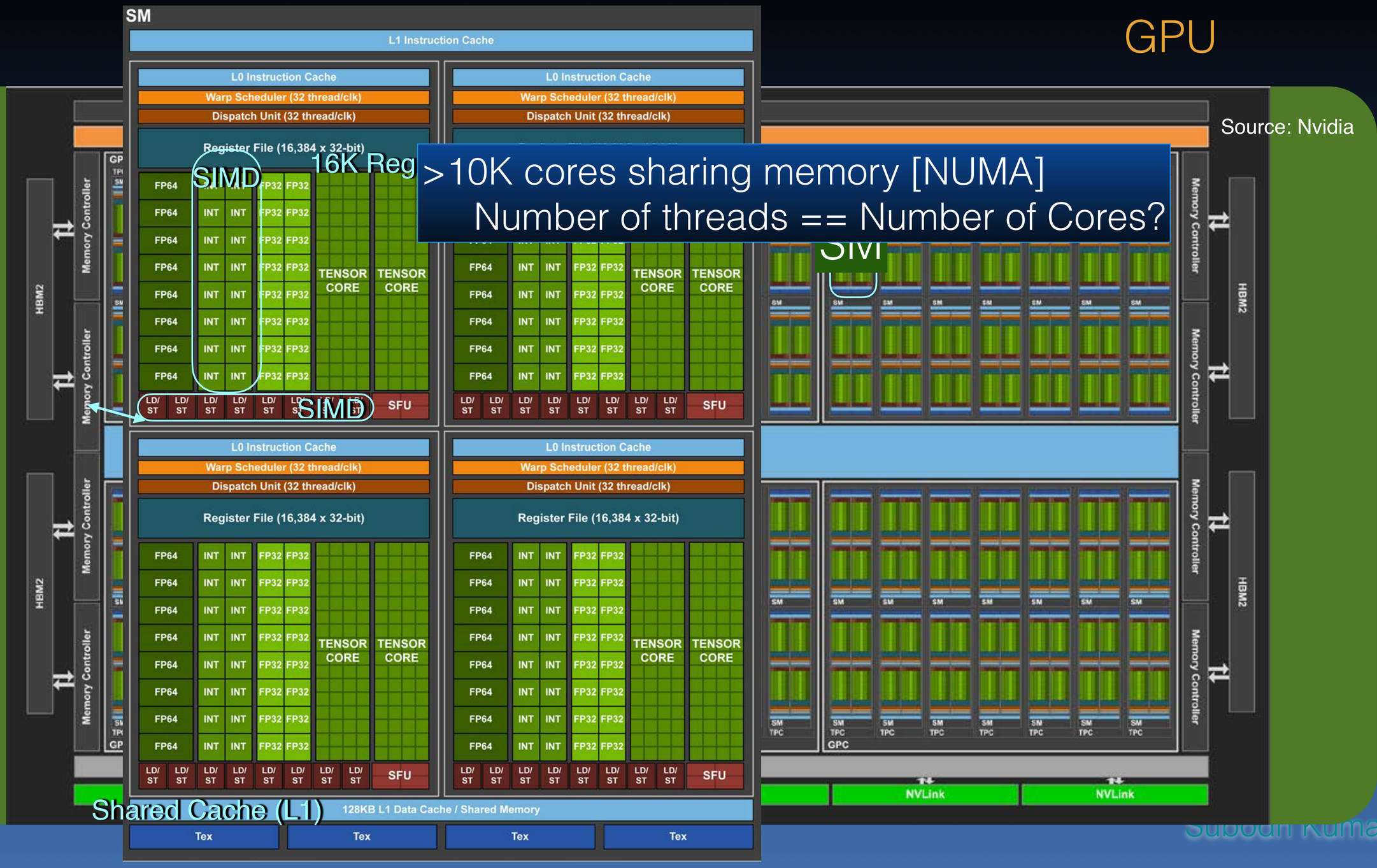
Source: Nvidia

GPU

Source: Nvidia

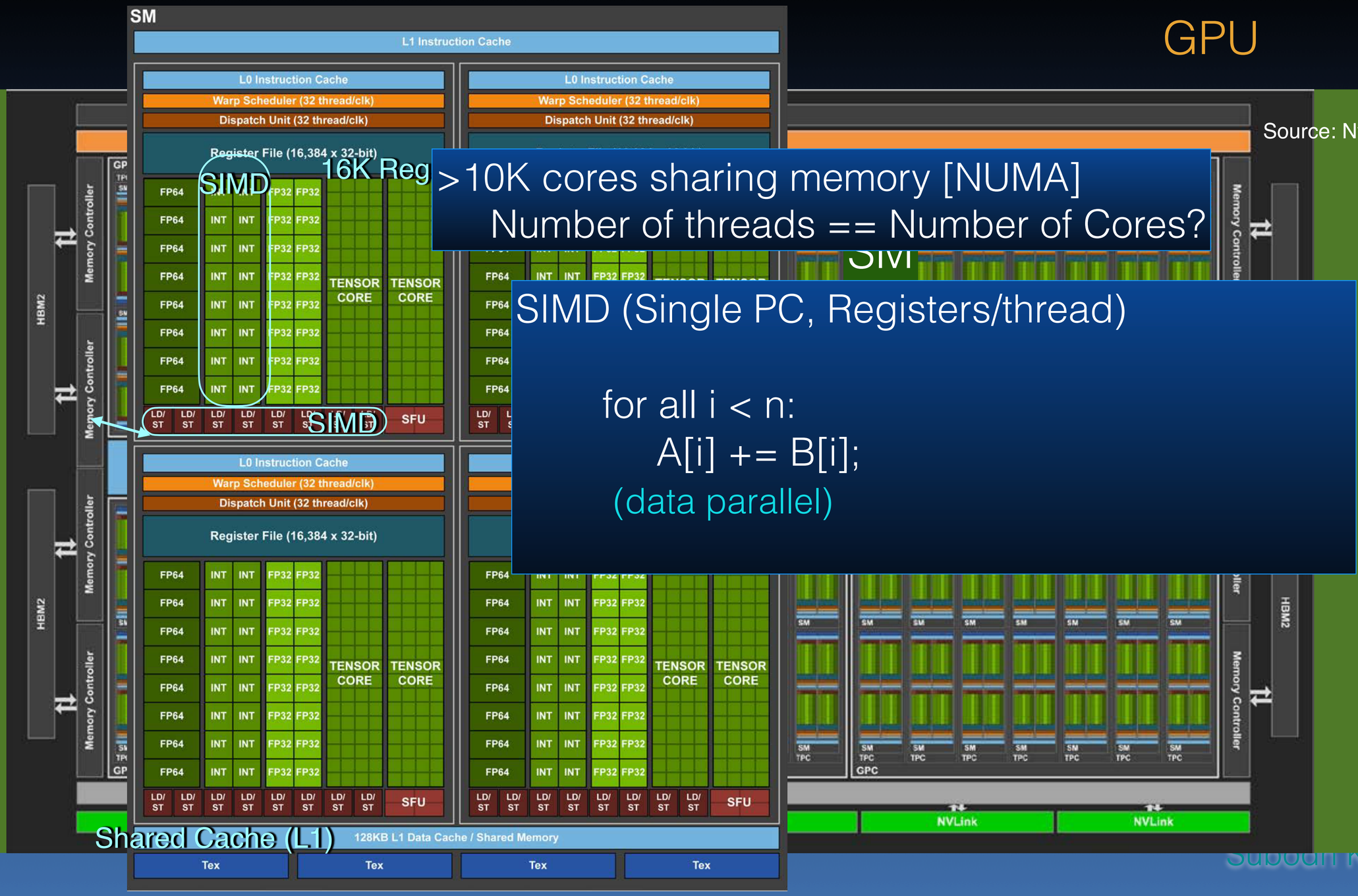
GPU

Source: Nvidia



GPU

Source: Nvidia



>10K cores sharing memory [NUMA]

Number of threads == Number of Cores?

SIMD (Single PC, Registers/thread)

```
for all i < n:
    A[i] += B[i];
```

(data parallel)

GPU

Source: Nvidia

>10K cores sharing memory [NUMA]

Number of threads == Number of Cores?

SIMD (Single PC, Registers/thread)

```
for all i < n:
    A[i] += B[i];
```

(data parallel)

```
for all i < n:
    if(f(i))
        A[i] += B[i];
```

GPU

Source: Nvidia

>10K cores sharing memory [NUMA]

Number of threads == Number of Cores?

SIMD (Single PC, Registers/thread)

```
for all i < n:
    A[i] += B[i];
```

(data parallel)

```
for all i < n:
    if(f(i))   g(..)
    else       h(..)
```

GPU

Source: Nvidia

>10K cores sharing memory [NUMA]

Number of threads == Number of Cores?

SIMD (Single PC, Registers/thread)

```
for all i < n:
    A[i] += B[i];
```

(data parallel)

```
for all i < n:
    pred = (f(i))
    pred  && h(i)
    !pred && g(i)
```

GPU

Source: Nvidia

>10K cores sharing memory [NUMA]

Number of threads == Number of Cores?

SIMD (Single PC, Registers/thread)

```
for all i < n:
    A[i] += B[i];
```

(data parallel)

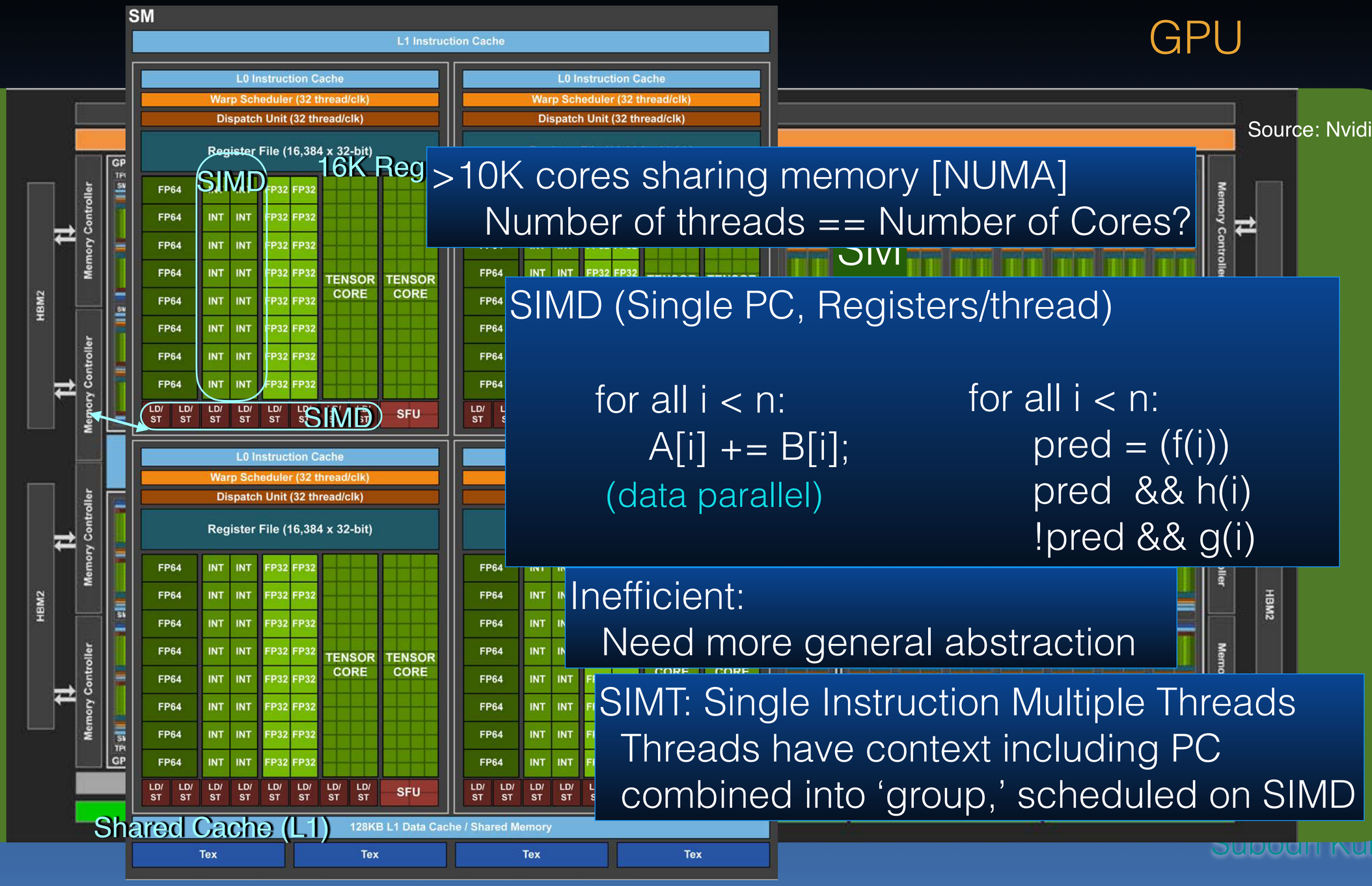
```
for all i < n:
    pred = (f(i))
    pred  && h(i)
    !pred && g(i)
```

Inefficient:

Need more general abstraction

GPU

Source: Nvidia



>10K cores sharing memory [NUMA]

Number of threads == Number of Cores?

SIMD (Single PC, Registers/thread)

```
for all i < n:
    A[i] += B[i];
```

(data parallel)

```
for all i < n:
    pred = (f(i))
    pred  && h(i)
    !pred && g(i)
```

Inefficient:

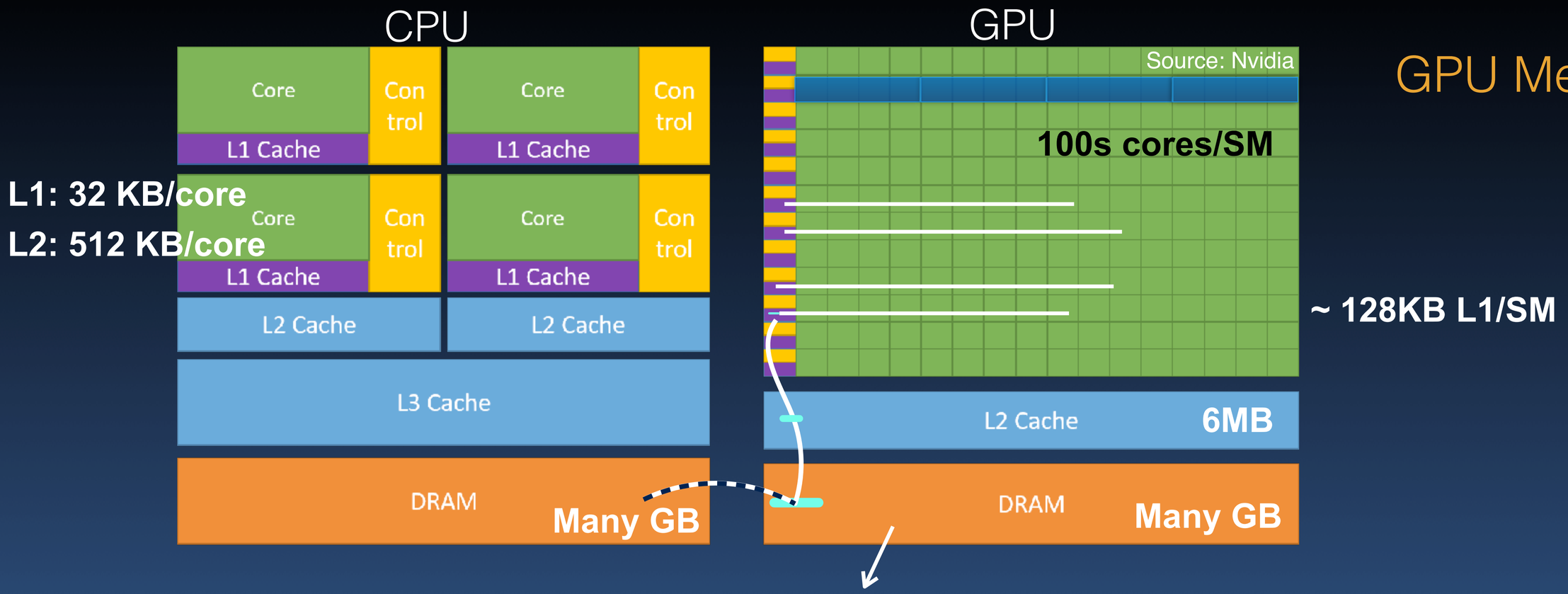
Need more general abstraction

SIMT: Single Instruction Multiple Threads

Threads have context including PC

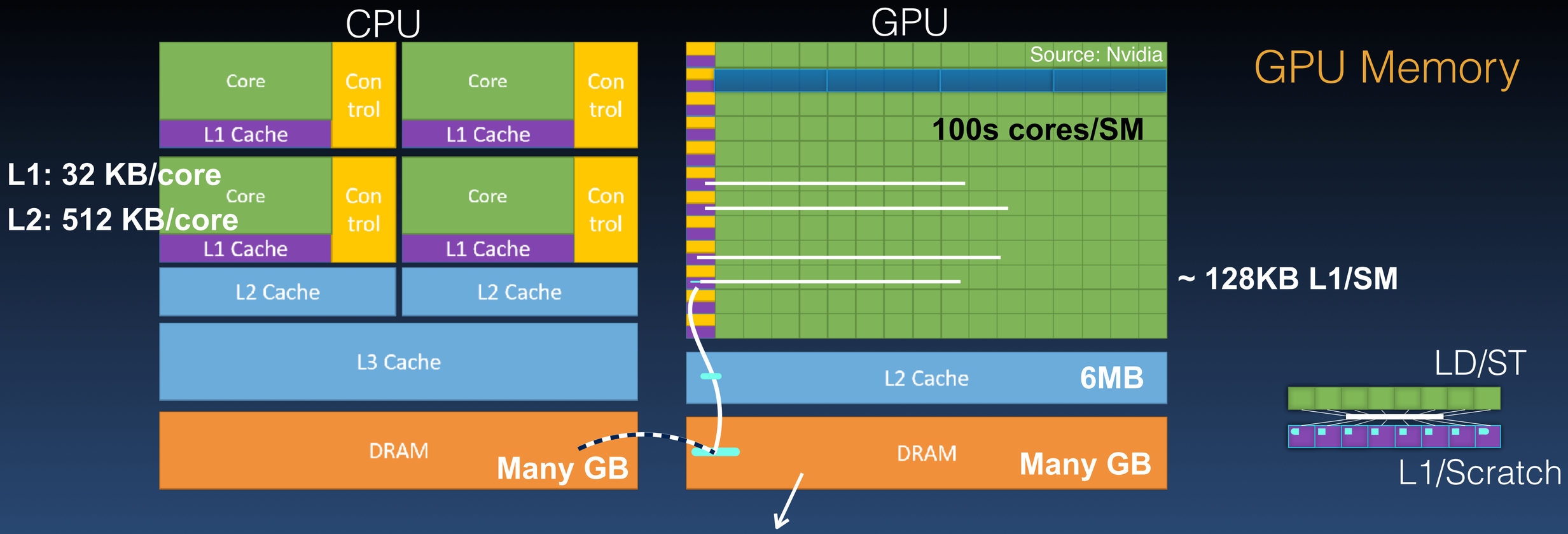
combined into ‘group,’ scheduled on SIMD

GPU Memory



- Many threads, incl. many memory operations
  - ➡ SIMT allows program-visible parallel memory operations
  - ➡ Memory latency can be high (use more local mem, hide latency)
    (Less cache per core)

GPU Memory



- Many threads, incl. many memory operations
  - ➡ SIMT allows program-visible parallel memory operations
  - ➡ Memory latency can be high (use more local mem, hide latency)
    (Less cache per core)

Offload Programming Model

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

OpenMP

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

(RPC)

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

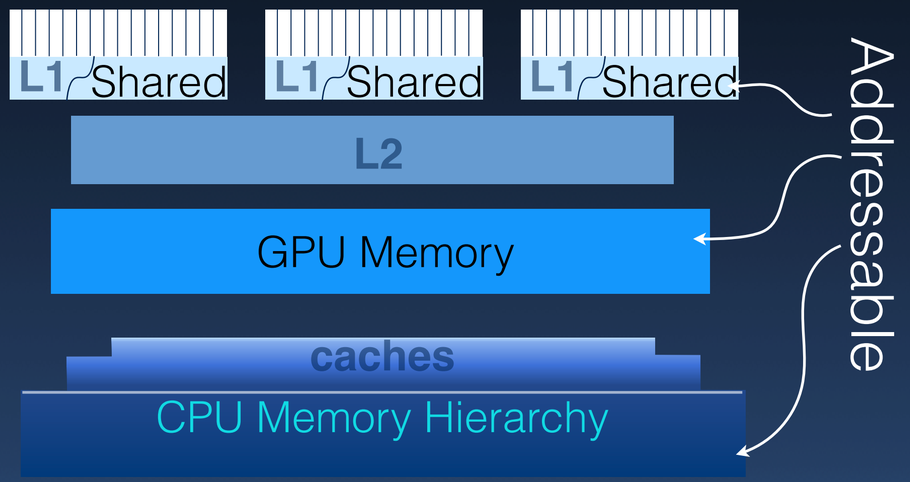
(function per thread)

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;
}
```

CUDA Programming Model

- Co-processor (with many cores)
  - ➡ CPU code offloads multi-threaded tasks
    - ▸ Message passing and shared memory models
- GPU Threads are organized hierarchically
  - ➡ Grids, Blocks, Warps
  - ➡ Core programming style is data-parallel (including Read/Write)
- Multiple addressable memory
  - ➡ Weak consistency, User controlled cache

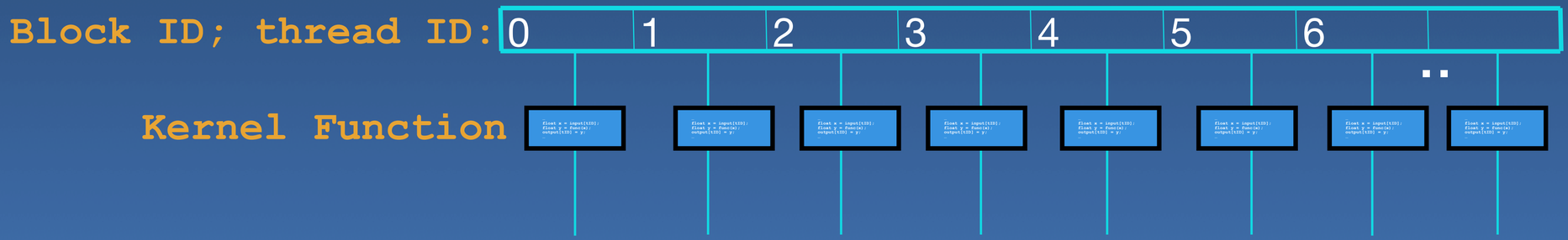


Device & Threads

- Each GPU is also called a device
  - ➡ GPU memory is device memory
  - ➡ CPU is the host
- Device executes GPU code (kernel)
  - ➡ Executed by multiple blocks of threads in parallel
- GPU threads are lightweight
  - ▸ Trivial creation and scheduling overhead, retain context
  - ▸ Many thousand threads

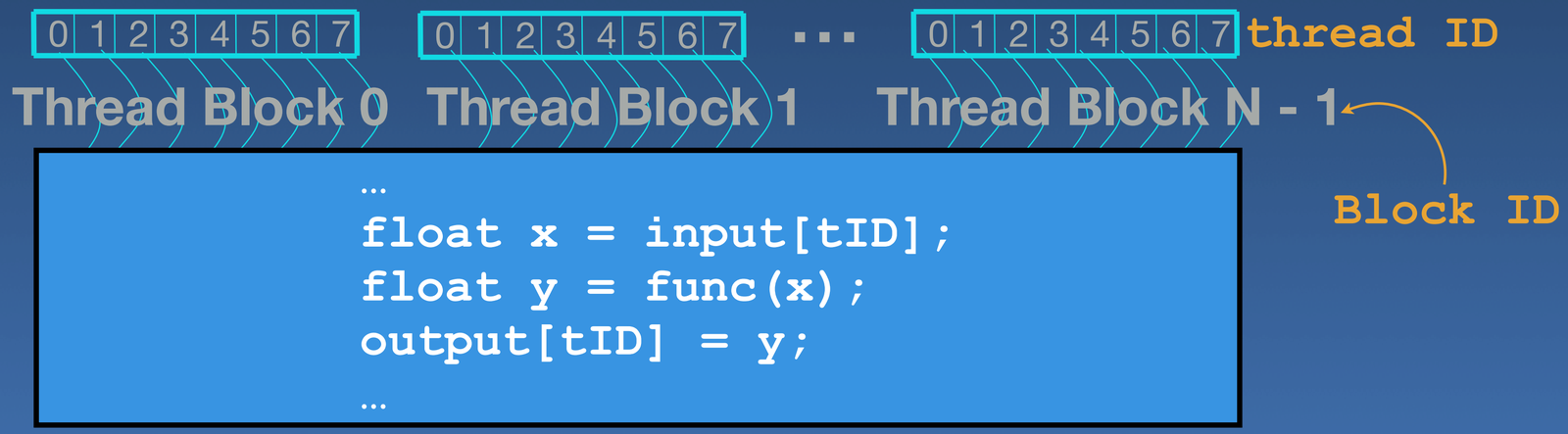
SIMT Threads

- CPU thread ‘launches’ a kernel executed by a grid of threads
  - ➡ Non-blocking: grid executes asynchronously with CPU thread
  - ➡ Multiple streams can be created
  - ➡ All blocks of the grid may execute in parallel; streams can be concurrent
    - ▸ Once a block begins, context remains live until its completion
    - ▸ A block is divided into warps of 32 threads, each warp is SIMT
      - Like SIMD; each thread has its own context; threads within a warp may diverge



Thread Blocks

- All threads share global memory
- Threads within a block also share shared memory
  - ➡ Additional Intra-warp register-based computation
- Barrier and fine-grained synchronization within warps and blocks
  - ➡ Memory Fences and CAS available across blocks

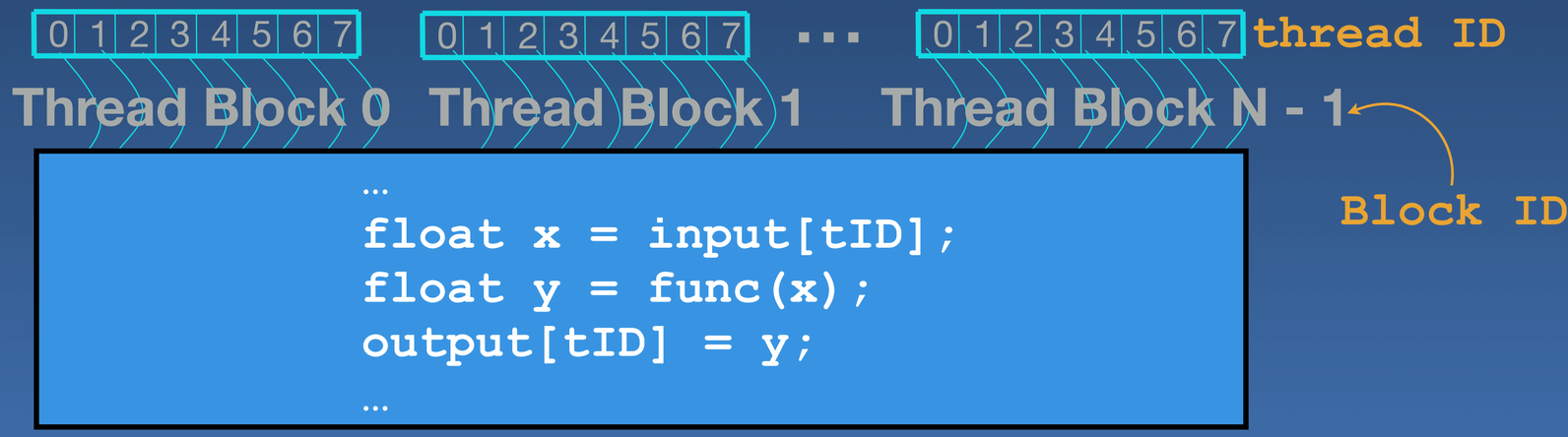


Thread Blocks

- All threads share global memory
- Threads within a block also share shared memory
  - ➡ Additional Intra-warp register-based computation
- Barrier and fine-grained synchronization within warps and blocks
  - ➡ Memory Fences and CAS available across blocks

Group must be resident

Can also group thread:
see Cooperative Groups

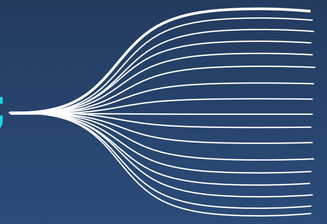


CUDA + C

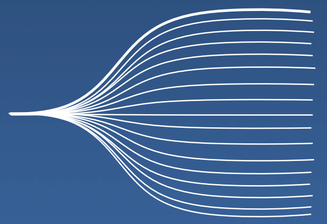
Source: Nvidia

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

host: **Serial Code()**
prefetch data transfer
GPU: **KernelA<<< nBlk, nTid >>>(args);**
host: optionally wait
postfetch data transfer
host: **Serial Code Another()**
prefetch data transfer
GPU: **KernelB<<< nBlk, nTid >>>(args);**
host: optionally wait
postfetch data transfer
host: **Serial Code More()**



implicit barrier at the end



implicit barrier at the end

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

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

```
__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);
```

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

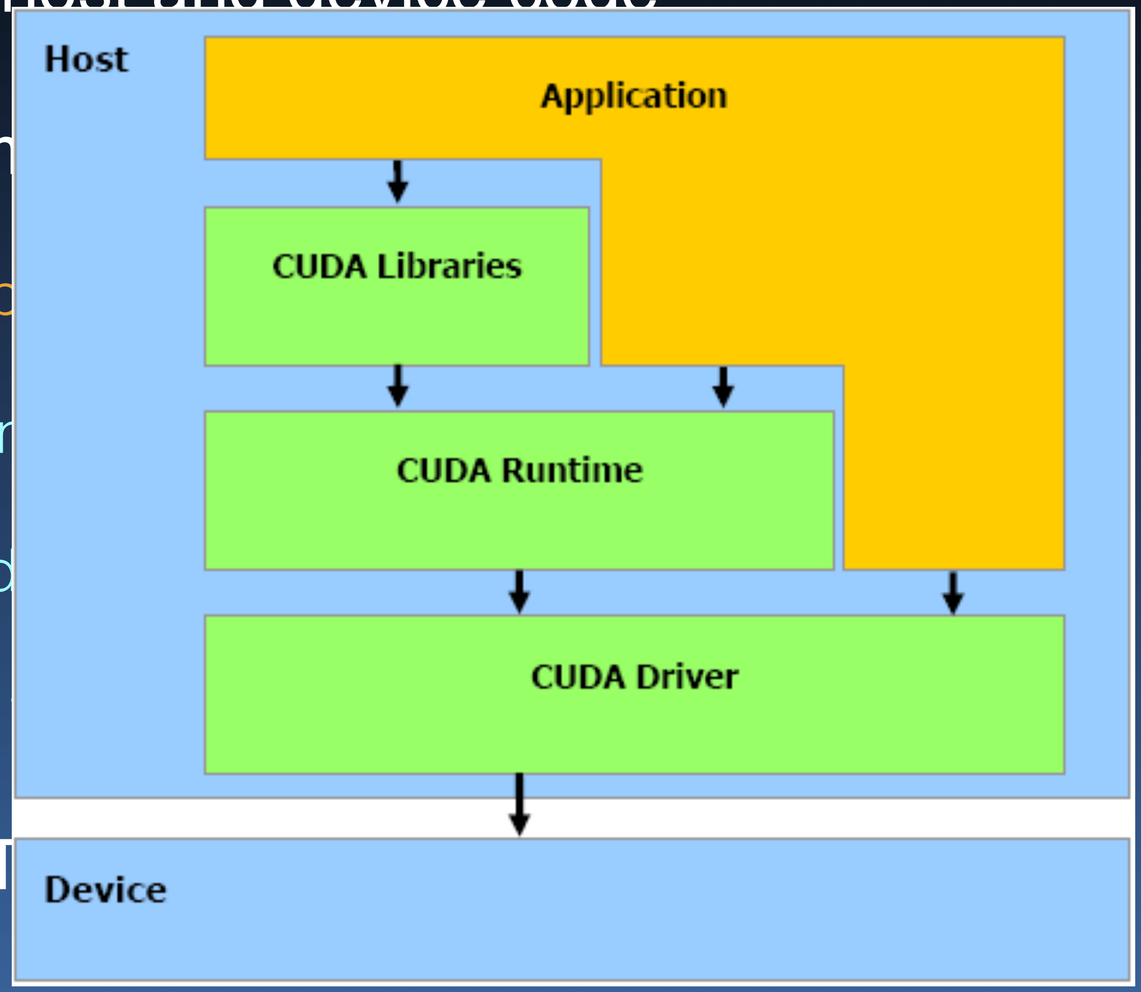
```
__device__ float filter[N];

__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

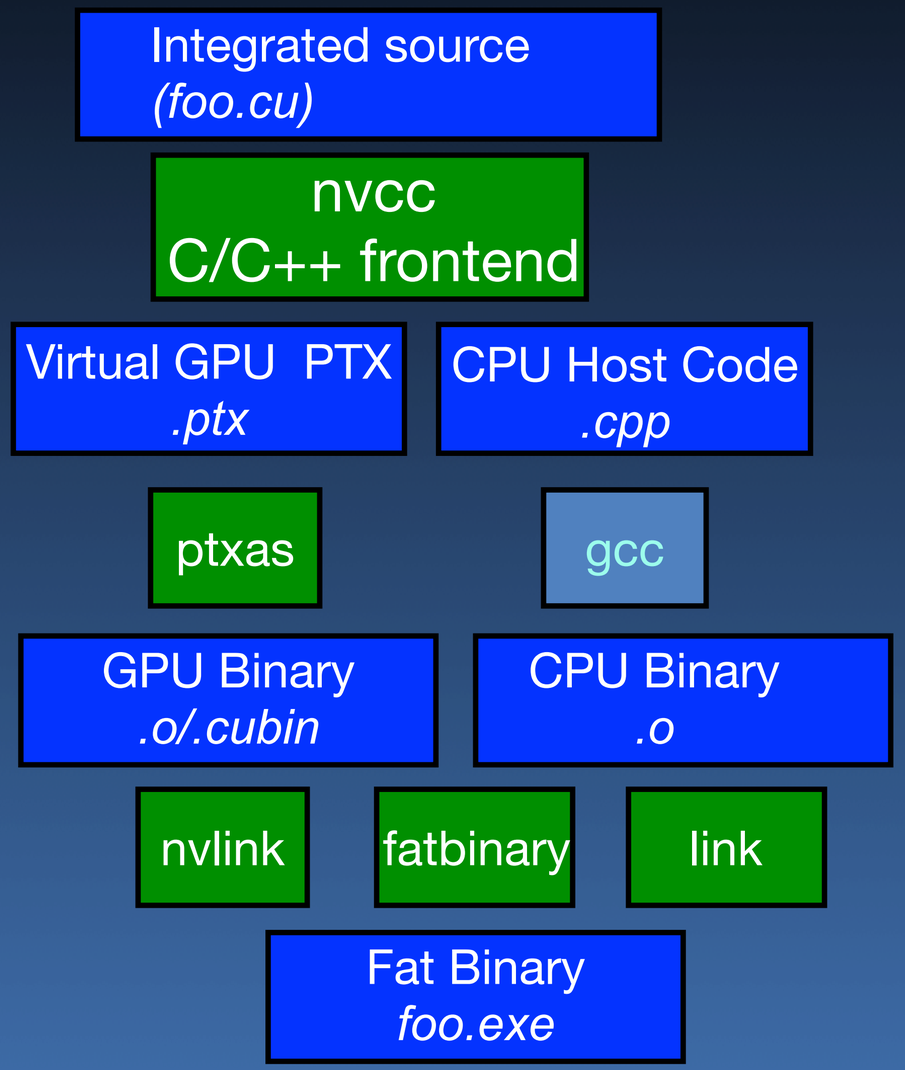
- Source files (.cu) have a mix of host and device code
- nvcc separates device code from host code
  - ➡ compiles device code into PTX/cubin
  - ➡ host code is output as C source (and C compiler invoked)
  - ➡ PTX/cubin incorporated in host code as a global initialized data array
  - ➡ includes cudart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

CUDA Tool-chain

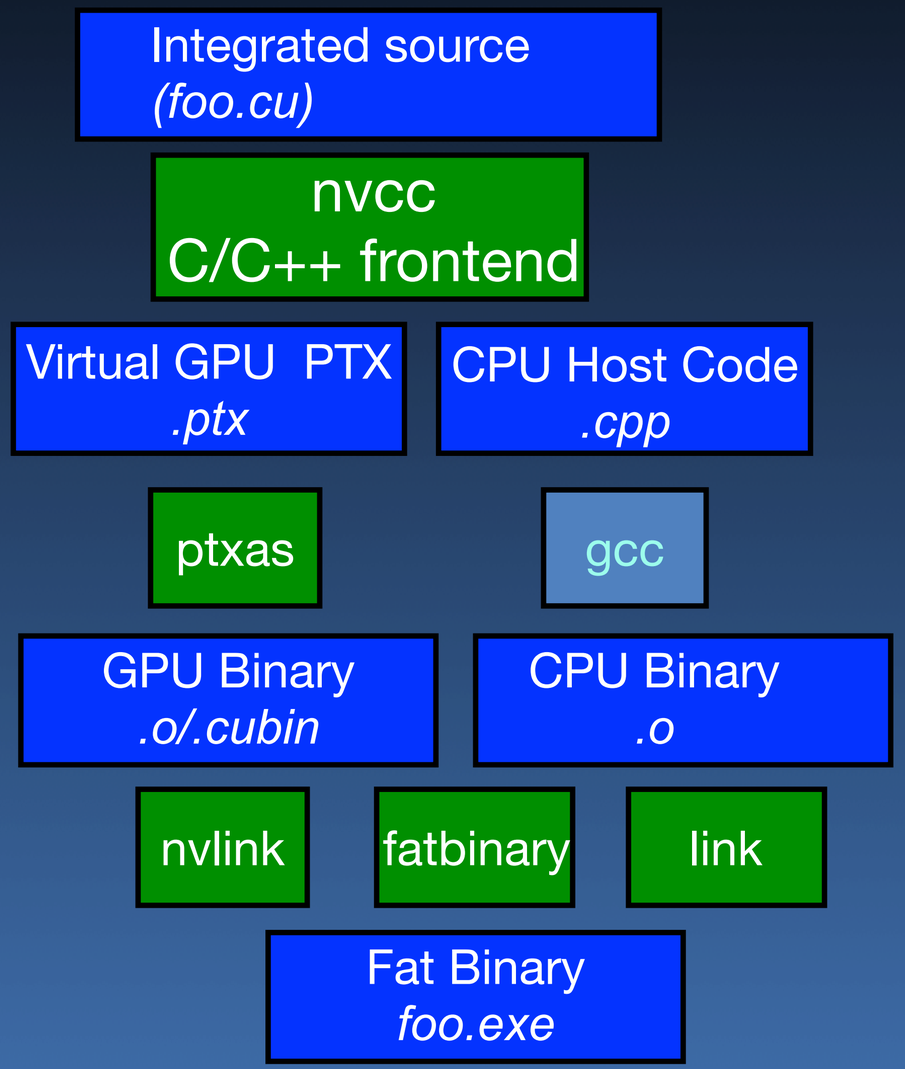
- Source files (.cu) have a mix of host and device code
- nvcc separates device code fro
  - ➡ compiles device code into PTX/cub
  - ➡ host code is output as C source (ar
  - ➡ PTX/cubin incorporated in host coc
  - ➡ includes cudart (CUDA C runtime)
- Possible to load and execute PT



Cuda Dev Tools



Cuda Dev Tools



cuobjdump | nvdisasm

cuda-gdb | cuda-memcheck

Parallel Nsight | CUDA Visual Profiler