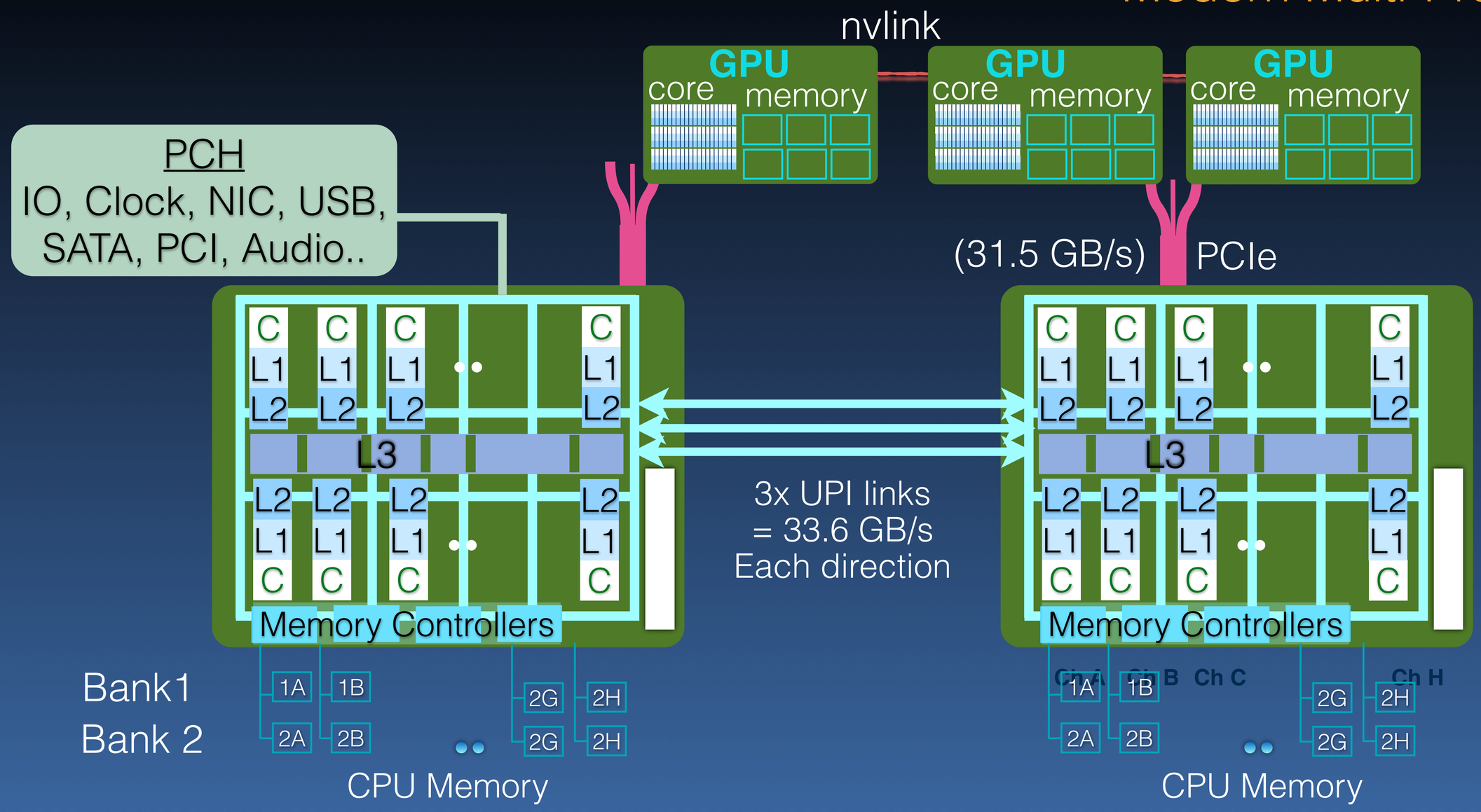
COL380

Introduction to
Parallel & Distributed Programming

Manycore GPU

Modern Multi-Processor



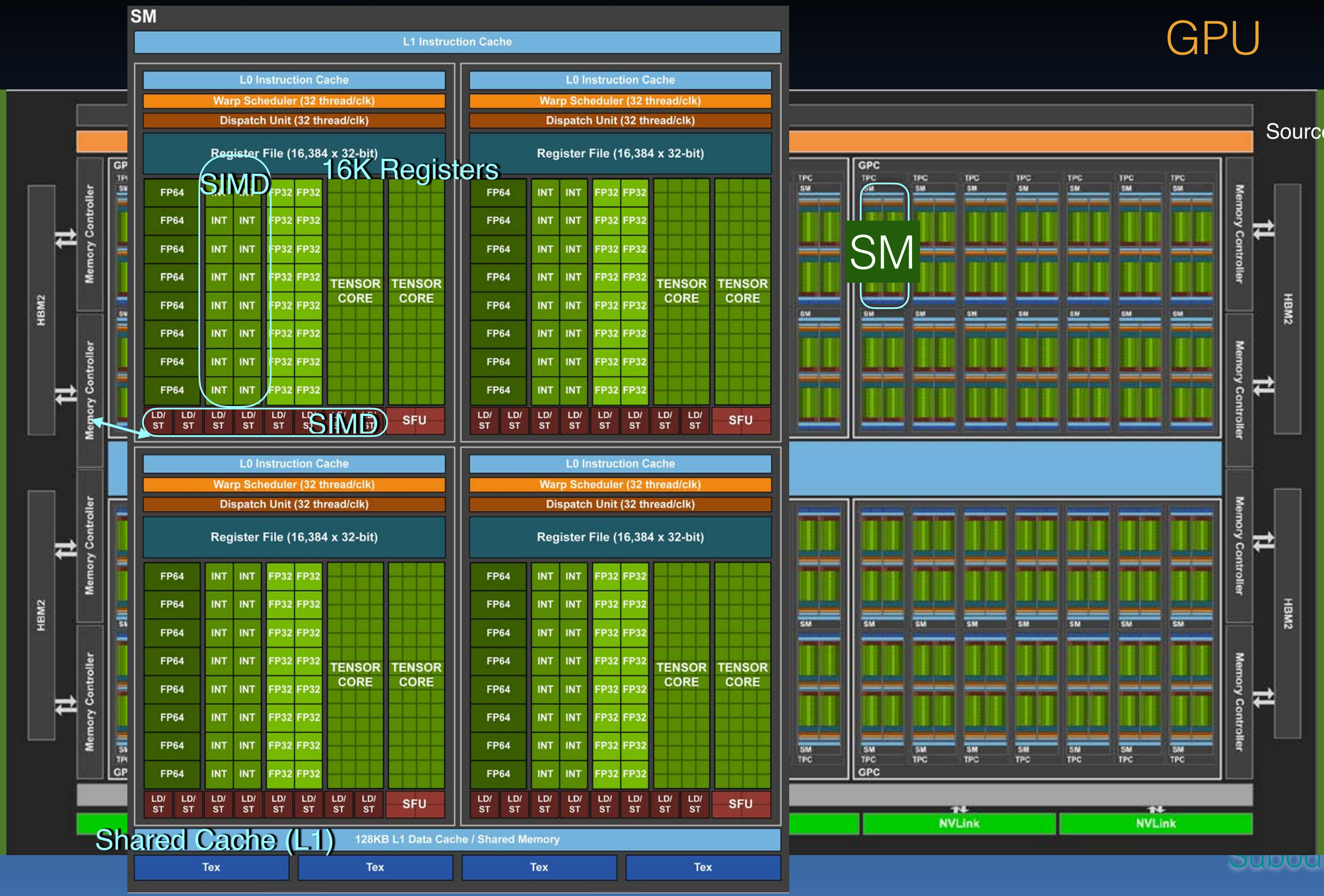
Subodh Kumar

GPU

Source: Nvidia

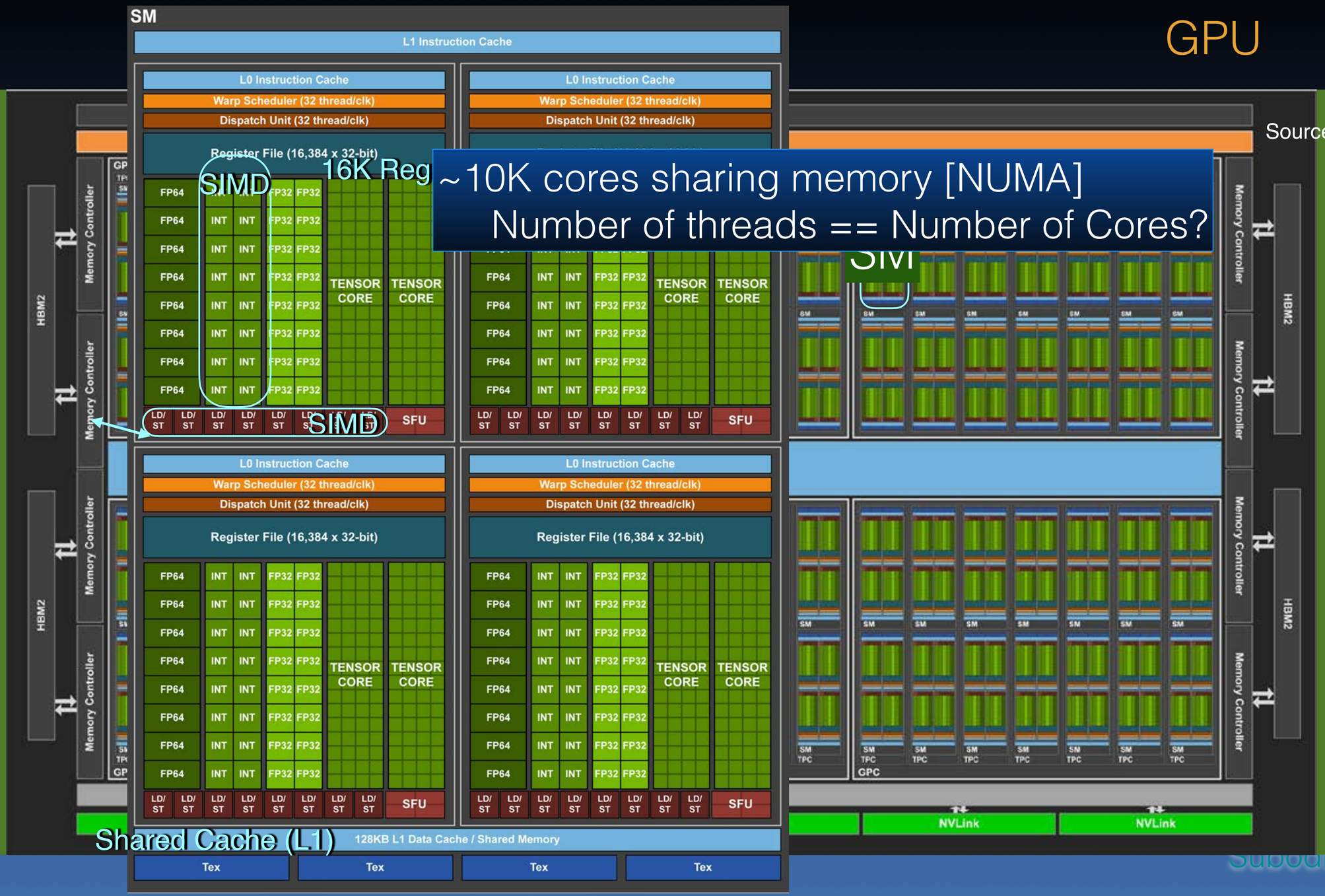
GPU

Source: Nvidia



GPU

Source: Nvidia



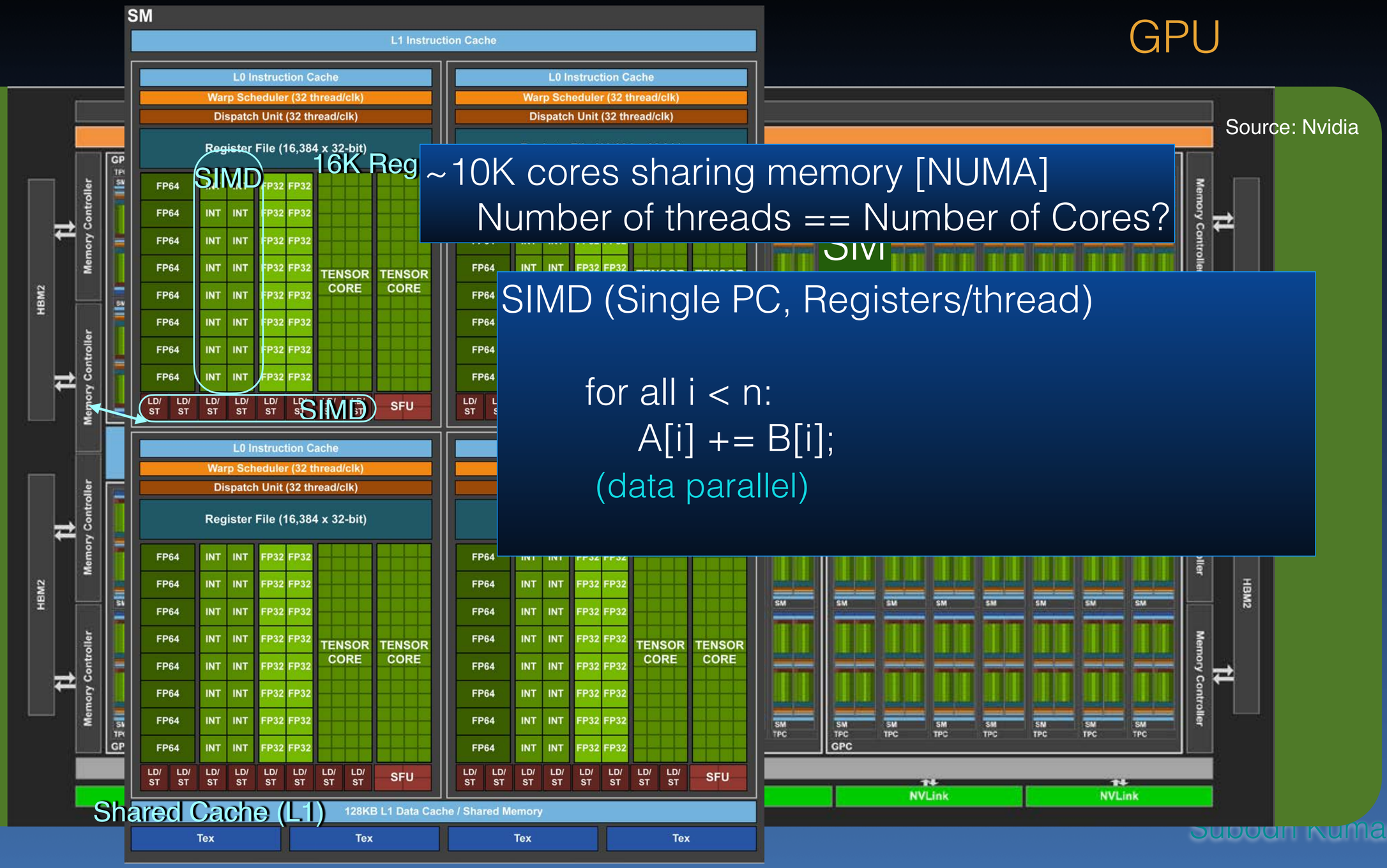
~10K cores sharing memory [NUMA]

Number of threads == Number of Cores?

Subodh Kumar

GPU

Source: Nvidia



~10K cores sharing memory [NUMA]

Number of threads == Number of Cores?

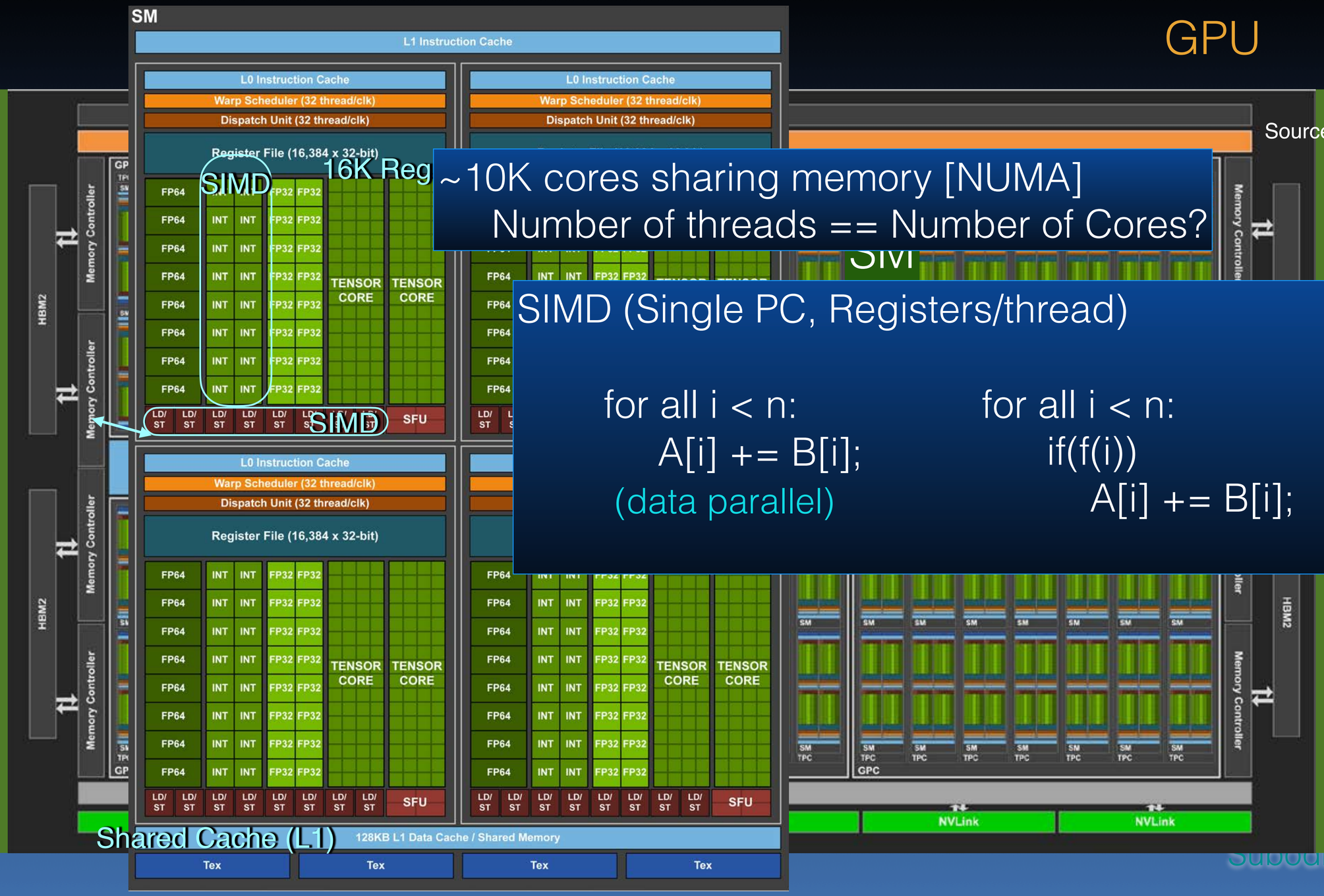
SIMD (Single PC, Registers/thread)

```
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)

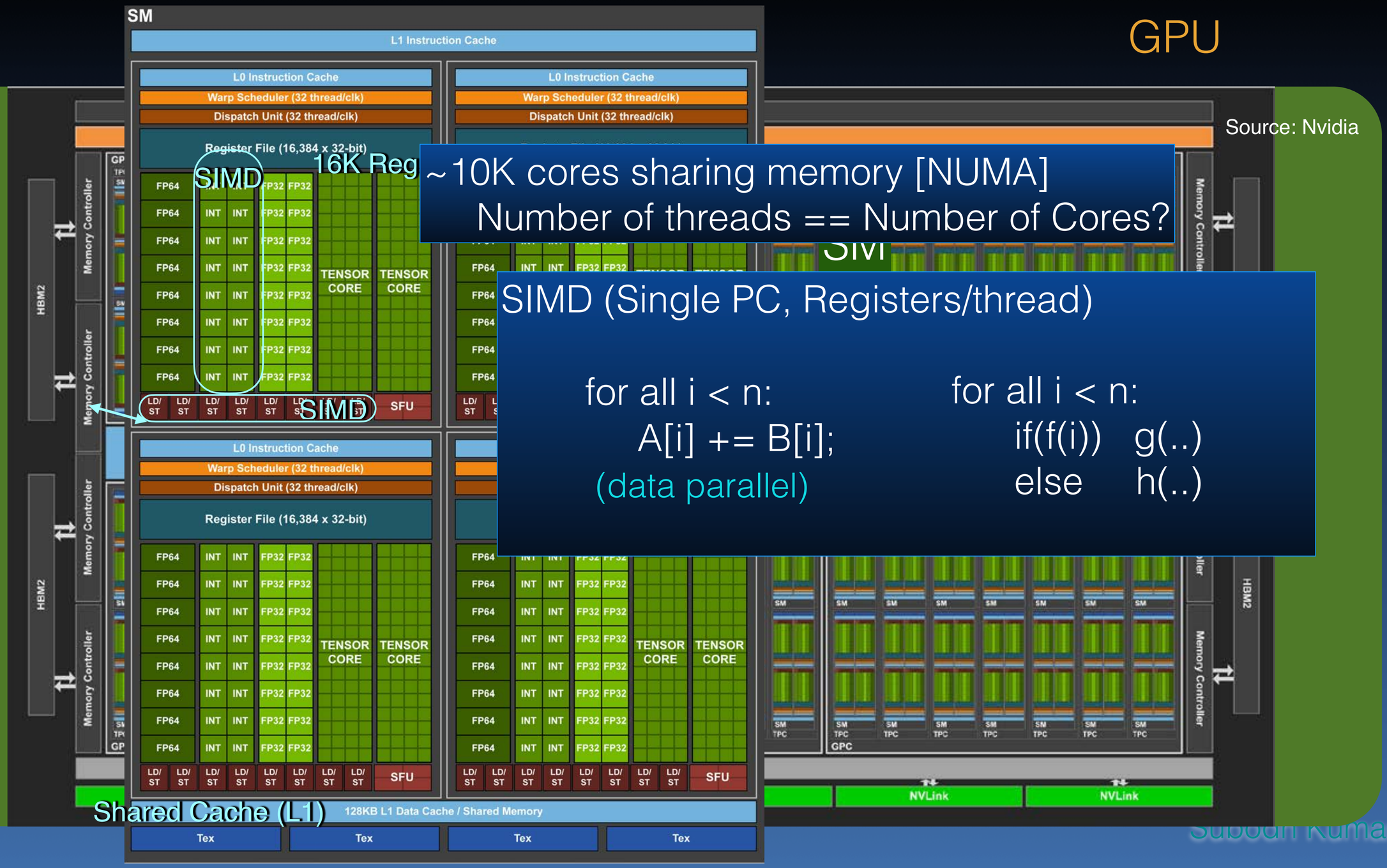
```
for all i < n:
    A[i] += B[i];
```

(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)

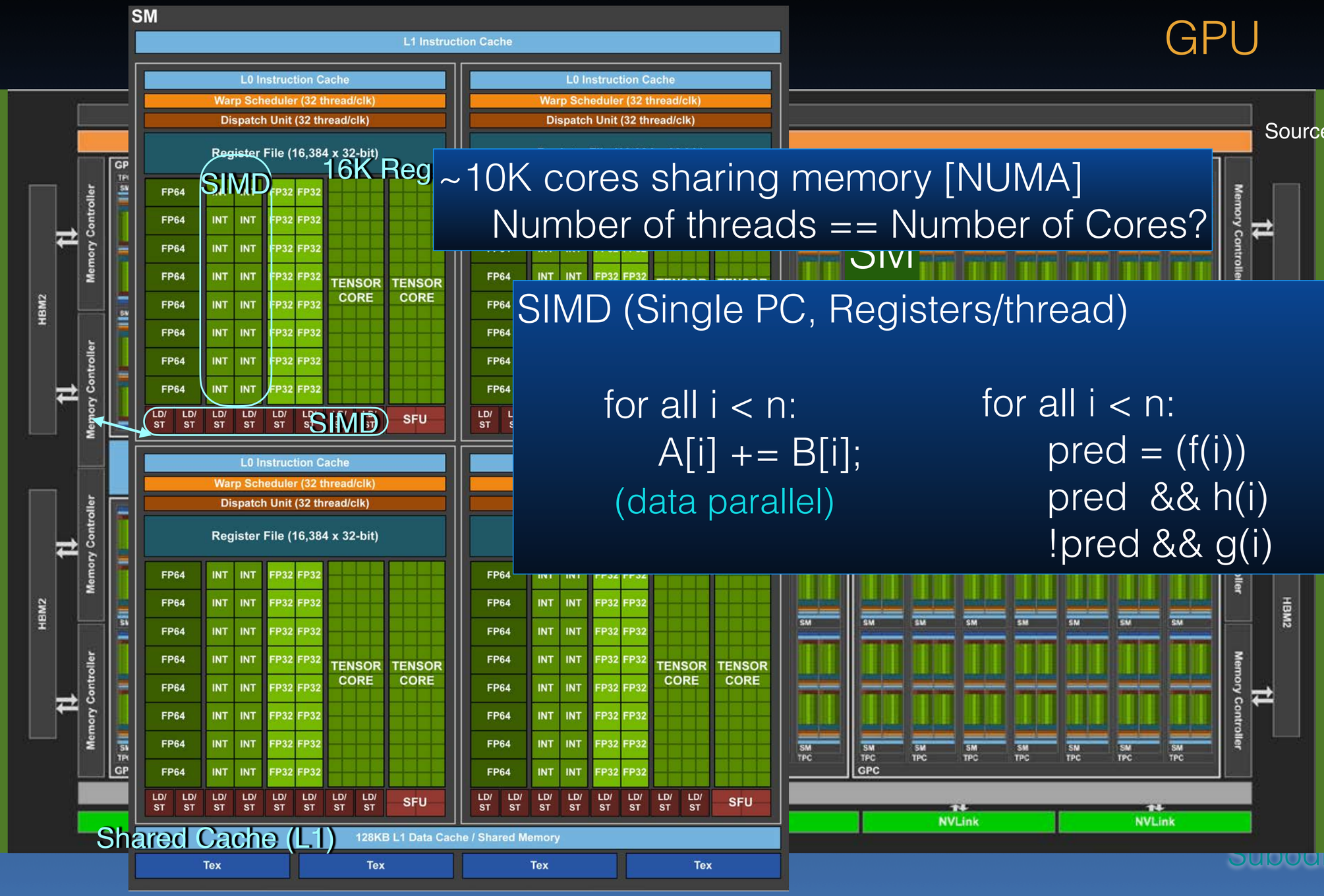
```
for all i < n:
    A[i] += B[i];
```

(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)

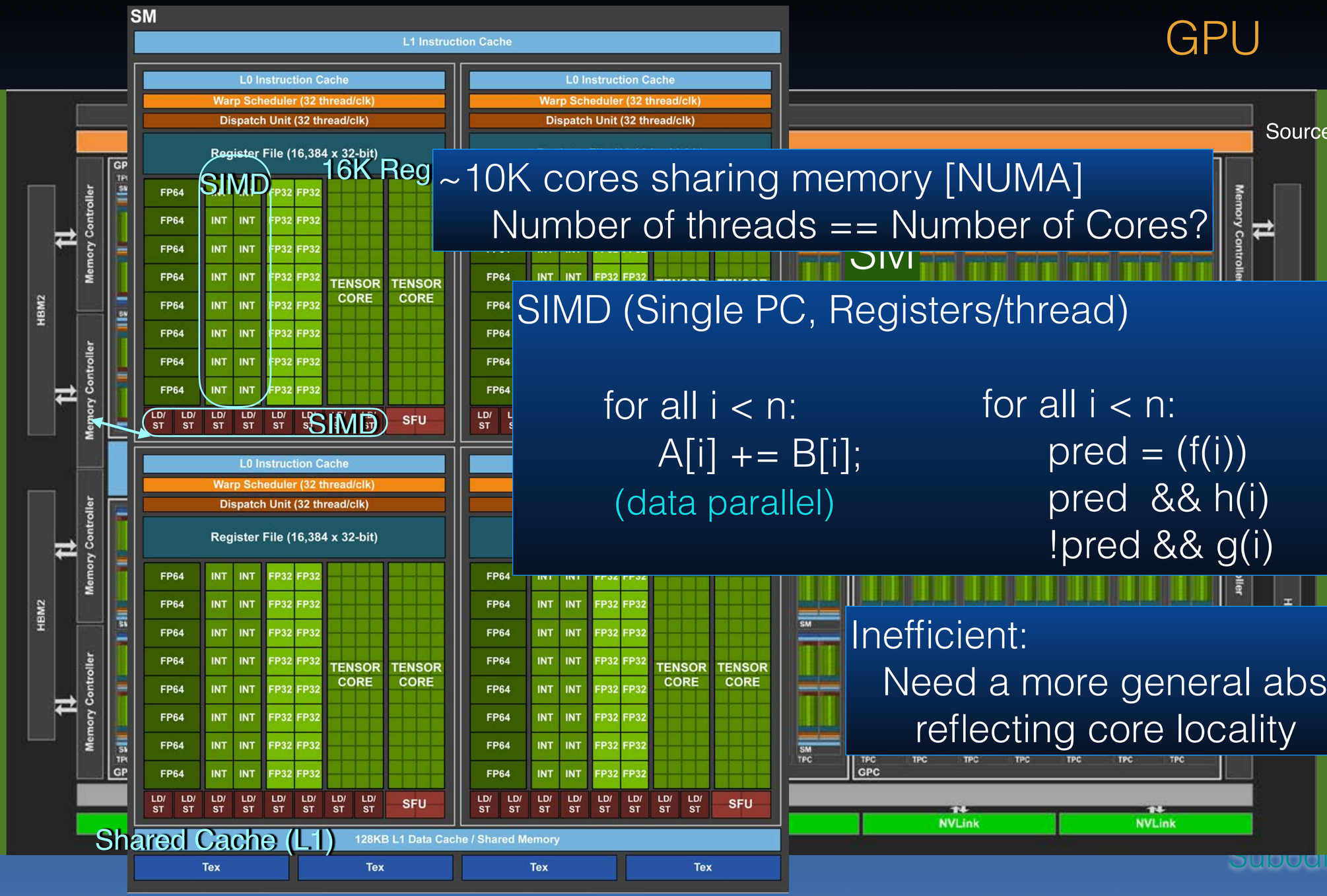
```
for all i < n:
    A[i] += B[i];
```

(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)

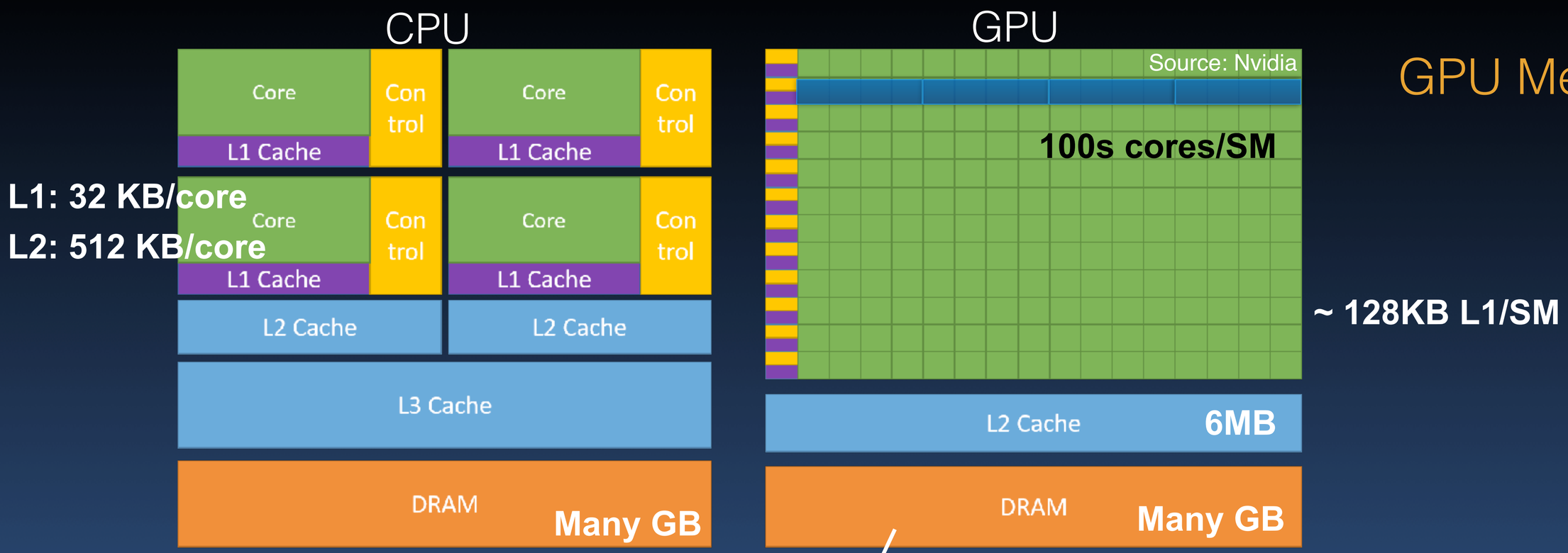
```
for all i < n:
    pred = (f(i))
    pred  && h(i)
    !pred && g(i)
```

Inefficient:

Need a more general abstraction reflecting core locality

Subodh Kumar

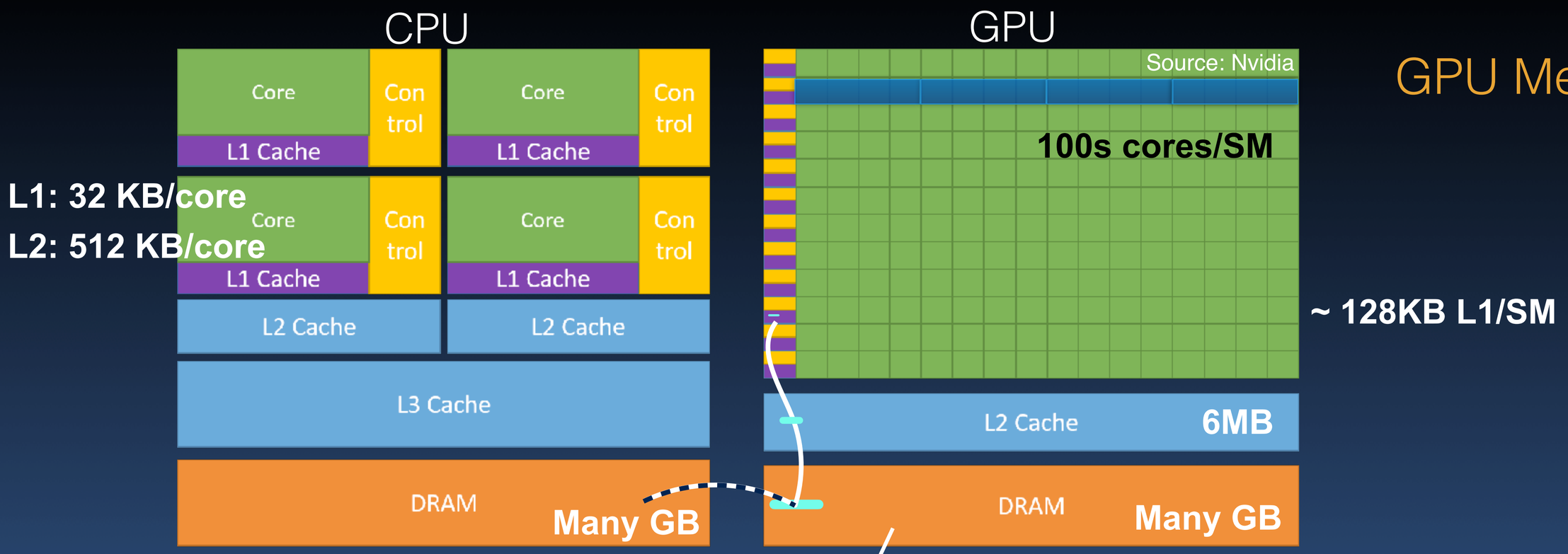
GPU Memory



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

Subodh Kumar

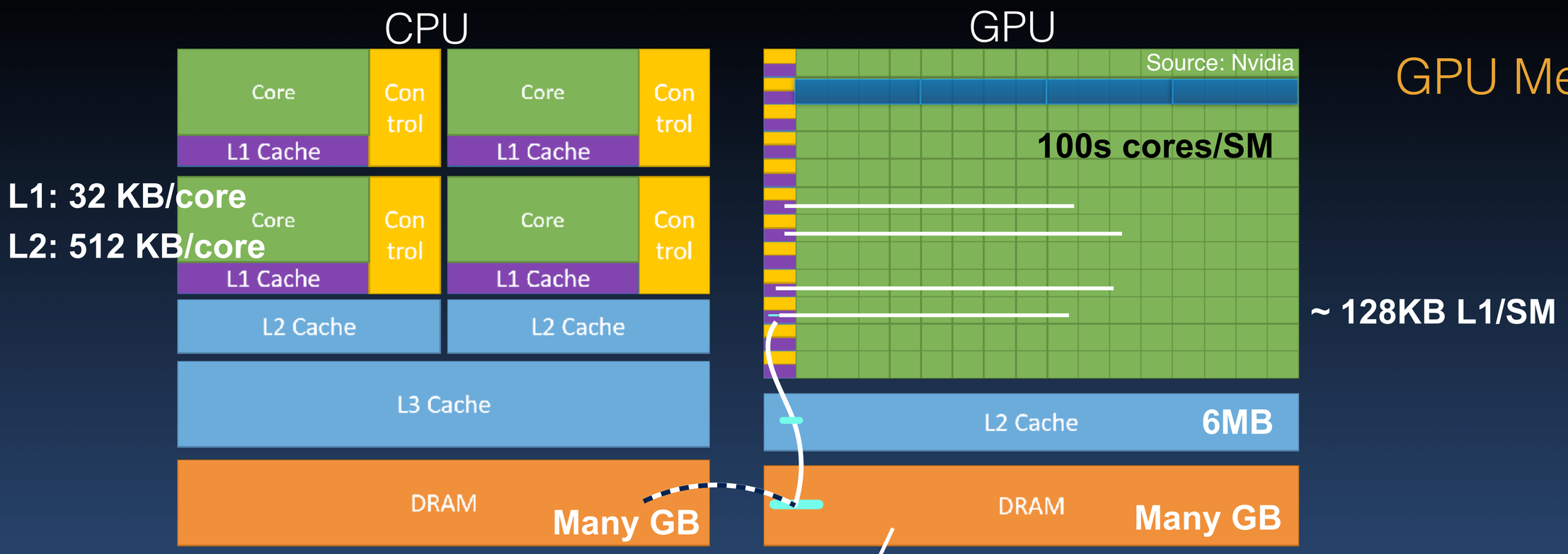
GPU Memory



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

Subodh Kumar

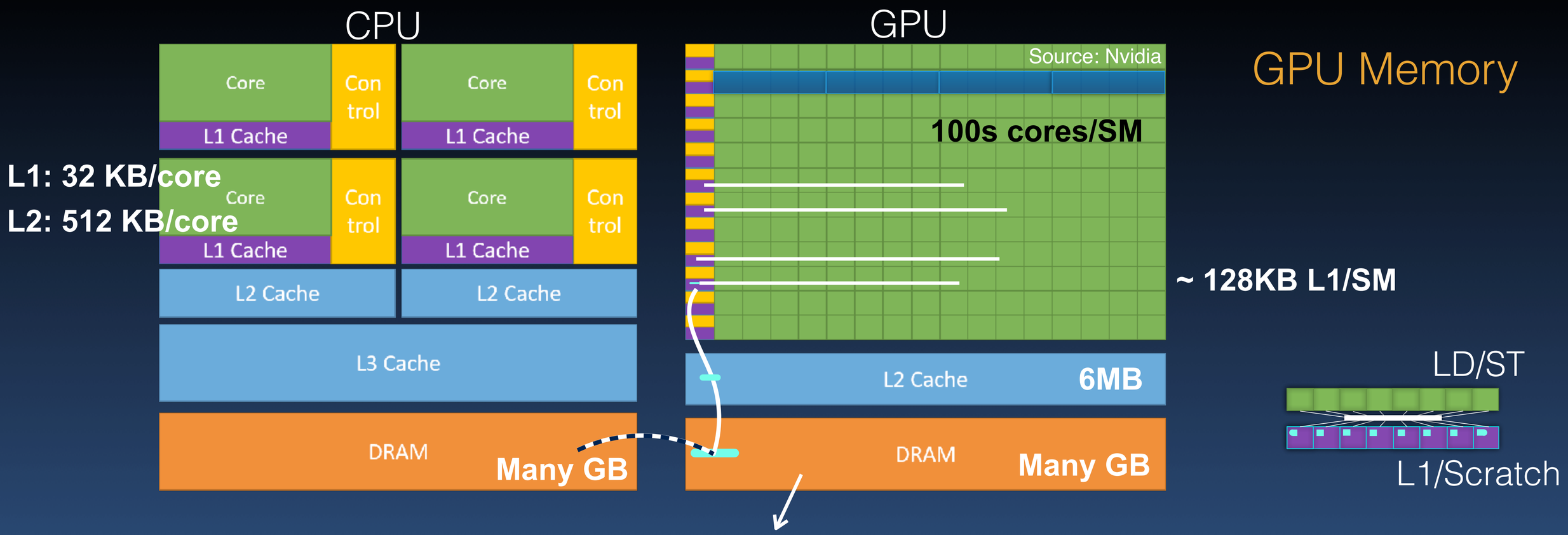
GPU Memory



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

Subodh Kumar

GPU Memory



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

Subodh Kumar

Offload Programming Model

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

OpenMP

Offload Programming Model

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

OpenMP

Offload Programming Model

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

OpenMP

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

Offload Programming Model

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

OpenMP

```
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;
}
```

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;
}
```

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;
}
```

Matrix Multiplication

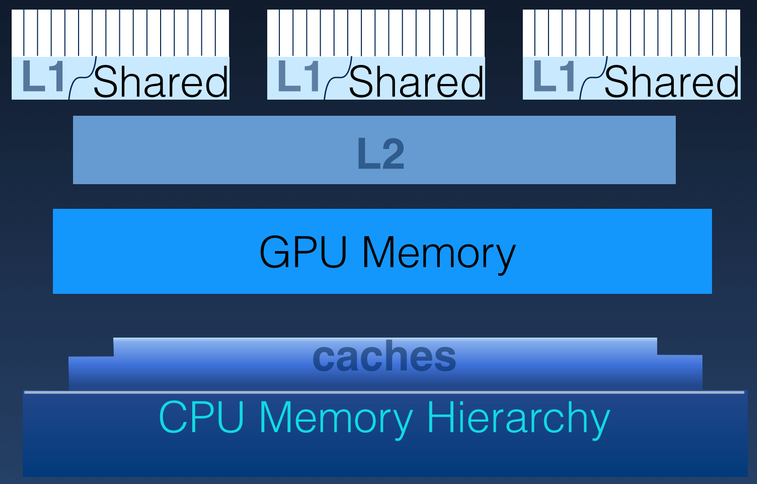
```
// Matrix multiplication kernel – per thread code
__global__ void
MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // Pvalue stores the matrix element computed by the thread
    float Pvalue = 0;

    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

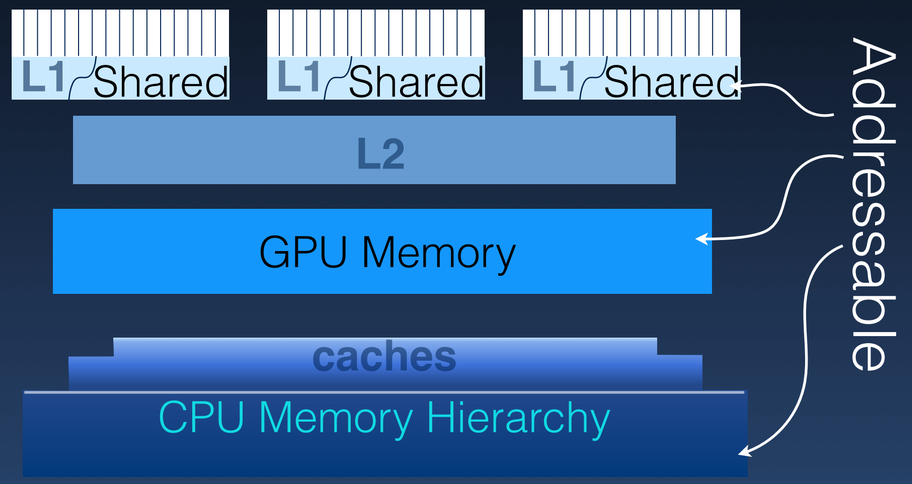
CUDA Programming Model

- 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



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

Device & Threads

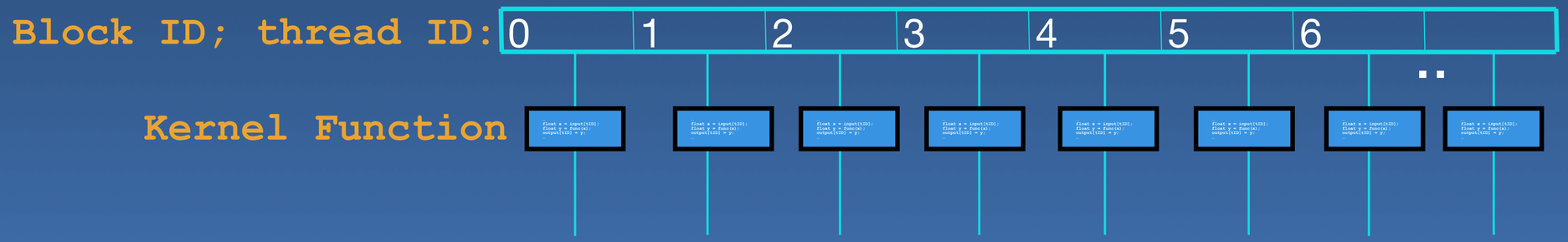
- Each GPU is also called a device
  - ➡ GPU memory is device memory
  - ➡ CPU is the host
- Device executes GPU code (kernel)
  - ➡ Executed by multiple blocks of threads in parallel
- GPU threads are lightweight
  - Trivial creation and scheduling overhead, retain context
  - Many thousand threads

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 <u>may</u> 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

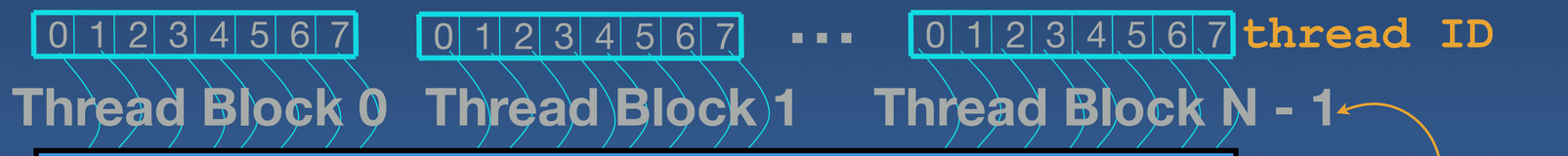
SIMT Threads

- 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 <u>may</u> 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




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

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


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

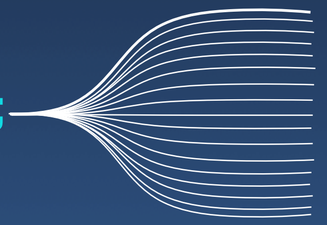
Source: Nvidia

CUDA + C

- 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()**

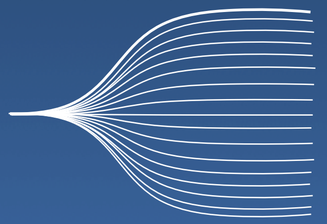
GPU: **KernelA<<< nBlk, nTid >>>(args);**



implicit barrier at the end

host: **Serial Code Another()**

GPU: **KernelB<<< nBlk, nTid >>>(args);**



implicit barrier at the end

host: **Serial Code More()**

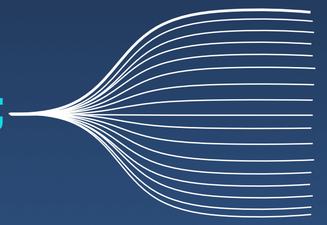
CUDA + C

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()**

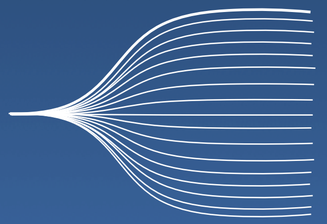
GPU: **KernelA<<< nBlk, nTid >>>(args);**
host: optionally wait



implicit barrier at the end

host: **Serial Code Another()**

GPU: **KernelB<<< nBlk, nTid >>>(args);**
host: optionally wait



implicit barrier at the end

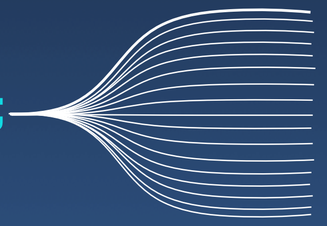
host: **Serial Code More()**

CUDA + C

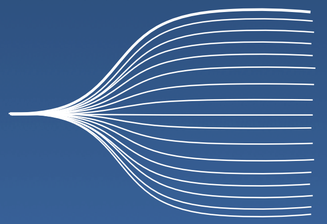
Source: Nvidia

- 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

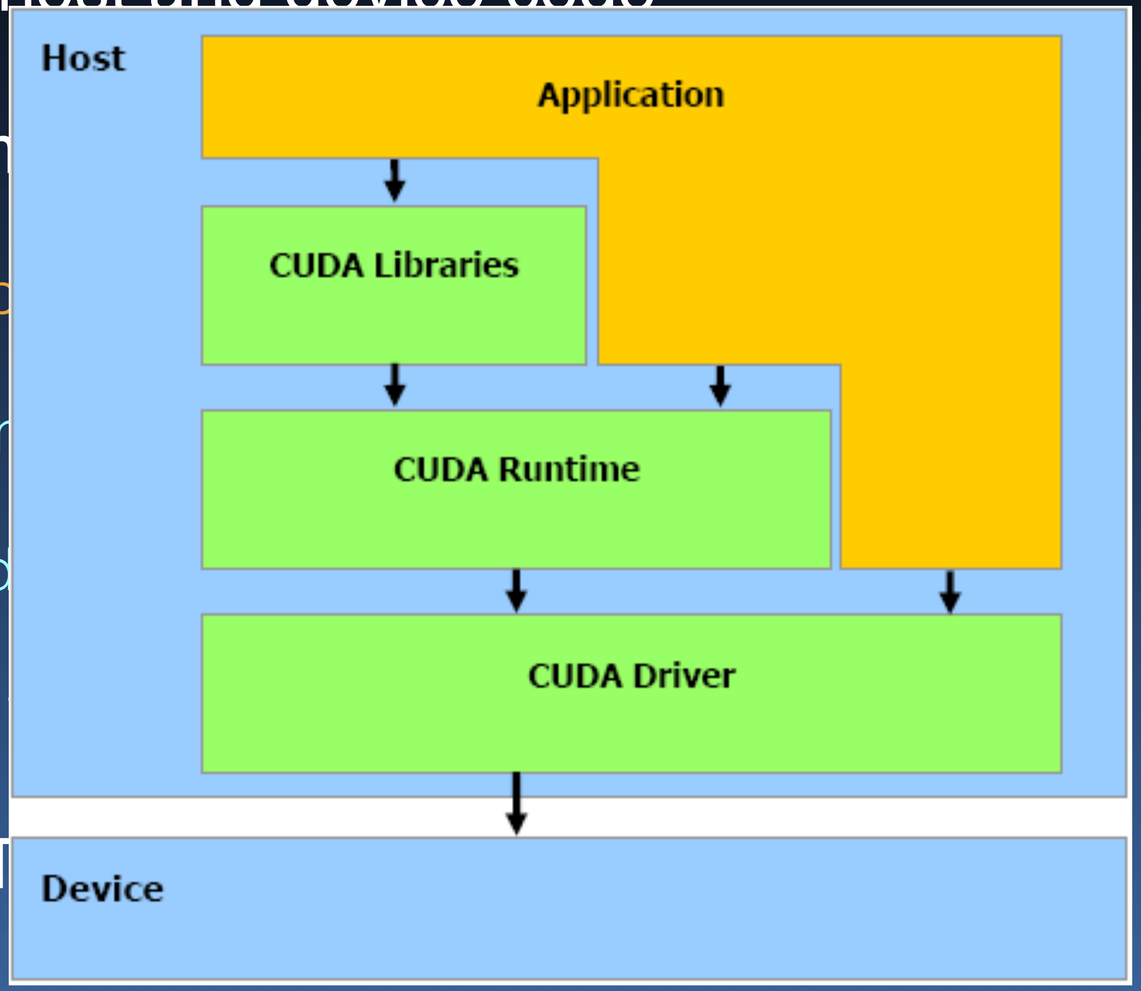
```
__device__ float filter[N];

__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

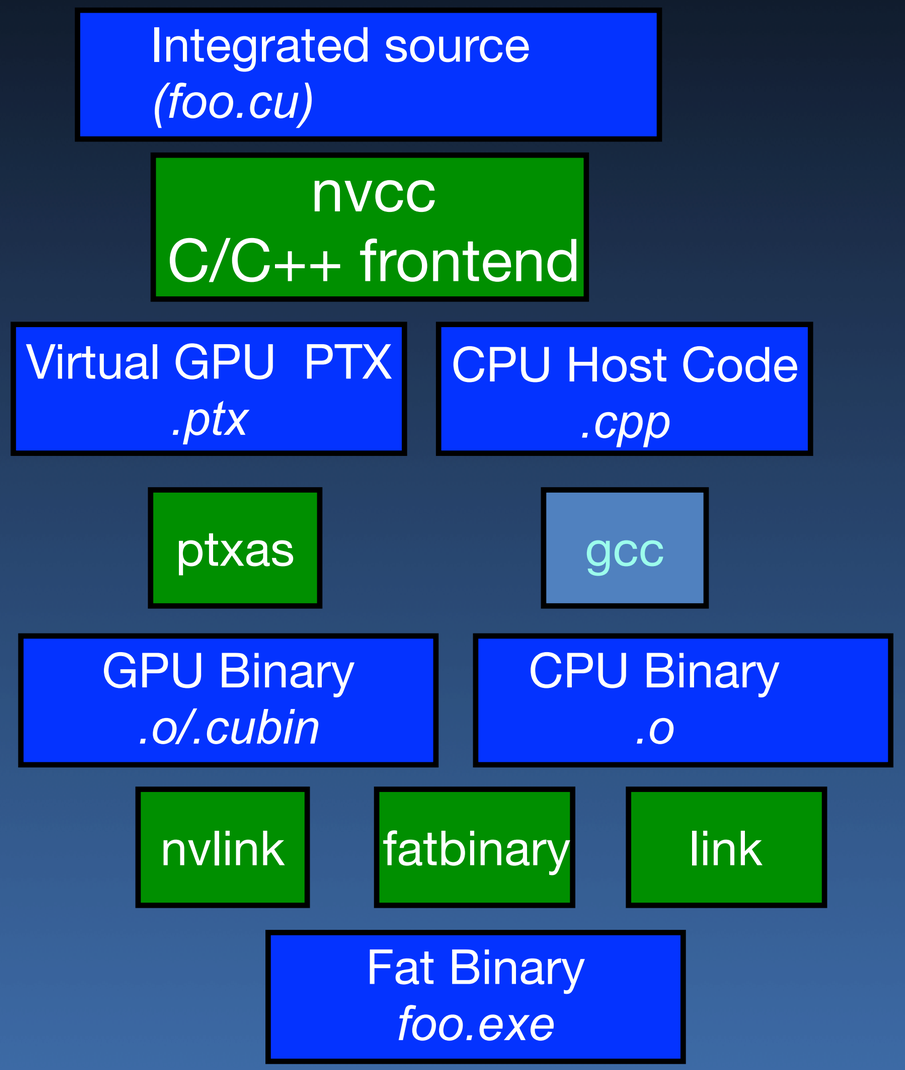
- Source files (.cu) have a mix of host and device code
- nvcc separates device code from host code
  - ➡ compiles device code into PTX/cubin
  - ➡ host code is output as C source (and C compiler invoked)
  - ➡ PTX/cubin incorporated in host code as a global initialized data array
  - ➡ includes cudart (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

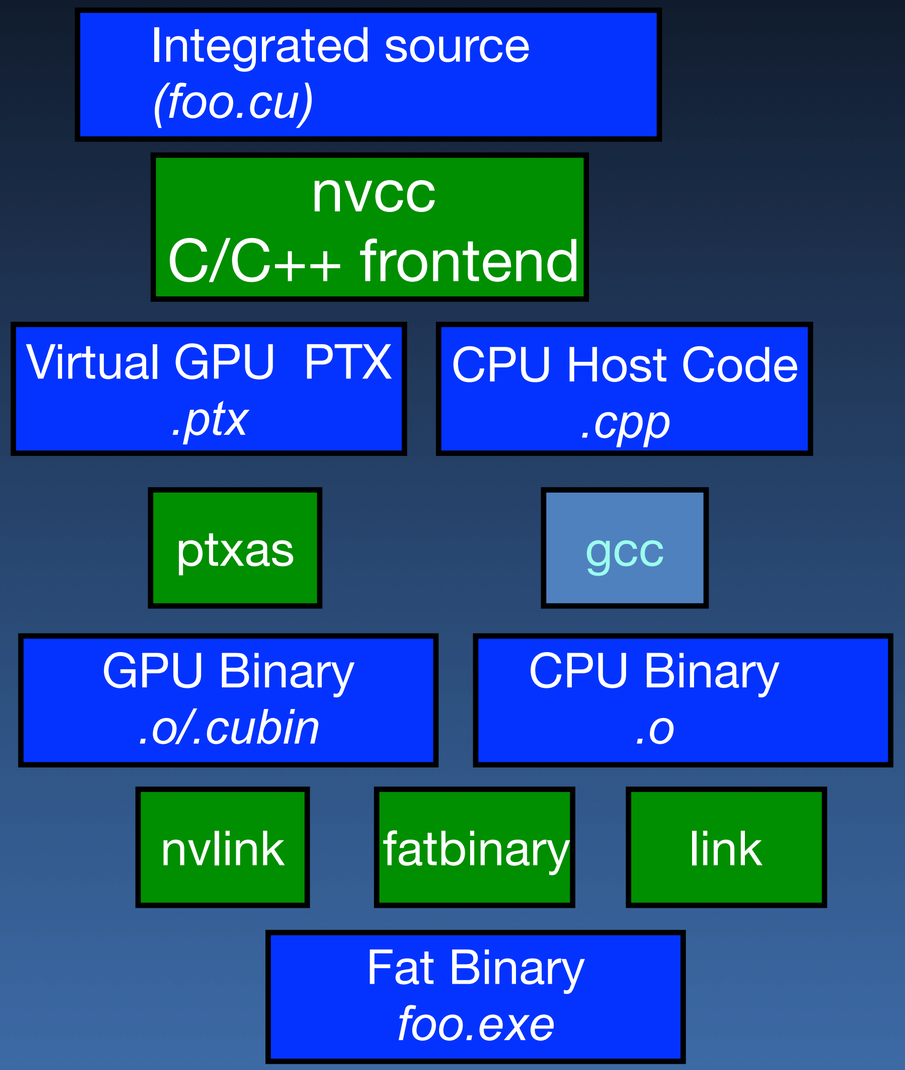
- Source files (.cu) have a mix of host and device code
- nvcc separates device code fron
  - ➡ compiles device code into PTX/cub
  - ➡ host code is output as C source (ar
  - ➡ PTX/cubin incorporated in host coc
  - ➡ includes cudart (CUDA C runtime)
- Possible to load and execute PT



Cuda Dev Tools



Cuda Dev Tools



cuobjdump

nvdisasm

cuda-gdb

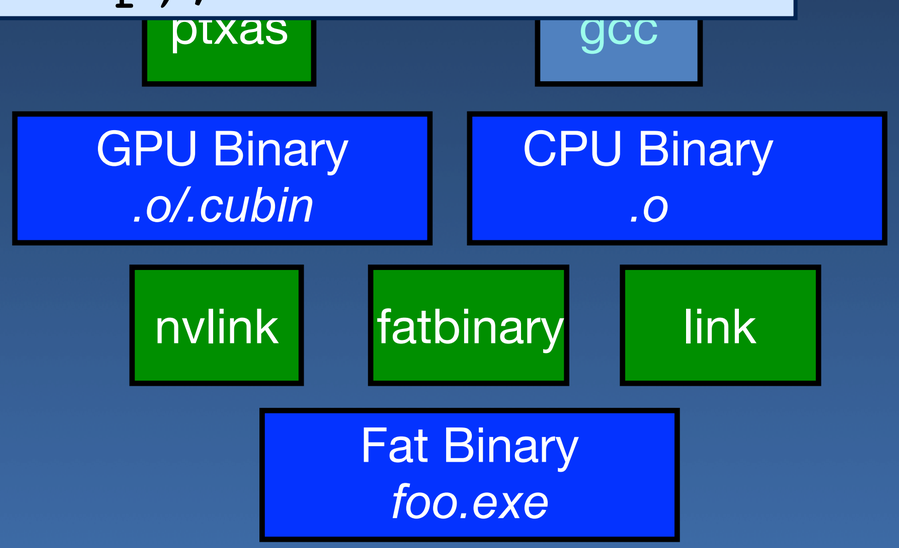
cuda-memcheck

Parallel Nsight

CUDA Visual Profiler

Cuda Dev Tools

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);
GPUKernel<<<B,T>>>(p);
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

Cuda Dev Tools

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);
GPUKernel<<<B,T>>>(p);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float ms;
cudaEventElapsedTime(&ms, start, stop);
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

cuobjdump

nvdisasm

cuda-gdb

cuda-memcheck

ptxa

GPU Bi
.o/.cu

nvlin

foo.exe

On HPC: Need GPU nodes

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

module add compiler/cuda/11.0/compilervars

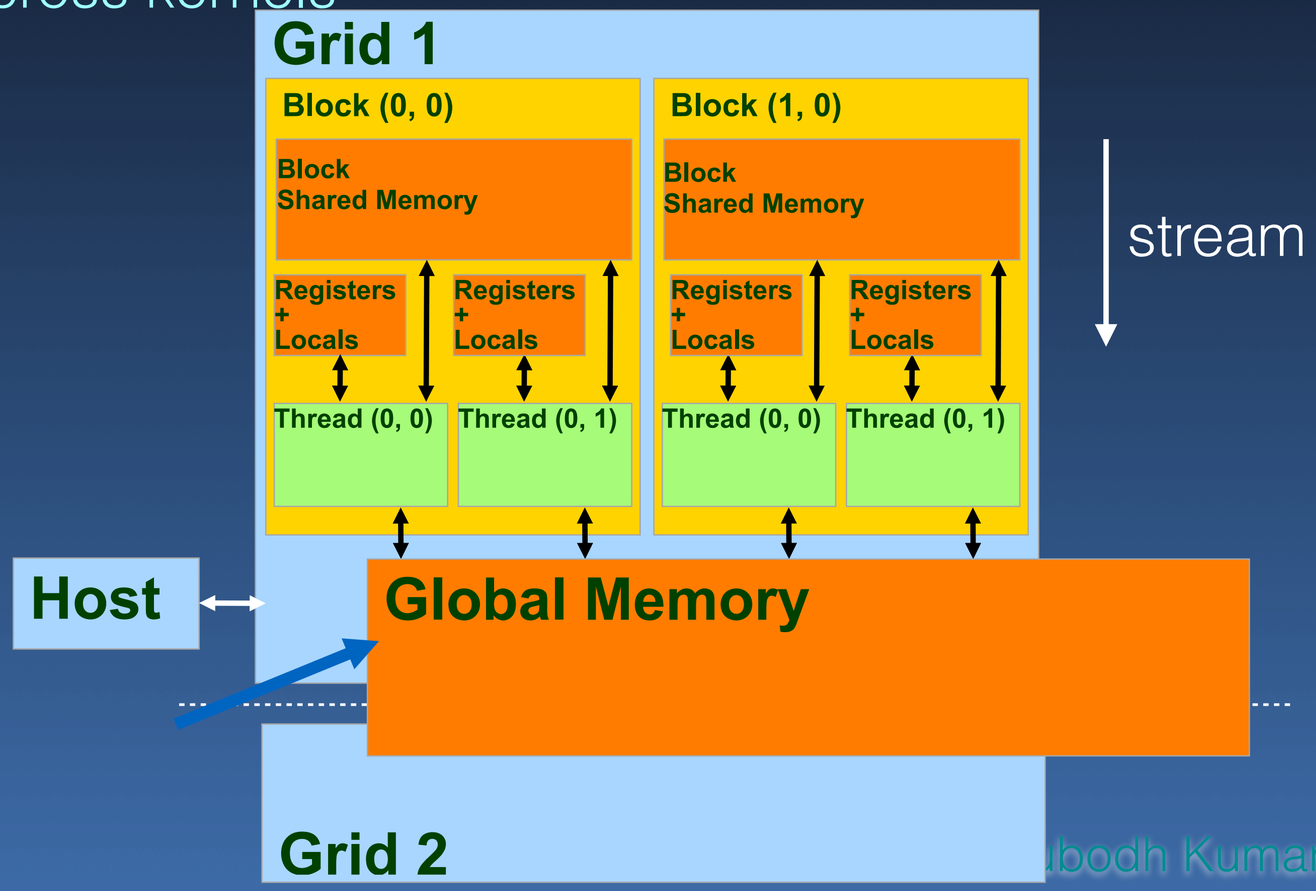
Available: V100 (CC7.0)

On css:

All nodes have GPU (RTX 3070 CC 8.6)

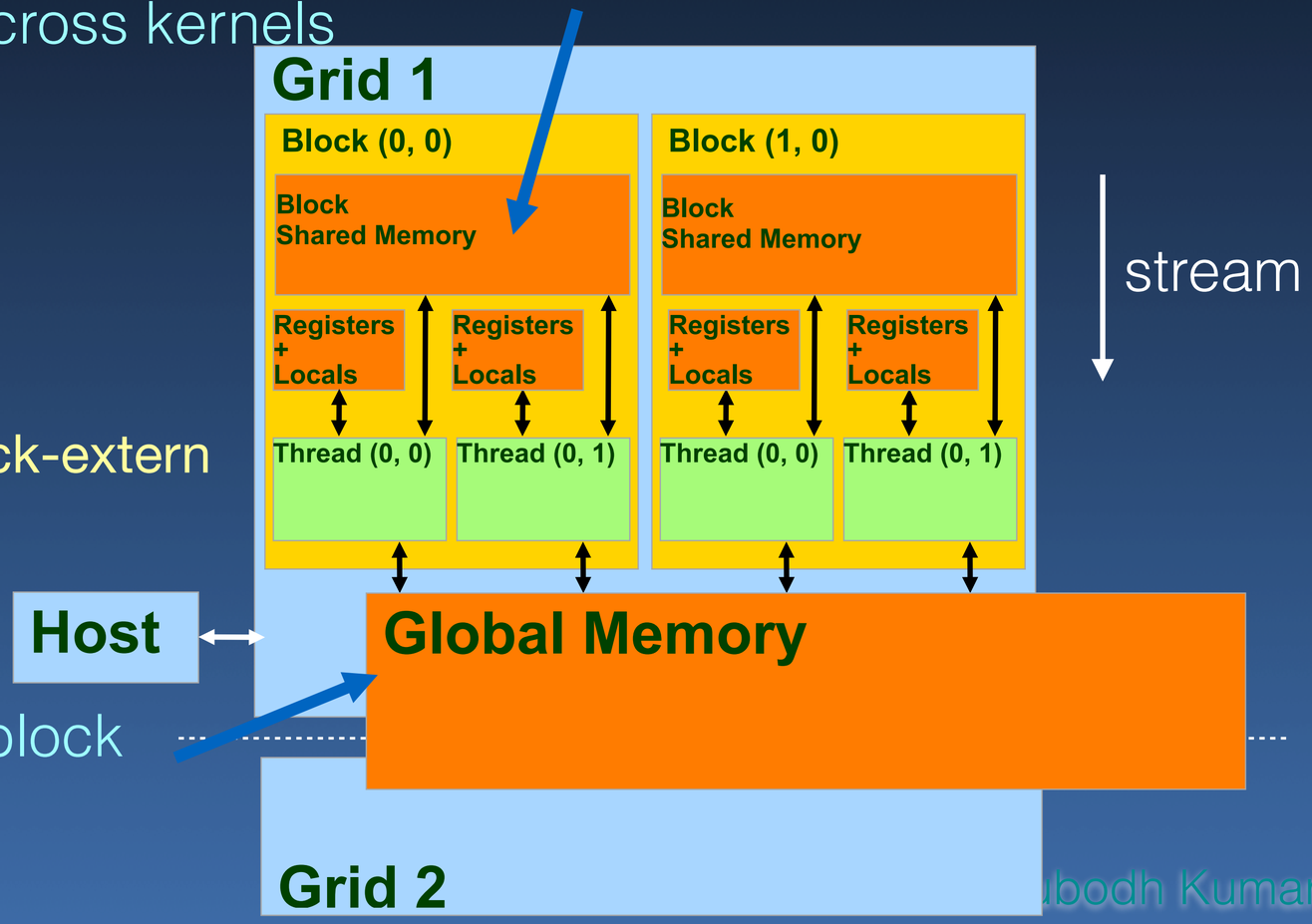
CUDA Memory Model

- Global memory cudaMalloc, __device__, __managed__
  - ➡ Host ↔ Device data communication
  - ➡ Visible to all threads, persistent across kernels
  - ➡ Long latency
  - ➡ Through L1 and L2



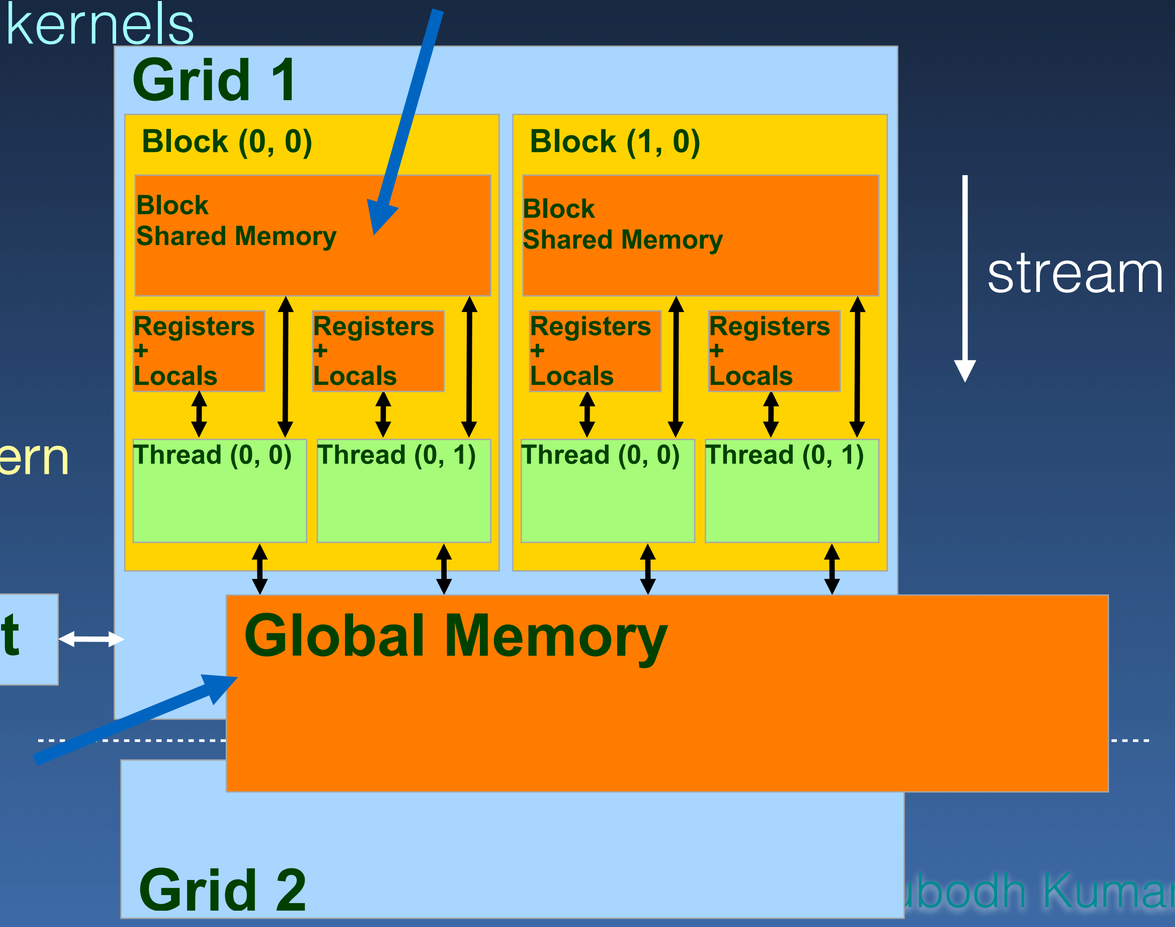
CUDA Memory Model

- **Global memory** cudaMalloc, __device__, __managed__
  - ➡ Host ↔ Device data communication
  - ➡ Visible to all threads, persistent across kernels
  - ➡ Long latency
  - ➡ Through L1 and L2
- **Shared Memory** __shared__, Block-extern
  - ➡ Fast memory (user-managed L1)
  - ➡ Shared across block, Lifetime of block



CUDA Memory Model

- **Global memory** cudaMalloc, __device__, __managed__
  - ➡ Host ↔ Device data communication
  - ➡ Visible to all threads, persistent across kernels
  - ➡ Long latency
  - ➡ Through L1 and L2
- **Shared Memory** __shared__, Block-extern
  - ➡ Fast memory (user-managed L1)
  - ➡ Shared across block, Lifetime of block



Other memory segments: Constant and Texture

Host ↔ Device Memory

- Allocation and Transfer can be both Explicit and Implicit

- Explicit Allocation
  - cudaMalloc(..)
- Implicit Allocation
  - declare variables __managed__
  - Kernel Launch Arguments

- Explicit Transfer
  - cudaMemcpy(..), cudaMemcpyToSymbol(..)
- Implicit Transfer
  - On page-fault
  - Kernel Arguments

Device Memory Allocation

- Code example:
  - ➡ Allocate a 64 * 64 single precision float array
  - ➡ Attach the allocated storage to dM

```
TILE = 64;
float *dM;
int size = TILE * TILE * sizeof(float);
…
cudaMalloc((void**)&dM, size);
…
cudaFree(dM);
```

Device Memory Allocation

- Code example:
  - ➡ Allocate a 64 * 64 single precision float array
  - ➡ Attach the allocated storage to dM

```
TILE = 64;
float *dM;
int size = TILE * TILE * sizeof(float);
...
cudaMalloc((void**)&dM, size);
...
cudaFree(dM);
```

Also see:

cudaHostAlloc(..)
Page-locked memory allocation
Can be Mapped on host and device

cudaMallocManaged(&x, nbytes);
Unified-memory allocation

cudaMallocPitch(..), cudaMalloc3D(..)
2D/3D arrays *aligned* to support parallel IO

Host ↔ Device Transfer

```
cudaMemcpy(dM, M, size, cudaMemcpyHostToDevice);
cudaMemcpy(M, dM, size, cudaMemcpyDeviceToHost);
cudaMemcpy(M, dM, size, cudaMemcpyDefault);
```

Predefined Constants

- Blocking call
- M is in host memory allocated regularly
- dM is in device memory

Host ↔ Device Transfer

```
cudaMemcpy(dM, M, size, cudaMemcpyHostToDevice);
cudaMemcpy(M, dM, size, cudaMemcpyDeviceToHost);
cudaMemcpy(M, dM, size, cudaMemcpyDefault);
```

Predefined Constants

- Blocking call But see cudaMemcpyAsync().
- M is in host memory allocated regularly
- dM is in device memory

Host ↔ Device Transfer

```
cudaMemcpy(dM, M, size, cudaMemcpyHostToDevice);
cudaMemcpy(M, dM, size, cudaMemcpyDeviceToHost);
cudaMemcpy(M, dM, size, cudaMemcpyDefault);
```

Predefined Constants

- Blocking call

But see cudaMemcpyAsync().

- M is in host memory allocated regularly
- dM is in device memory

Also see:

cudaMemcpy2D(..)
cudaMemcpy3D(..)
cudaMemcpyToSymbol(..)

Memcpy Example

```
size_t size = N * sizeof(float);
float* hA = (float*)malloc(size); // Allocate vector @host
float *dA, *dB, *dC;              // Device vector addresses

cudaMalloc(&dA, size);            // Allocate vectors @device
cudaMalloc(&dB, size);            // dA, dB are opaque on CPU
cudaMemcpy(dA, hA, size, cudaMemcpyDefault);// Copy hA→dA

// Pass kernel to GPU for execution
GPUfunc<<<blocksInGrid, threadsPerBlock>>>(dA, dB, N);

cudaMemcpy(hB, dB, size, cudaMemcpyDeviceToHost); // Copy dB→HA

cudaFree(dA);  // Free device memory
cudaFree(dB);
```

Managed Memory

```
__device__ __managed__ int N = 65535;        // Managed
float *bothp;
cudaMallocManaged(&bothp, N*sizeof(float)); // Managed

initialize(bothp, N);  // initialize bothp on host

Kernel<<<N/1024,1024>>>(bothp); // Launch on GPU
cudaDeviceSynchronize(); // Block until Kernel completes

cpuProcess(bothp); // Use the values computed by GPU
cudaFree(bothp);    // Free memory
```

```
__device__ float F(float x) {
   ...
}

__global__ void Kernel (float *bothp)
{
     int index = blockIdx.x * blockDim.x +
                                   threadIdx.x;
     if (index < N)
          bothp[index] = F(bothp[index]);
}
```

Managed Memory

```
__device__ __managed__ int N = 65535;        // Managed
float *bothp;
cudaMallocManaged(&bothp, N*sizeof(float)); // Managed

initialize(bothp, N);  // initialize bothp on host

Kernel<<<N/1024,1024>>>(bothp); // Launch on GPU
cudaDeviceSynchronize(); // Block until Kernel completes

cpuProcess(bothp); // Use the values computed by GPU
cudaFree(bothp);    // Free memory
```

```
// Prefetch the data to the GPU
int dev = -1;
cudaGetDevice(&dev);
cudaMemPrefetchAsync(bothp, N*sizeof(float), dev);
```

```
__device__ float F(float x) {
   ...
}

__global__ void Kernel (float *bothp)
{
     int index = blockIdx.x * blockDim.x +
                                   threadIdx.x;
     if (index < N)
          bothp[index] = F(bothp[index]);
}
```

CUDA Function Qualifiers

- __global__ defines a kernel function
  - ➡ Must return void
  - ➡ called from host, run on device
  - ➡ No recursion

```
__device__ float dSomeName() {}
__global__ void  kSomeName() {}
```

- __device__ are executed and called on device
- __host__ qualifier also exists
  - ➡ __device__ and __host__ can be used together

Kernel Invocation

- kernels called with execution configuration:

```
__global__ void KernelFunc(...);
dim3   DimGrid(128, 32);       // Total 4096 thread blocks
dim3   DimBlock(8, 8, 8);      // 512 threads/block
size_t SharedMemBytes = 2048; // 2KB of shared mem per block
KernelFunc<<<DimGrid, DimBlock, SharedMemBytes>>>(...);
```

- Calls to a kernel function are asynchronous

  ➡ Parameters are passed through shared/constant memory

- Call cudaDeviceSynchronize() to block on the kernels in flight

Kernel Invocation

- kernels called with execution configuration:

```
__global__ void KernelFunc(...);
dim3   DimGrid(128, 32);       // Total 4096 thread blocks
dim3   DimBlock(8, 8, 8);      // 512 threads/block
size_t SharedMemBytes = 2048; // 2KB of shared mem per block
KernelFunc<<<DimGrid, DimBlock, SharedMemBytes>>>(...);
```

Allocated on launch

- Calls to a kernel function are asynchronous

➡ Parameters are passed through shared/constant memory

- Call cudaDeviceSynchronize() to block on the kernels in flight

Kernel Invocation

- kernels called with execution configuration:

```
__global__ void KernelFunc(...);
dim3   DimGrid(128, 32);       // Total 4096 thread blocks
dim3   DimBlock(8, 8, 8);      // 512 threads/block
size_t SharedMemBytes = 2048; // 2KB of shared mem
KernelFunc<<<DimGrid, DimBlock, SharedMemBytes>>>(...);
```

```
extern __shared__ float dblock[];
__global__ void KernelFunc()
{
    float* mat1 = (float*) dblock;
    float* mat2 = mat1 + 128;
}
```

- Calls to a kernel function are asynchronous

➡ Parameters are passed through shared/constant memory

- Call cudaDeviceSynchronize() to block on the kernels in flight

Review

- Introduction to GPU architecture
- CUDA execution model: Grid, Block, Threads
  - ➡ Synchronization between threads
- CUDA streaming model
- CUDA memory model
  - ➡ CPU memory, Global memory, Shared memory, Unified Memory
  - ➡ Weakly consistent
  - ➡ Memory allocation and transfer