COL380

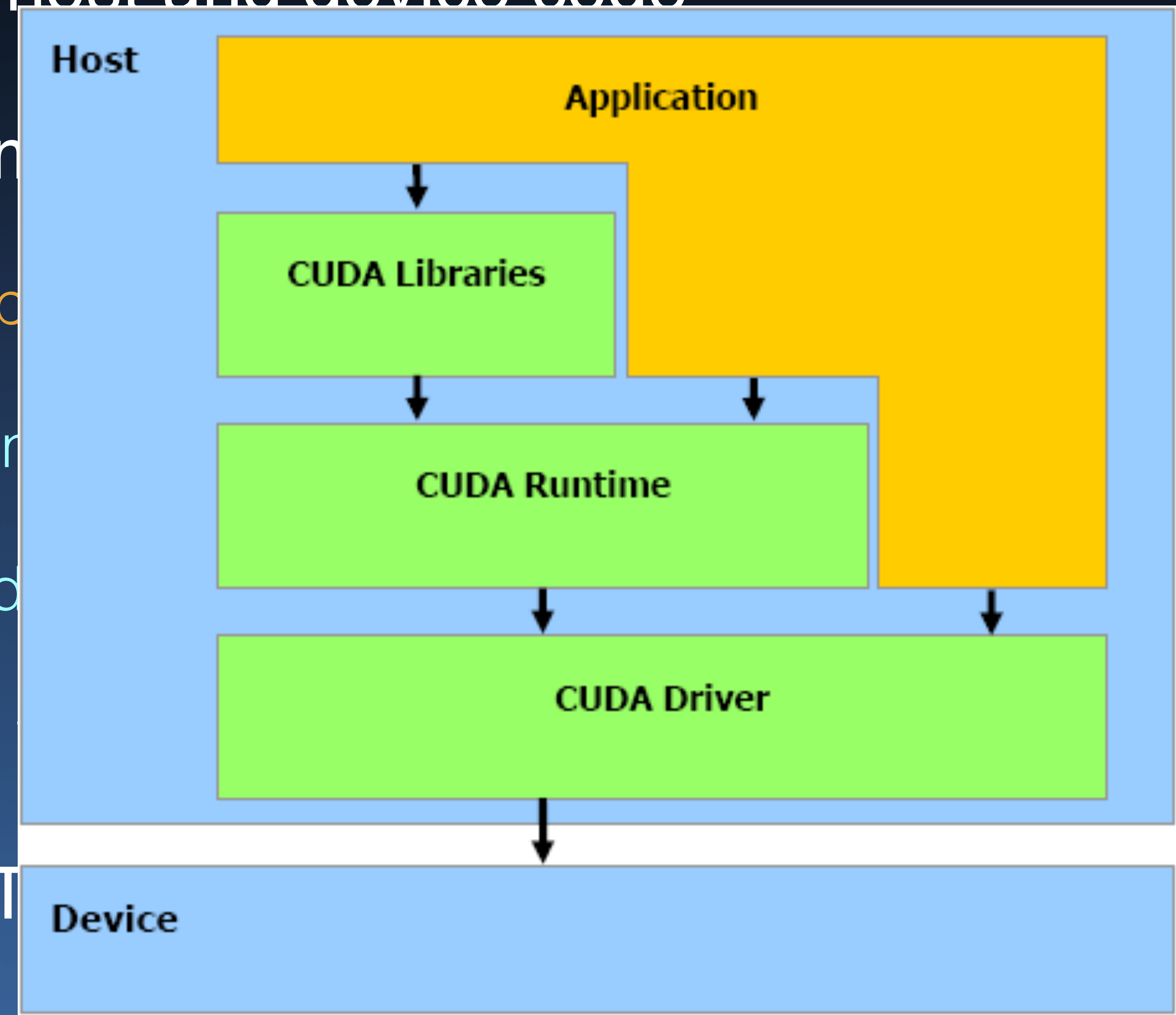
Introduction to
Parallel & Distributed Programming

CUDA Tool-chain

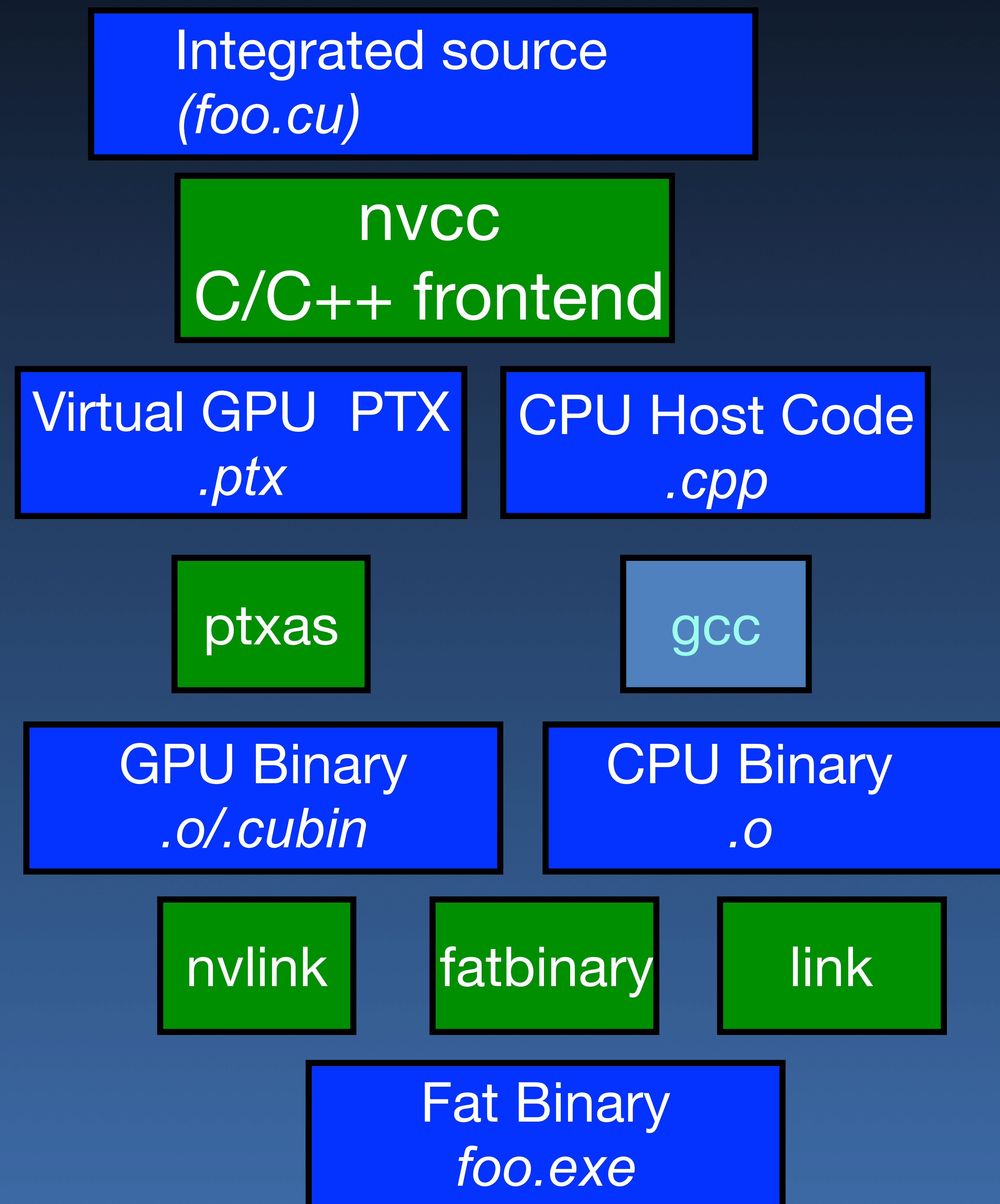
- Source files (.cu) have a mix of host and device code
- nvcc separates device code from host code
  - ➡ compiles device code into PTX/cubin
  - ➡ host code is output as C source (and C compiler invoked)
  - ➡ PTX/cubin incorporated in host code as a global initialized data array
  - ➡ includes cudart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

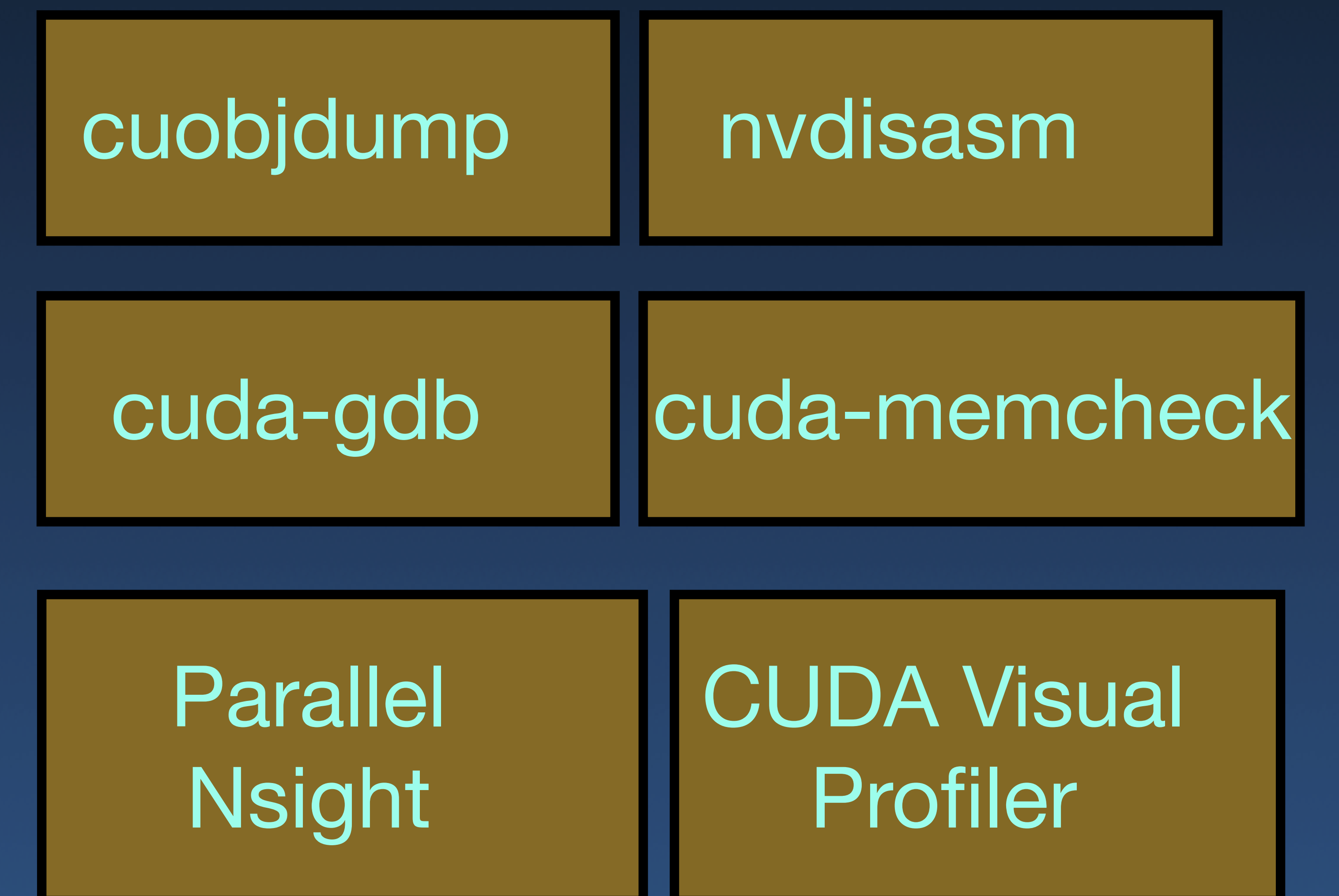
CUDA Tool-chain

- 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 cod
  - ➡ includes cudart (CUDA C runtime)
- Possible to load and execute PT



Cuda Dev Tools

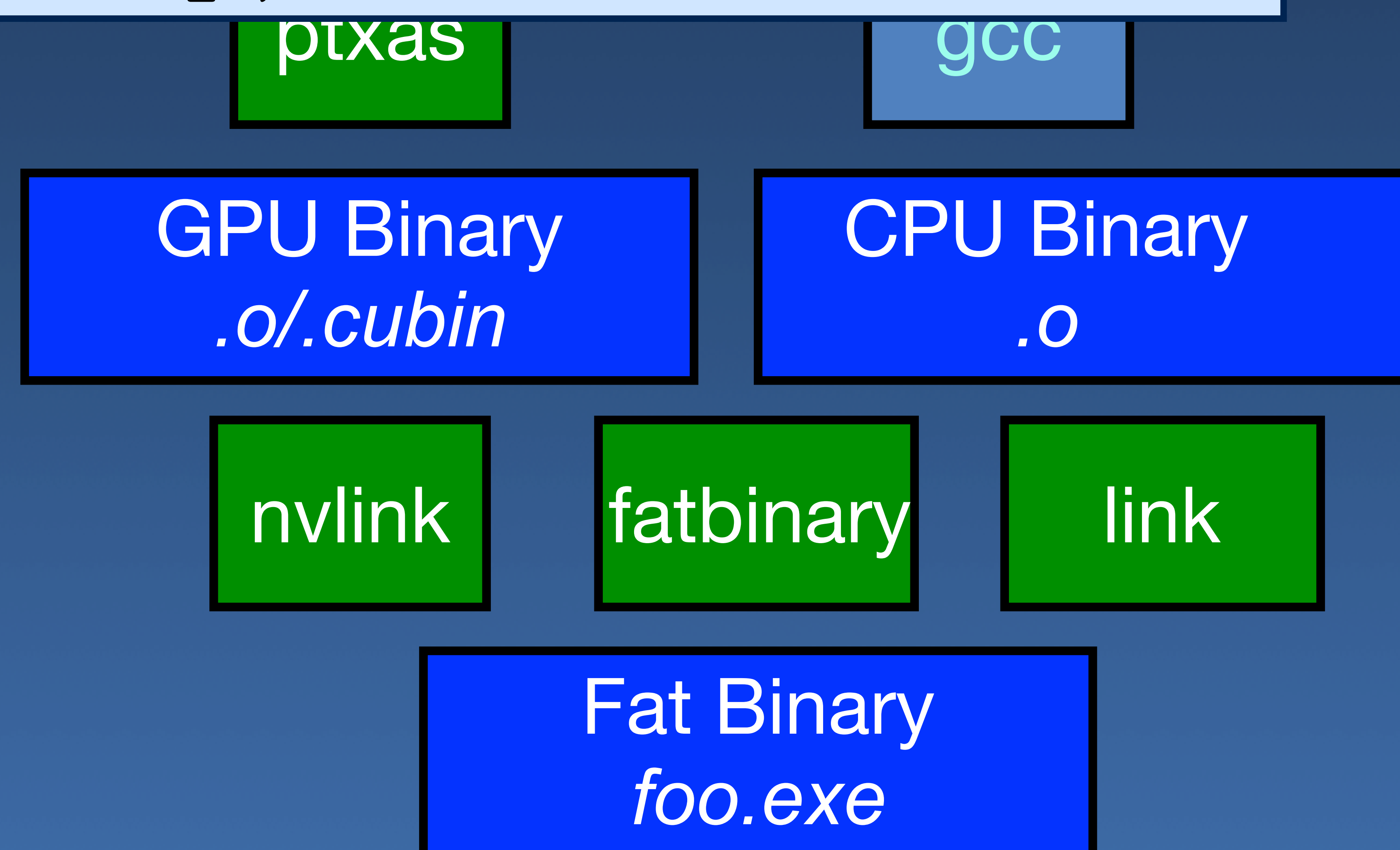


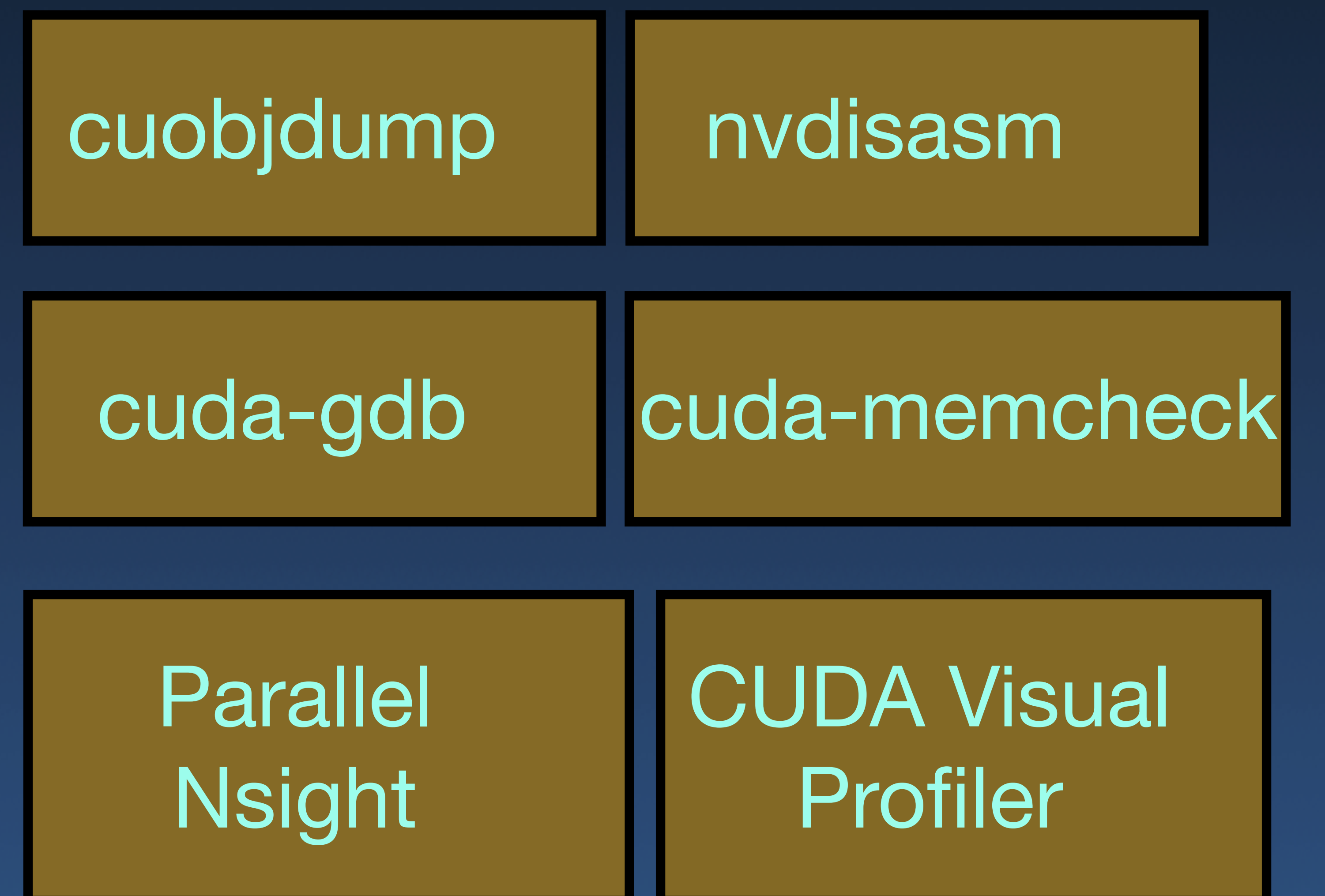


Subodh Kumar

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);
```





Subodh Kumar

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/.cub

nvlink

foo.exe

On HPC: Need GPU nodes

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

module add compiler/cuda/11.0/compilervars

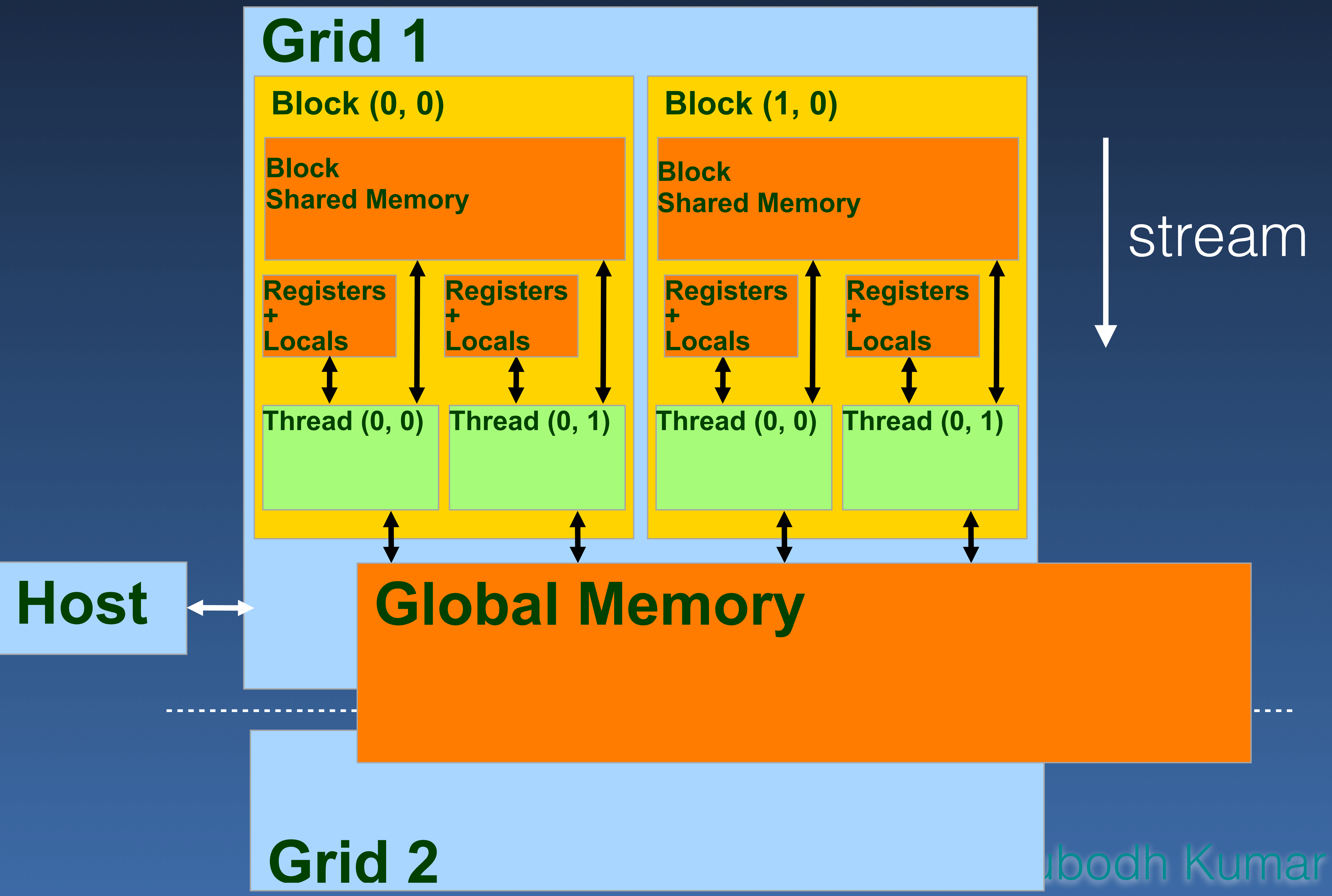
Available: V100 (CC7.0)

On css:

All nodes have GPU (RTX 3070 CC 8.6)

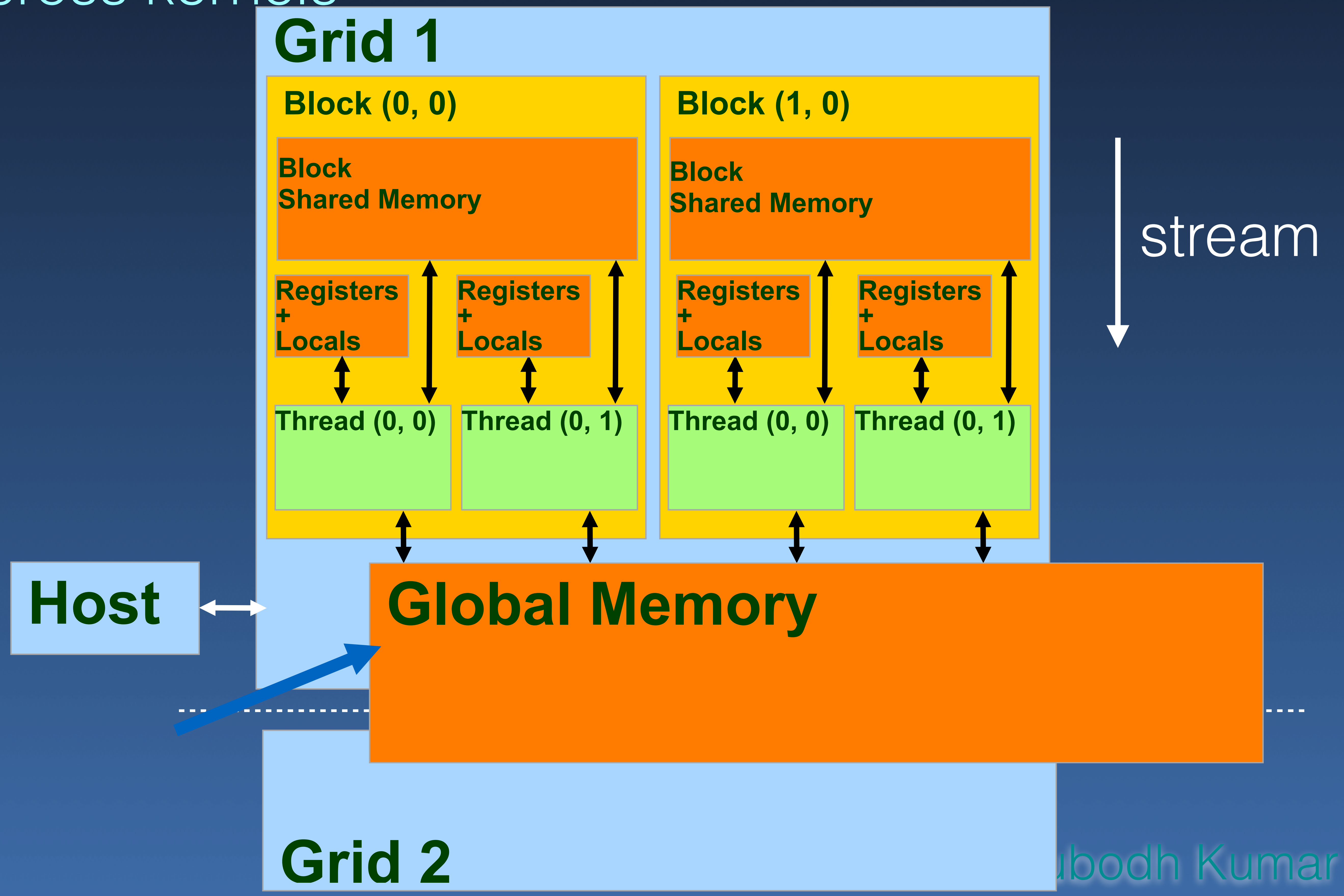
Subodh Kumar

CUDA Memory Model



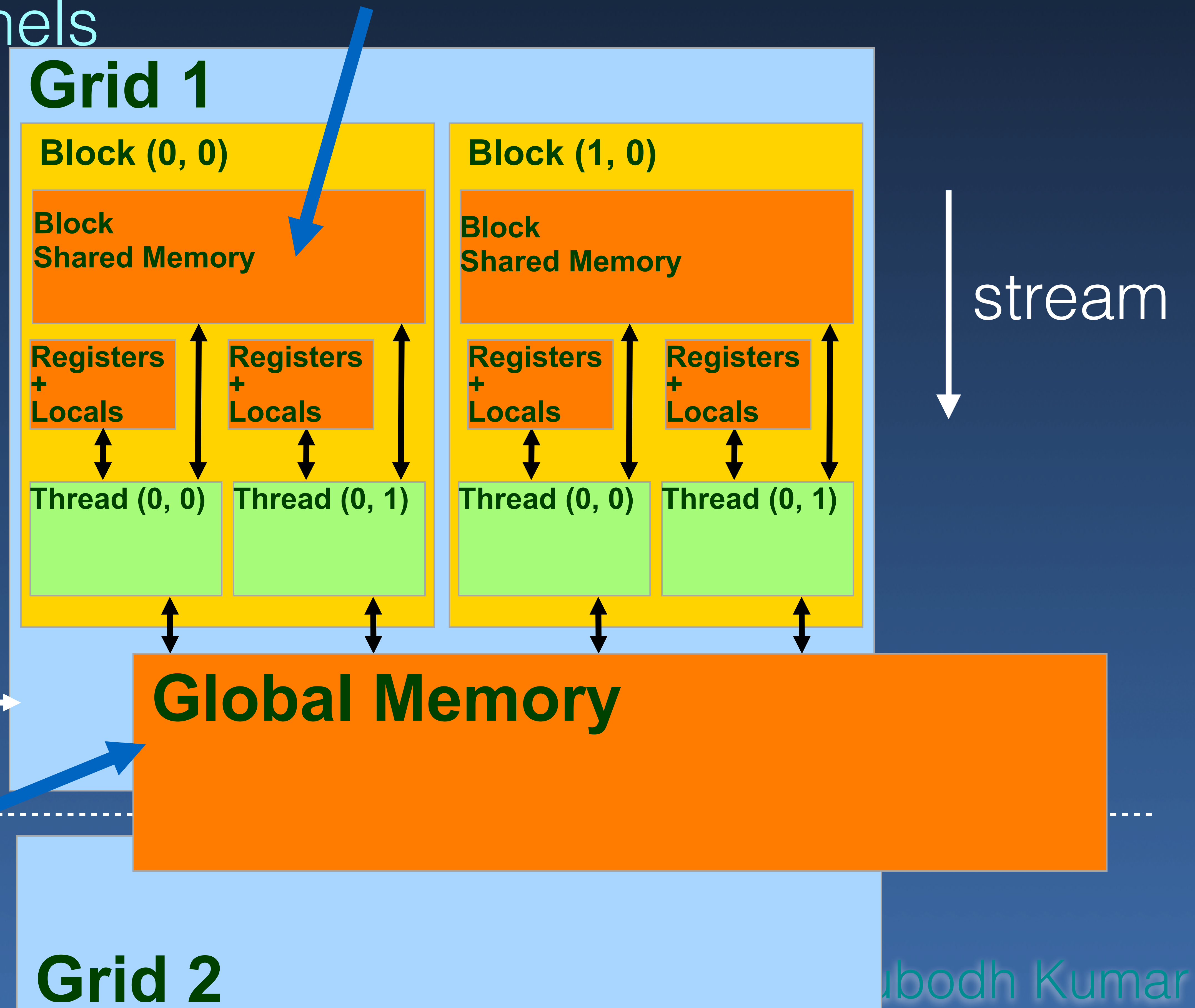
CUDA Memory Model

- 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

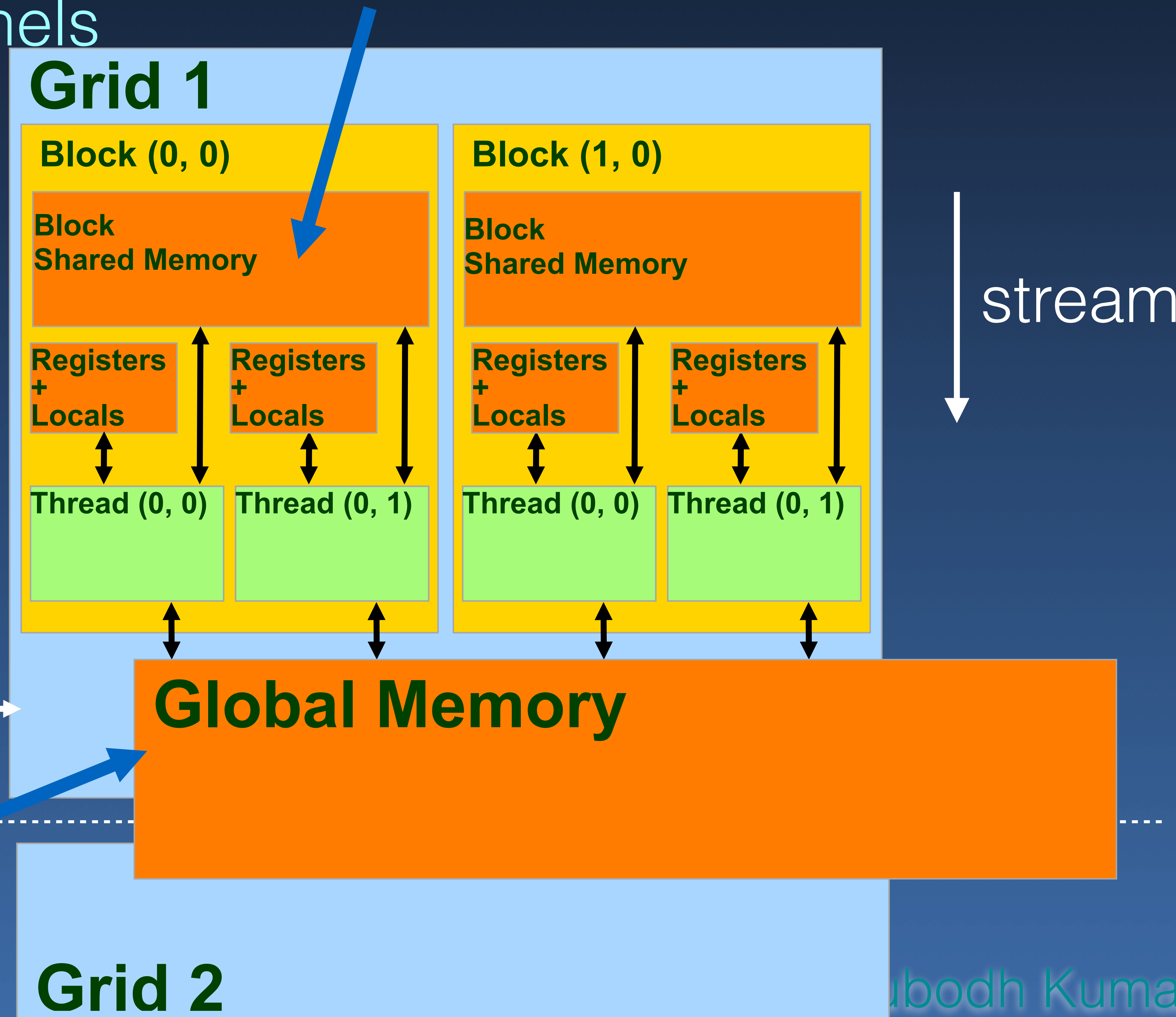
- Global memory cudaMalloc, __device__, __managed__
  - ➡ Host ↔ Device data communication
  - ➡ Visible to all threads, persistent across kernels
  - ➡ Long latency
  - ➡ Through L1 and L2
- Shared Memory __shared__, Block-extern
  - ➡ Fast memory (user-managed L1)
  - ➡ Shared across block, Lifetime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



Subodh Kumar

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

ubodh Kumar

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);
```

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

Also see:

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

Subodh Kumar

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);
```

Subodh Kumar

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]);
}
```

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

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

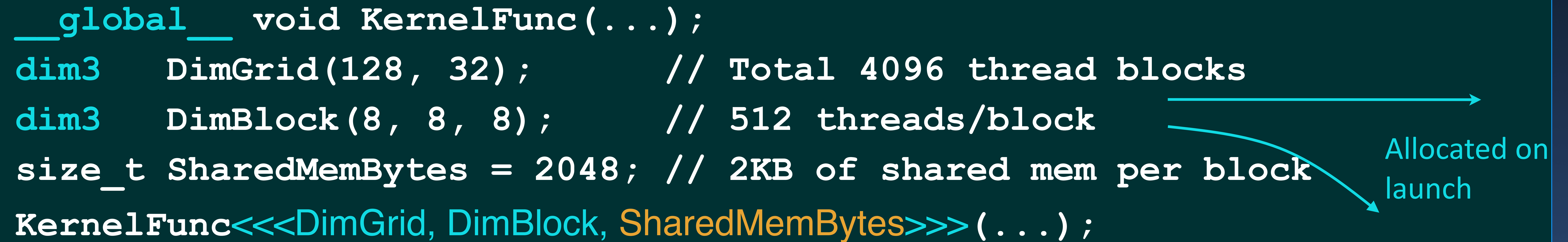
- kernels called with execution configuration:

```
__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:



- 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

e.g., Matrix Multiplication

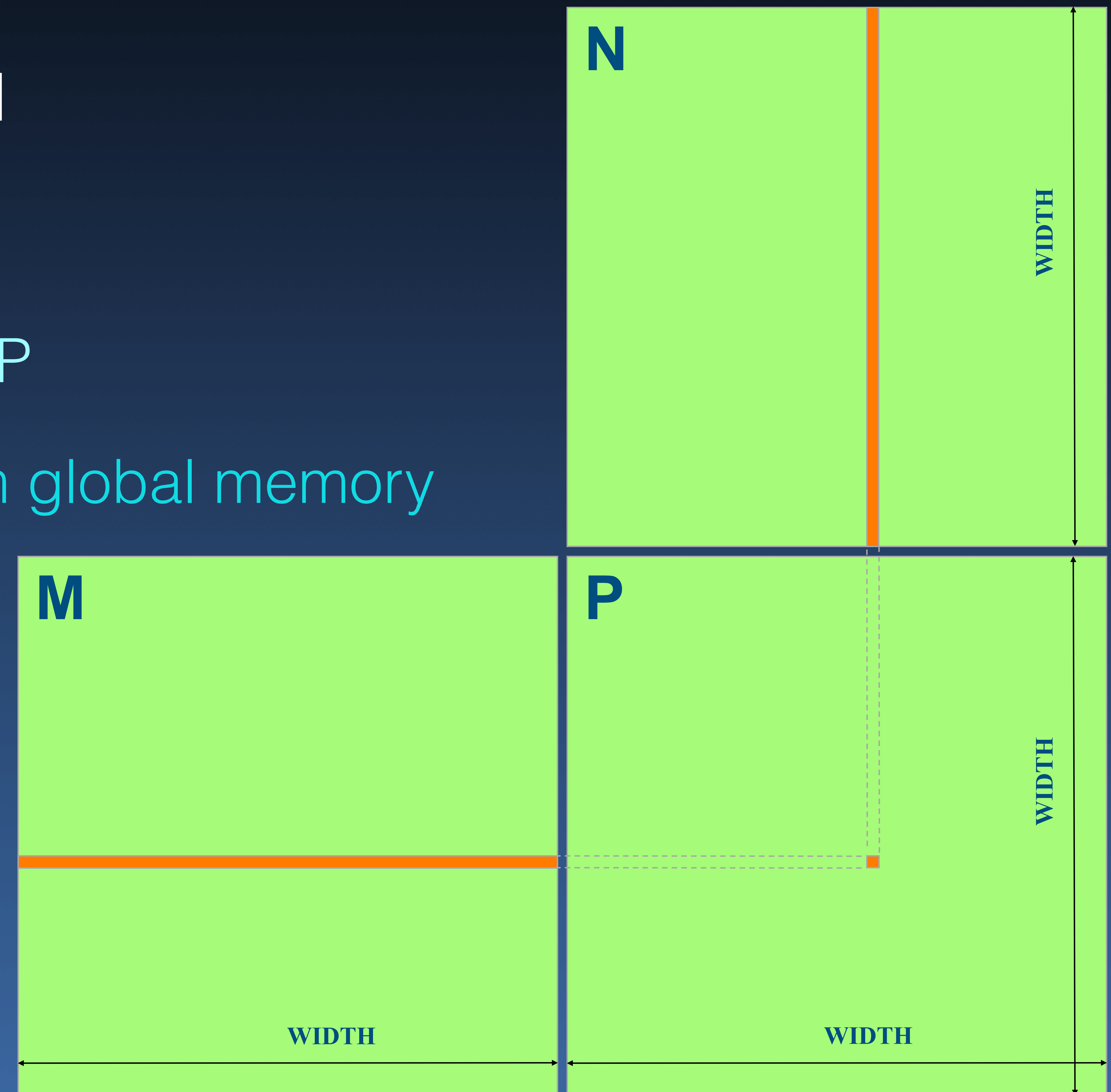
- Illustrates basic memory/thread management
- Assumes square matrix for simplicity
- No shared memory usage yet
- Local, register usage
- Thread ID usage
- Memory data transfer between host and device

Cuda functions return cudaError_t

Subodh Kumar

GPU Matrix Multiplication

- P = M * N of size WIDTH x WIDTH
- Without tiling:
  - One thread calculates one element of P
  - M and N are loaded WIDTH times from global memory

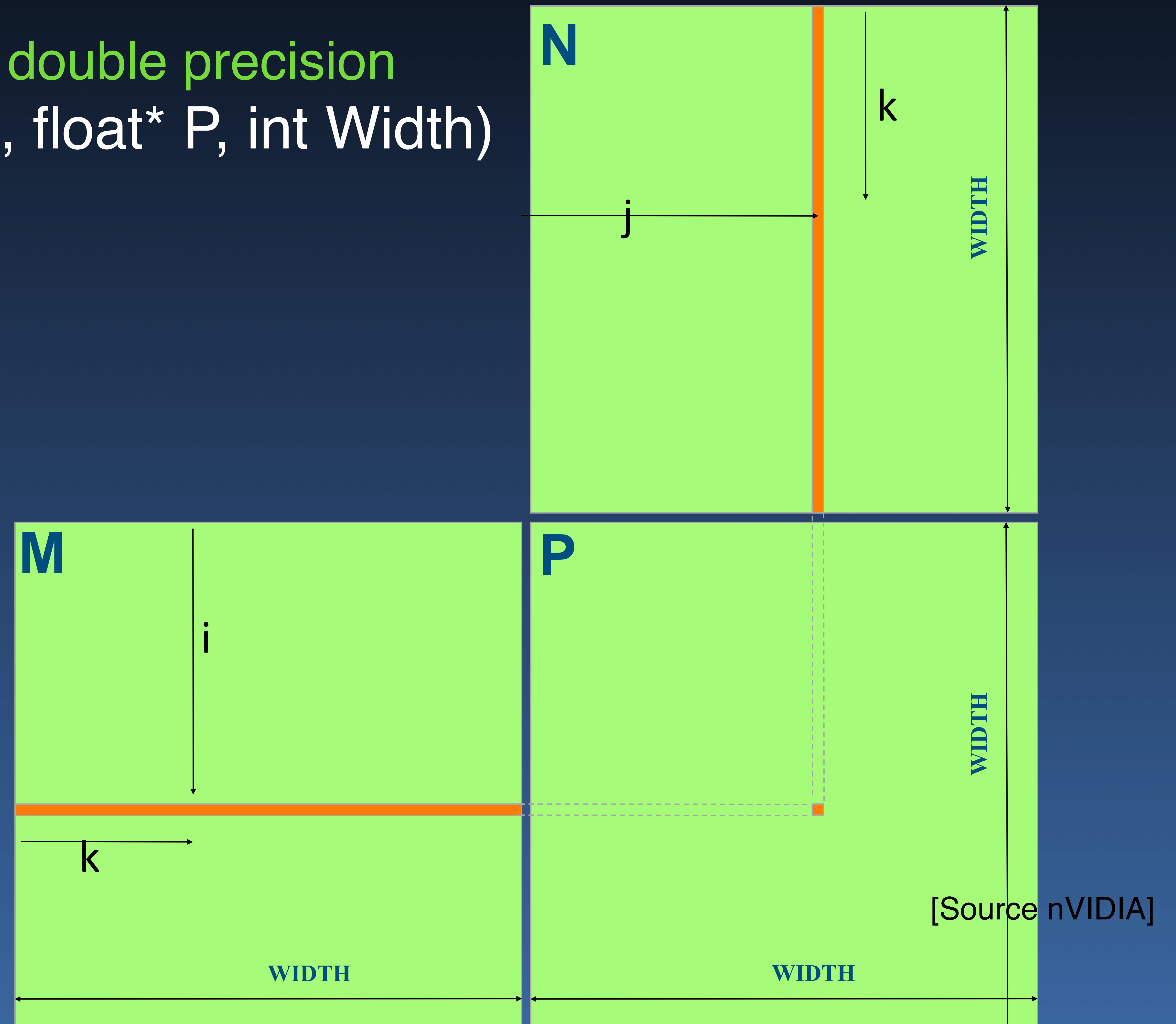


[Source nVIDIA]

Subodh Kumar

Multiply Matrices on Host

```
// Matrix multiplication on the (CPU) host in double precision
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
  for (int i = 0; i < Width; ++i)
    for (int j = 0; j < Width; ++j) {
      double sum = 0;
      for (int k = 0; k < Width; ++k) {
        double a = M[i * width + k];
        double b = N[k * width + j];
        sum += a * b;
      }
      P[i * Width + j] = sum;
    }
}
```



Matrix Data Transfer

```
void MatrixMulOnDevice(float* M, float* N,
                       float* P, int Width)
{
    int size = Width * Width * sizeof(float);
    float *dM, *dN, *dP;
    …
1.  // Allocate and Load M, N to device memory
    cudaMalloc(&dM, size);
    cudaMemcpy(dM, M, size, cudaMemcpyDefault);
    cudaMalloc(&dN, size);
    cudaMemcpy(dN, N, size, cudaMemcpyDefault);
    // Allocate P on the device
    cudaMalloc(&dP, size);
2. // Kernel invocation code (to be shown later)
     …
3.  // Read P from the device
    cudaMemcpy(P, dP, size, cudaMemcpyDeviceToHost);
    // Free device matrices
    cudaFree(dM); cudaFree(dN); cudaFree (dP);
}
```

See: cudaMemcpyAsync

Blocks until copy is complete

[Example from Kirk & Hwu]

Matrix Data Transfer

```
void MatrixMulOnDevice(float* M, float* N,
                       float* P, int Width)
{
   int size = Width * Width * sizeof(float);
   float *dM, *dN, *dP;
   …
1. // Allocate and Load M, N to device memory
   cudaMalloc(&dM, size);
   cudaMemcpy(dM, M, size, cudaMemcpyDefault);
   cudaMalloc(&dN, size);
   cudaMemcpy(dN, N, size, cudaMemcpyDefault);
   // Allocate P on the device
   cudaMalloc(&dP, size);
2. // Kernel invocation code (to be shown later)
     …
3. // Read P from the device
   cudaMemcpy(P, dP, size, cudaMemcpyDeviceToHost);
   // Free device matrices
   cudaFree(dM); cudaFree(dN); cudaFree (dP);
}
```

See: cudaMemcpyAsync

```
cudaMallocManaged(..);
// invoke Kernel
cudaFree(..);
```

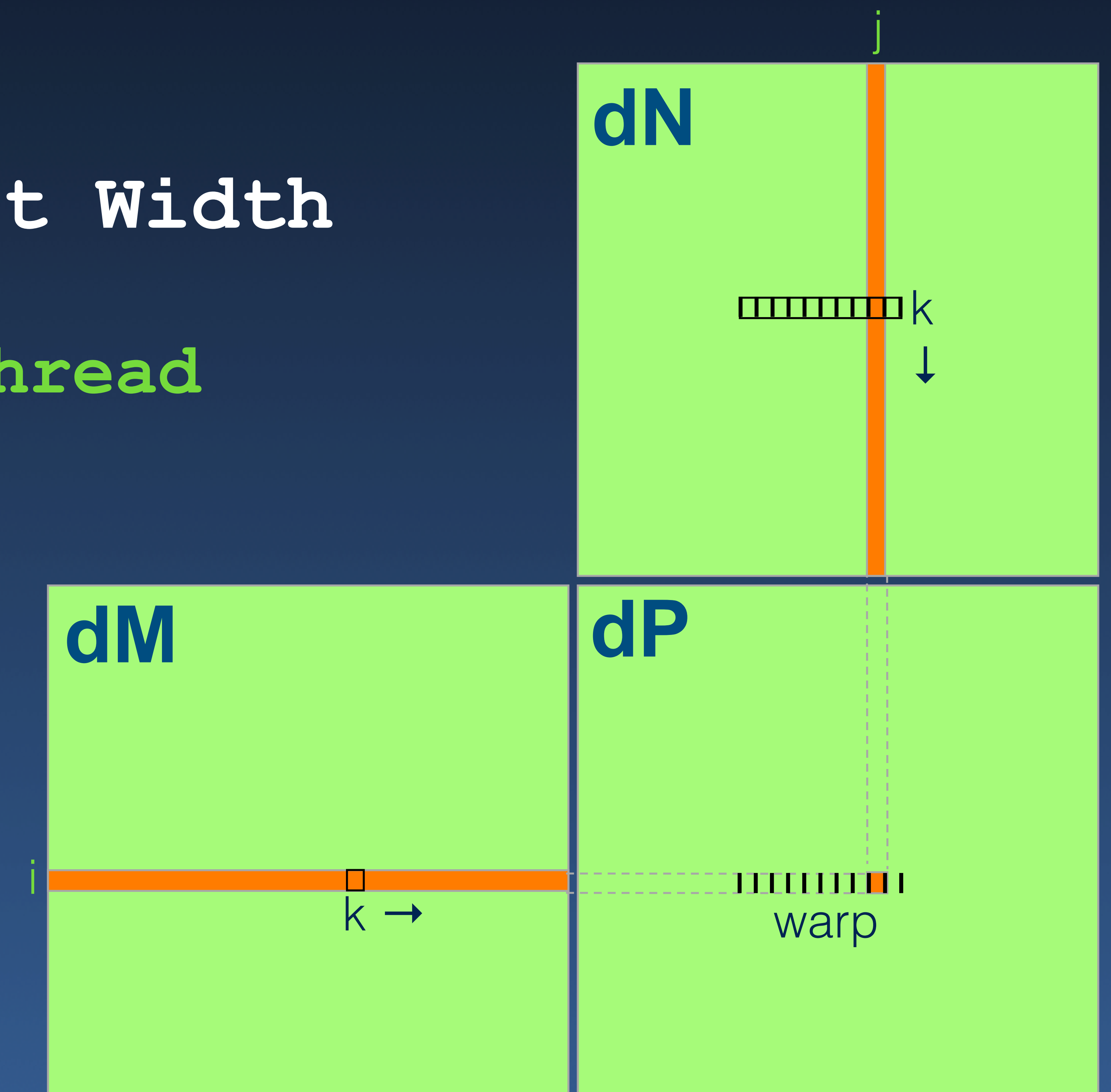
[Example from Kirk & Hwu]

Kernel Function

// Matrix multiplication kernel – per thread code

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

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



Subodh Kumar

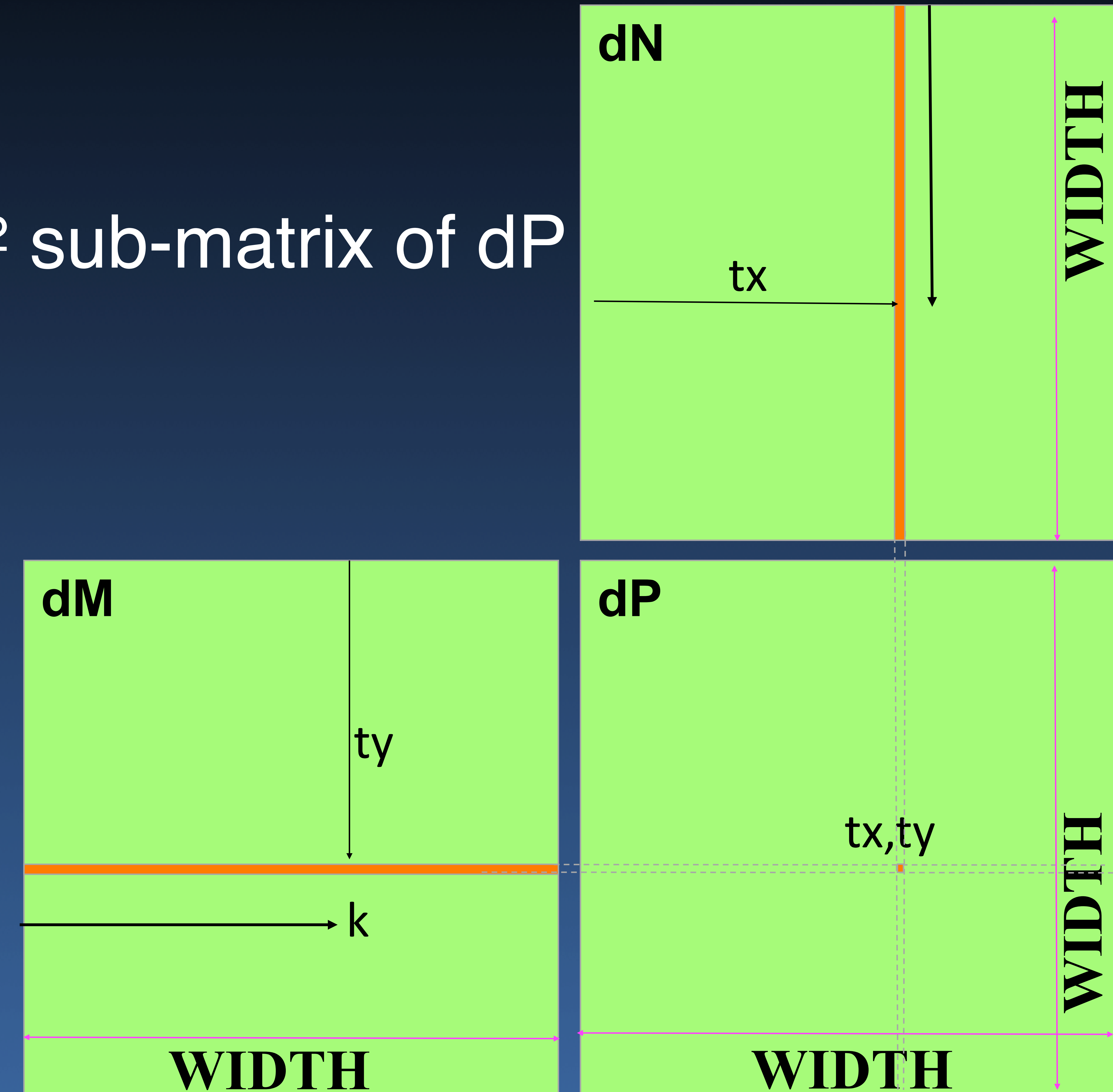
Matrix Data Transfer

```
void MatrixMulOnDevice(float* M, float* N,
                       float* P, int Width)
{
     int size = Width * Width * sizeof(float);
     float *dM, *dN, *dP;
 …
1.  // Allocate and Load M, N to device memory
    cudaMalloc(&dM, size);
    cudaMemcpy(dM, M, size, cudaMemcpyHostToDevice);
    cudaMalloc(&dN, size);
    cudaMemcpy(dN, N, size, cudaMemcpyHostToDevice);
    // Allocate P on the device
    cudaMalloc(&dP, size);
2. // Setup the execution configuration
    dim3 dimBlock(Width, Width);
    // Launch the device computation threads!
    MatrixMulKernel<<<1, dimBlock>>>(dM, dN, dP, Width);
3.   // Read P from the device
    cudaMemcpy(P, dP, size, cudaMemcpyDeviceToHost);
    // Free device matrices
    cudaFree(dM); cudaFree(dN); cudaFree (dP);
}
```

Just one block?

Multiple Thread Blocks

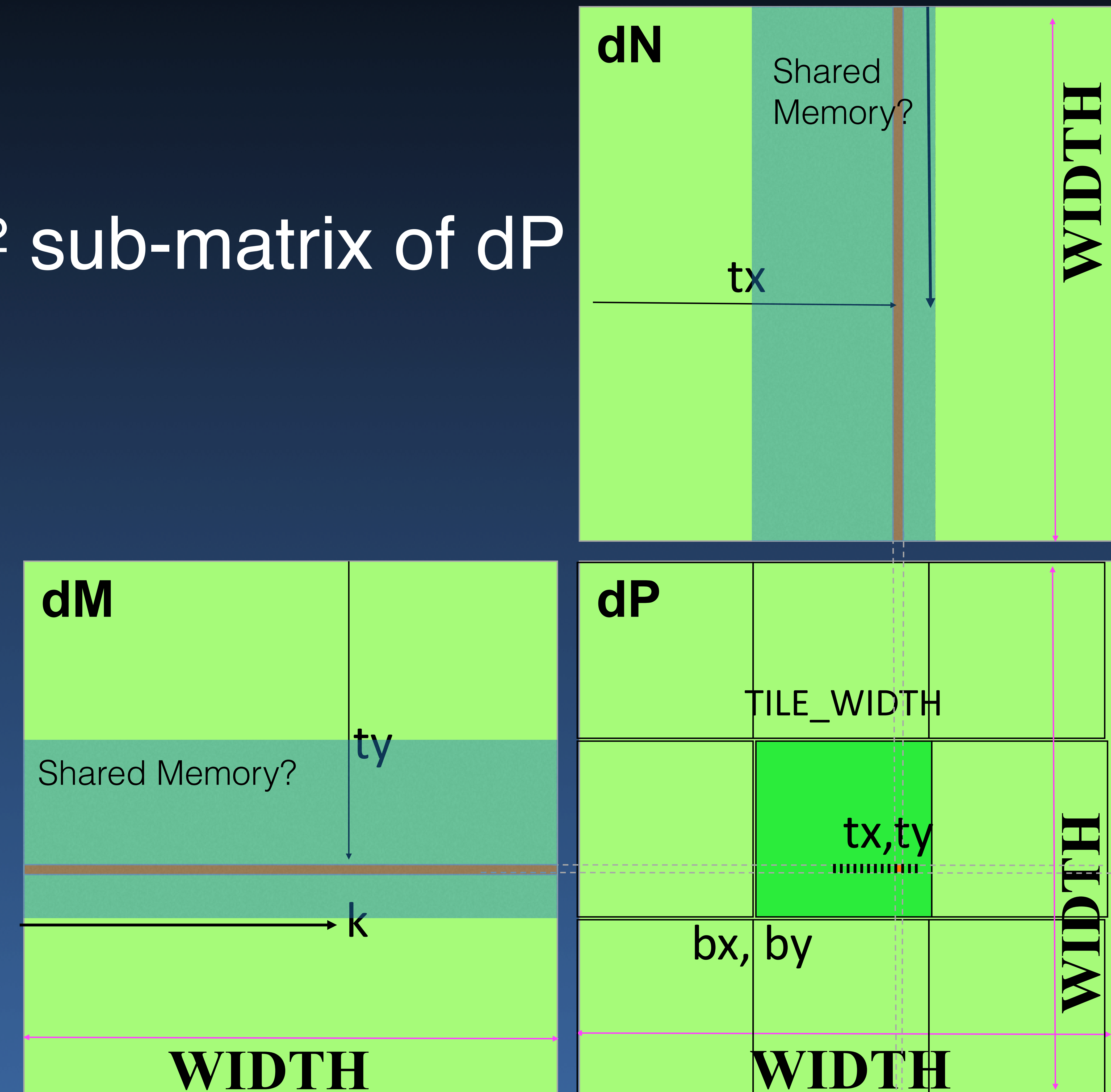
- 2D thread block computes a $(TILE_WIDTH)^2$ sub-matrix of dP
    - $(TILE_WIDTH)^2$ threads/block
- Generate a 2D Grid
- $(WIDTH/TILE_WIDTH)^2$ blocks



Multiple Thread Blocks

- 2D thread block computes a $(\text{TILE_WIDTH})^2$ sub-matrix of dP
  - $(\text{TILE_WIDTH})^2$ threads/block
- Generate a 2D Grid
- $(\text{WIDTH/TILE_WIDTH})^2$ blocks

If more blocks needed than the maximum allowed, break into multiple kernels.



Subodh Kumar

Memory Access Efficiency

- All (active) threads of the warp load/store
- Concurrent accesses by a warp can be coalesced into memory transactions
- Shared memory distributed into banks

➡Non-conflicting access efficient

- Atomic operations are serialized

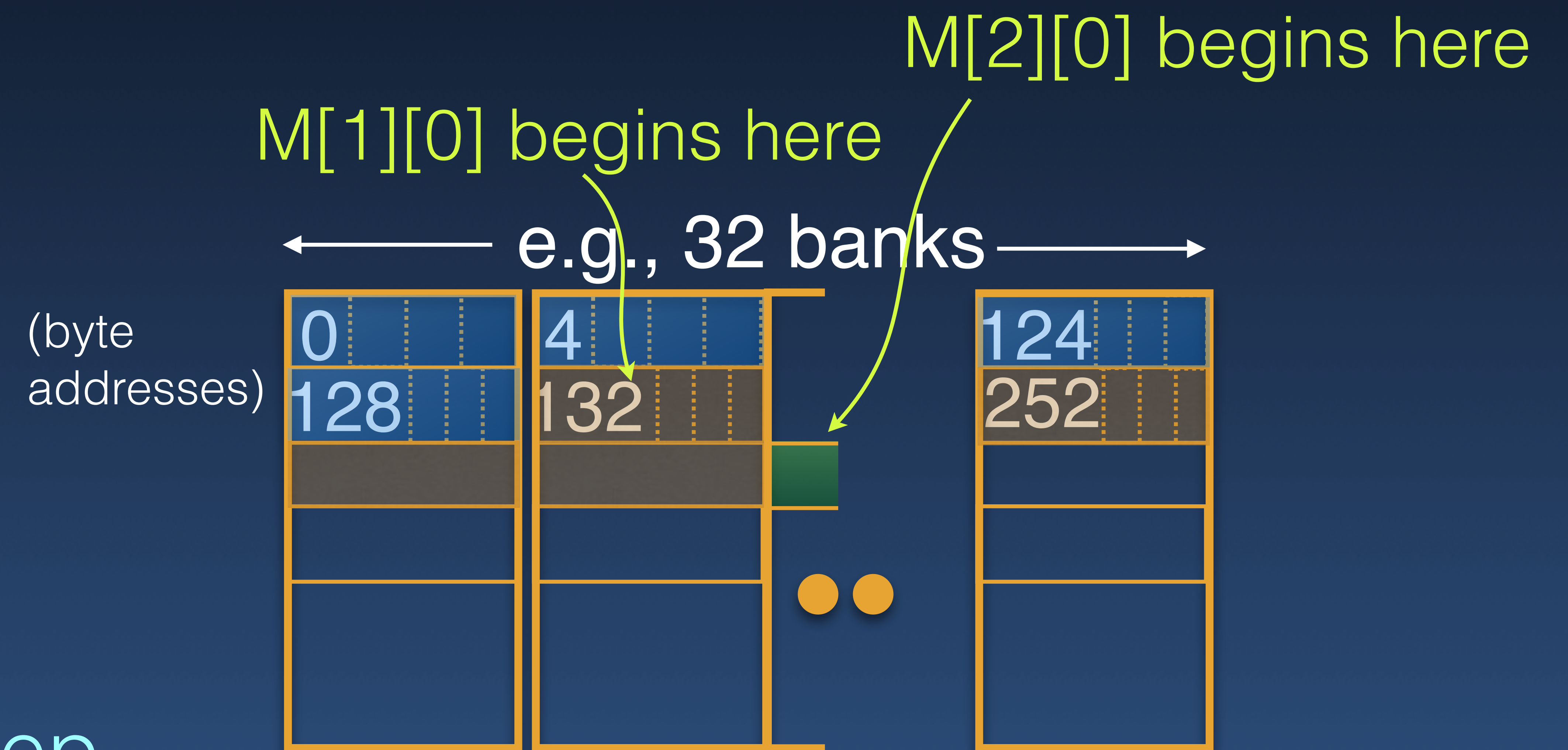
- Multi-ported, word size 4 bytes
  - ➡ Address space interleaved in banks
  - ➡ 32 banks, one word each
- __shared__ float floats[N];
  - ➡ float data = floats[step * tid];
  - ➡ No bank conflict for odd values of step
- __shared__ float M[][33]
  - ➡ Efficient Column-major access

- Multi-ported, word size 4 bytes
  - ➡ Address space interleaved in banks
  - ➡ 32 banks, one word each
- __shared__ float floats[N];
  - ➡ float data = floats[step * tid];
  - ➡ No bank conflict for odd values of step
- __shared__ float M[][33]
  - ➡ Efficient Column-major access

- Multi-ported, word size 4 bytes
  - ➡ Address space interleaved in banks
  - ➡ 32 banks, one word each
- __shared__ float floats[N];
  - ➡ float data = floats[step * tid];
  - ➡ No bank conflict for odd values of step
- __shared__ float M[][33]
  - ➡ Efficient Column-major access

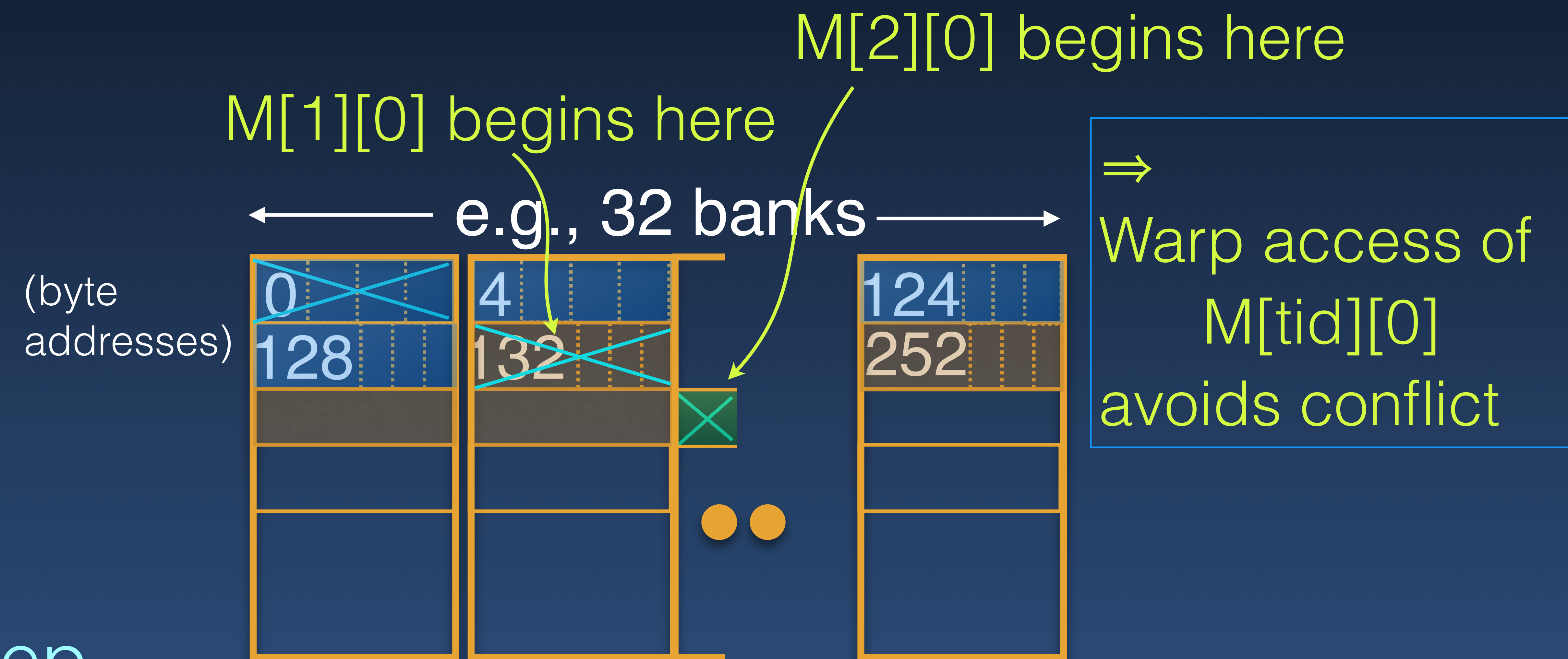
Shared Memory Coalescing

- Multi-ported, word size 4 bytes
  - ➡ Address space interleaved in banks
  - ➡ 32 banks, one word each
- __shared__ float floats[N];
  - ➡ float data = floats[step * tid];
  - ➡ No bank conflict for odd values of step
- __shared__ float M[][33]
  - ➡ Efficient Column-major access

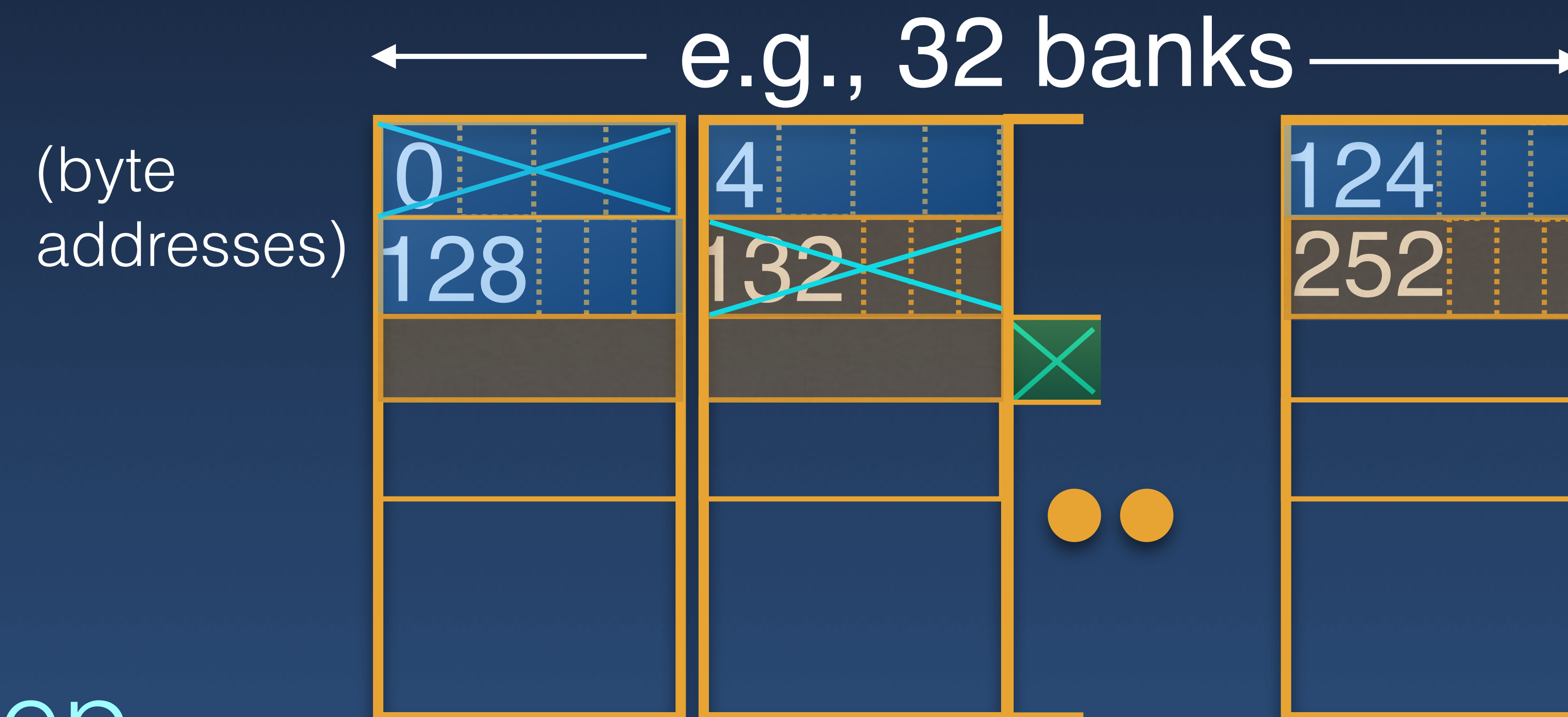


Shared Memory Coalescing

- Multi-ported, word size 4 bytes
  - ➡ Address space interleaved in banks
  - ➡ 32 banks, one word each
- __shared__ float floats[N];
  - ➡ float data = floats[step * tid];
  - ➡ No bank conflict for odd values of step
- __shared__ float M[][33]
  - ➡ Efficient Column-major access



- Multi-ported, word size 4 bytes
    - ➡ Address space interleaved in banks
    - ➡ 32 banks, one word each
- __shared__ float floats[N];
    - ➡ float data = floats[step * tid];
    - ➡ No bank conflict for odd values of step
- __shared__ float M[][33]
    - ➡ Efficient Column-major access



L1 and L2 have standard cache behavior
e.g., 128 byte Cache line for L1
32 byte line size for L2, Global memory
Some cache policy control is possible

Block Synchronization

- void __syncthreads()
  - ➡ block-wide barrier AND
  - ➡ block memory fence (global/shared memory accesses by threads in the block)
- int __syncthreads_count(int predicate)
  - ➡ returns the count of threads where predicate != 0
- int __syncthreads_and(int predicate)
  - ➡ returns non-zero iff predicate != 0 for all threads
- int __syncthreads_or(int predicate)
  - ➡ returns non-zero iff predicate != 0 for any threads

- void __syncthreads()
  - ➡ block-wide barrier AND
  - ➡ b
- int _
  - ➡ re
- int _
  - ➡ re
- int _
  - ➡ returns non-zero iff predicate != 0 for any threads

```
// Read into Shared Mem and operate
__device__ void sortShared() ..
__global__ static void ParallelSort(int * values)
{
    extern __shared__ int shared[];
    const unsigned int tid = threadIdx.x;

    shared[tid] = values[tid]; // Copy input to shared mem
    __syncthreads();
    sortShared();
  values[tid] = shared[tid]; // Write result
}
```

Intra Warp Primitives

- void __syncwarp(unsigned mask=FULL_MASK)
  - ➡ Warp-level barrier, Only active threads need to barrier
- int __all_sync(unsigned mask, int predicate)
  - ➡ Return non-0 iff predicate != 0 for all active threads in mask.
    - int __any_sync(unsigned mask, int predicate)
- T __shfl_sync(unsigned mask, T var, int src)
  - ➡ Return var specified by src
- uint __ballot_sync(unsigned mask, int predicate)
  - ➡ Return has i^{th} bit set iff predicate is true for the ith thread of the warp
- uint __activemask()
  - ➡ Return bit mask with 1 for active threads

Warp-collectives for active threads and common mask

Intra Warp Primitives

- void __syncwarp(unsigned mask=FULL_MASK)
  - ➡ Warp-level barrier, Only active threads need to barrier
- int __all_sync(unsigned mask, int predicate)
  - ➡ Return non-0 iff predicate != 0 for all active threads in mask.
    - int __any_sync(unsigned mask, int predicate)
- T __shfl_sync(unsigned mask, T var, int src)
  - ➡ Return v
- uint __ba
  - ➡ Return has i^{th} bit set iff predicate is true for the ith thread of the warp
- uint __activemask()
  - ➡ Return bit mask with 1 for active threads

Warp-collectives for active threads and common mask

```
// Reduce mySum
for (int offset = warpSize/2; offset > 0; offset /= 2)
      mySum += __shfl_down_sync(FULL_MASK, mySum, offset);
```

Subodh Kumar

Atomic Operations

- Read-modify-write on one 32-bit word
  - ➡ Also 64-bit word in newer architecture
- Block-wide atomics: All threads within the same block
  - ➡ e.g., atomicAdd_block
- Device-wide atomics: All GPU threads
  - ➡ e.g., atomicAdd, atomicSub, atomicMin, atomicExchg
- System-wide atomics: CPU+GPU threads
  - ➡ e.g., atomicAdd_system
- More generic Compare and Swap
  - ➡ atomicCAS()

Need corresponding CPU functions, e.g., __sync_fetch_and_add

- __threadfence_block
  - ➡ wait until all global/shared memory accesses by the caller are visible to block
- __threadfence
  - ➡ wait until all memory accesses by the caller are visible to
    - All threads in the device for global memory accesses. Also, all threads in the thread block for shared memory accesses
- __threadfence_system
  - ➡ wait until all memory accesses by the caller are visible to:
    - All threads in the thread block for shared memory accesses, all threads in the device for global memory accesses, and host threads for page-locked host memory accesses