PDC Summer School 2013

Introduction to CUDA
Lecture 1

2013-08-23

Michael Schliephake

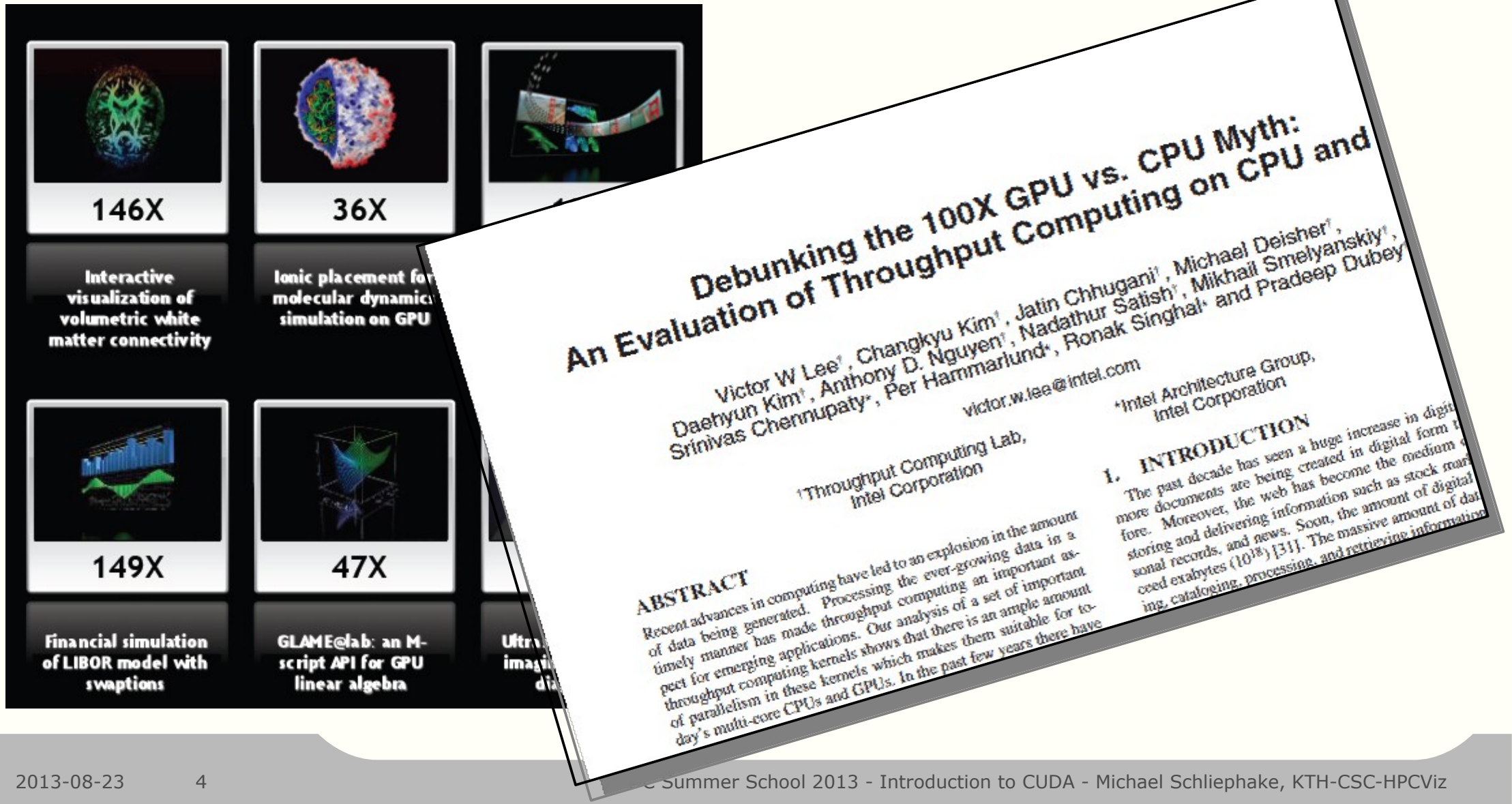
KTH – CSC – HPCViz

Disposition

1. Introduction
2. CUDA Threads
3. CUDA Memories
4. Matrix Multiplication

1. Introduction

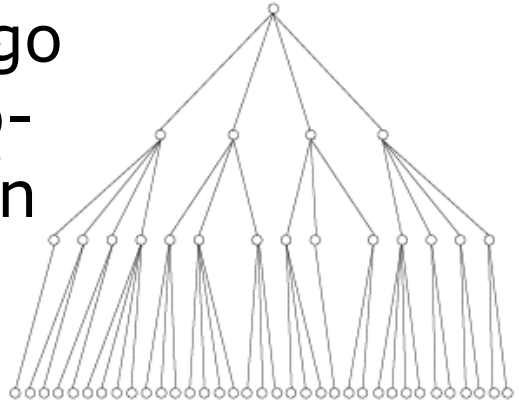
The Debate GPU vs. CPU

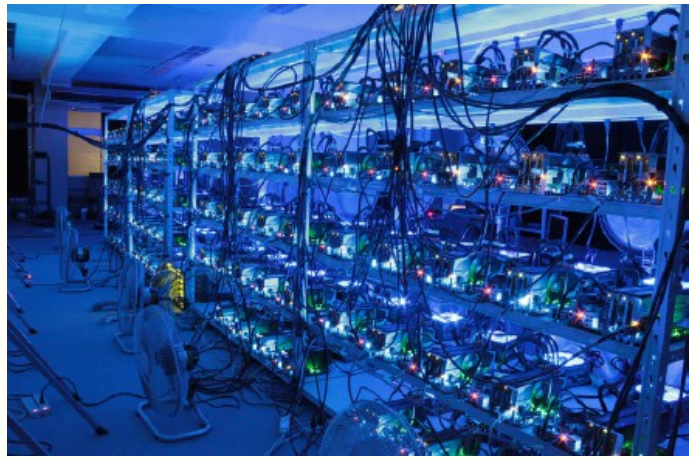


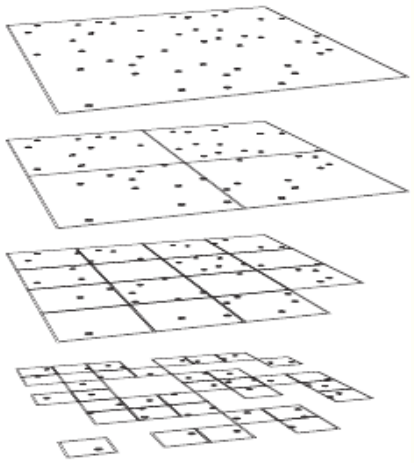
Example: 190 TFlops astrophysical n-body simulation

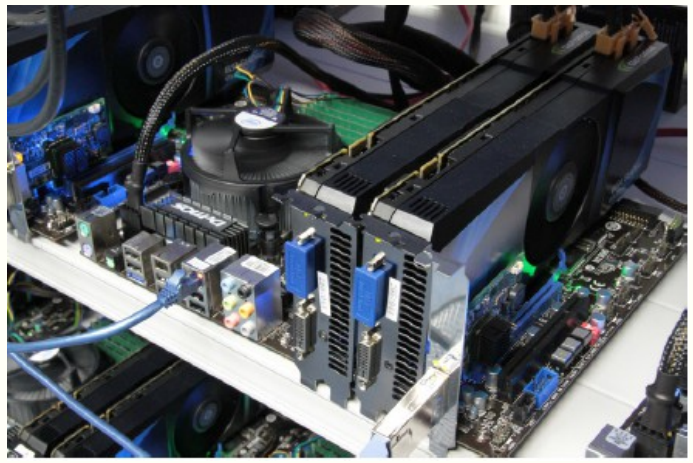
Tsuyoshi Hamada and Keigo Nitadori: 190 TFlops Astro-physical N-body Simulation on cluster of GPUs.

- Univ. of Nagasaki, RIKEN
- DEGIMA cluster: 144 compute nodes with 576 GPU (GT200)
- 3.278.982.596 particles
- 190 TFlops sustained performance
- Barnes-Hut treecode for GPU developed





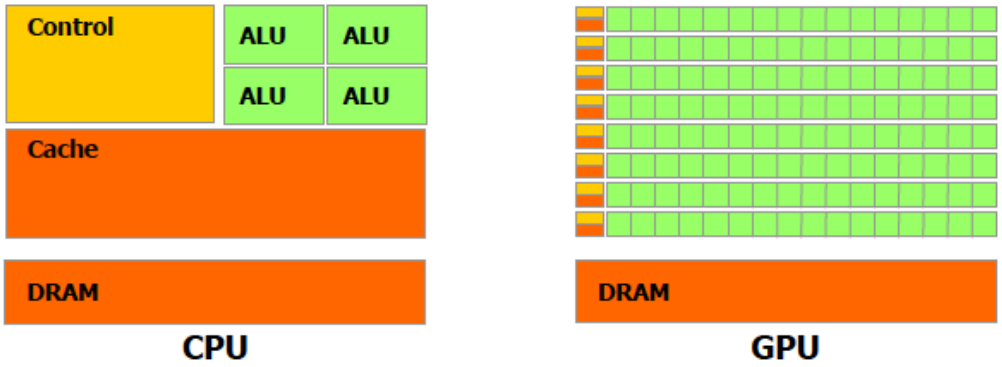


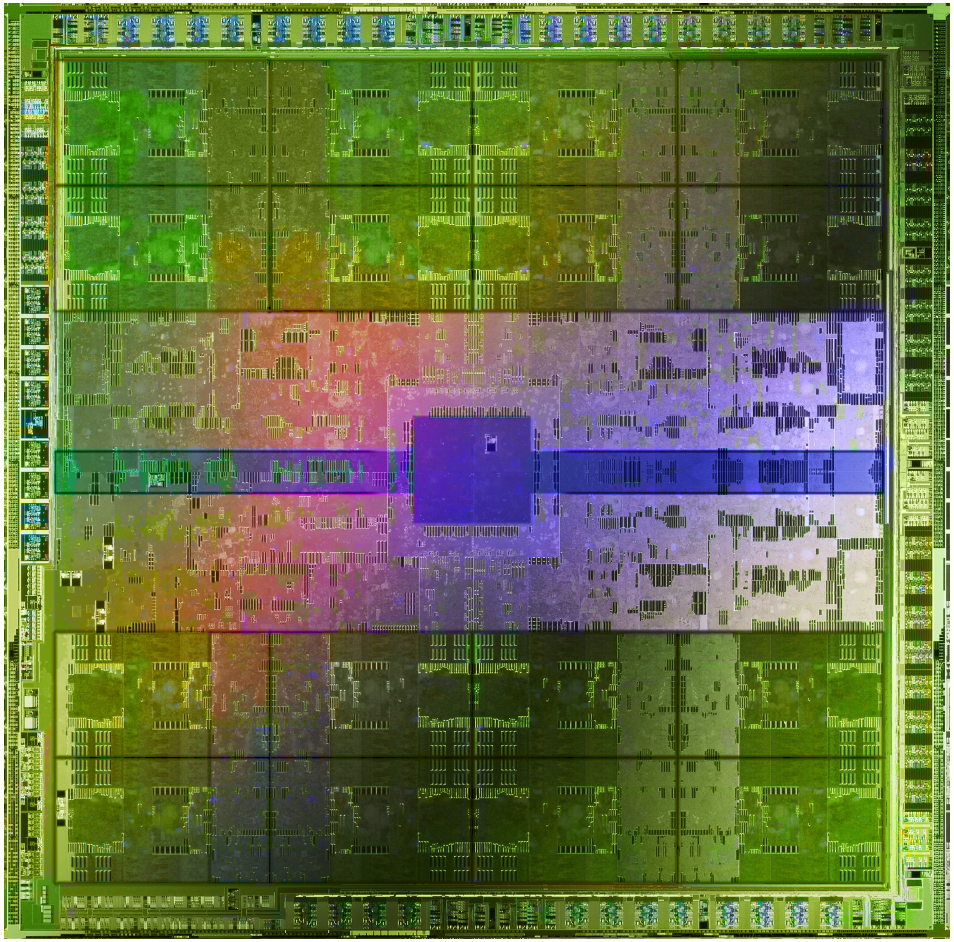


Information and images: SC10, http://dx.doi.org/10.1109/SC.2010.1

GPU processor characteristics

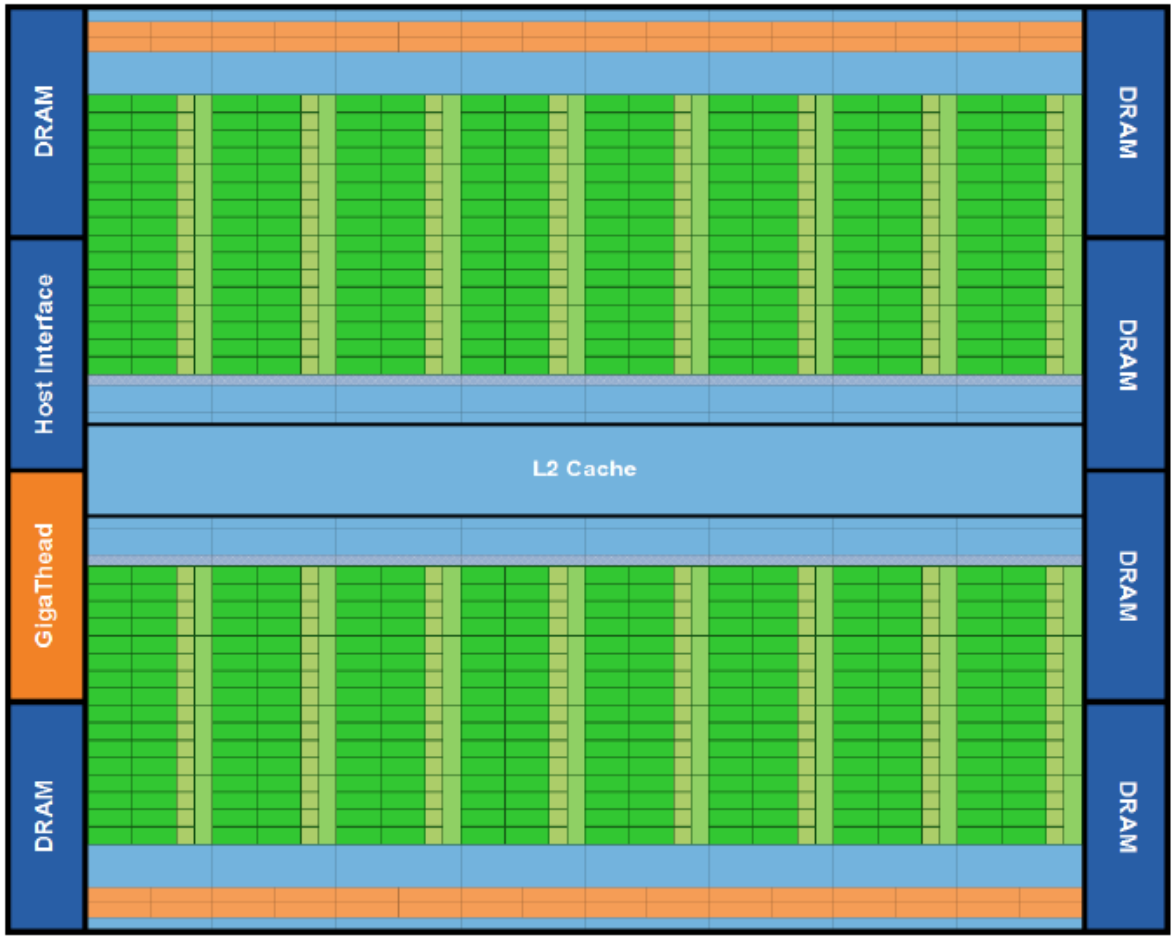
- More transistors for compute cores
- Many, simple cores
- Small caches
- Many memory controllers for high bandwidth

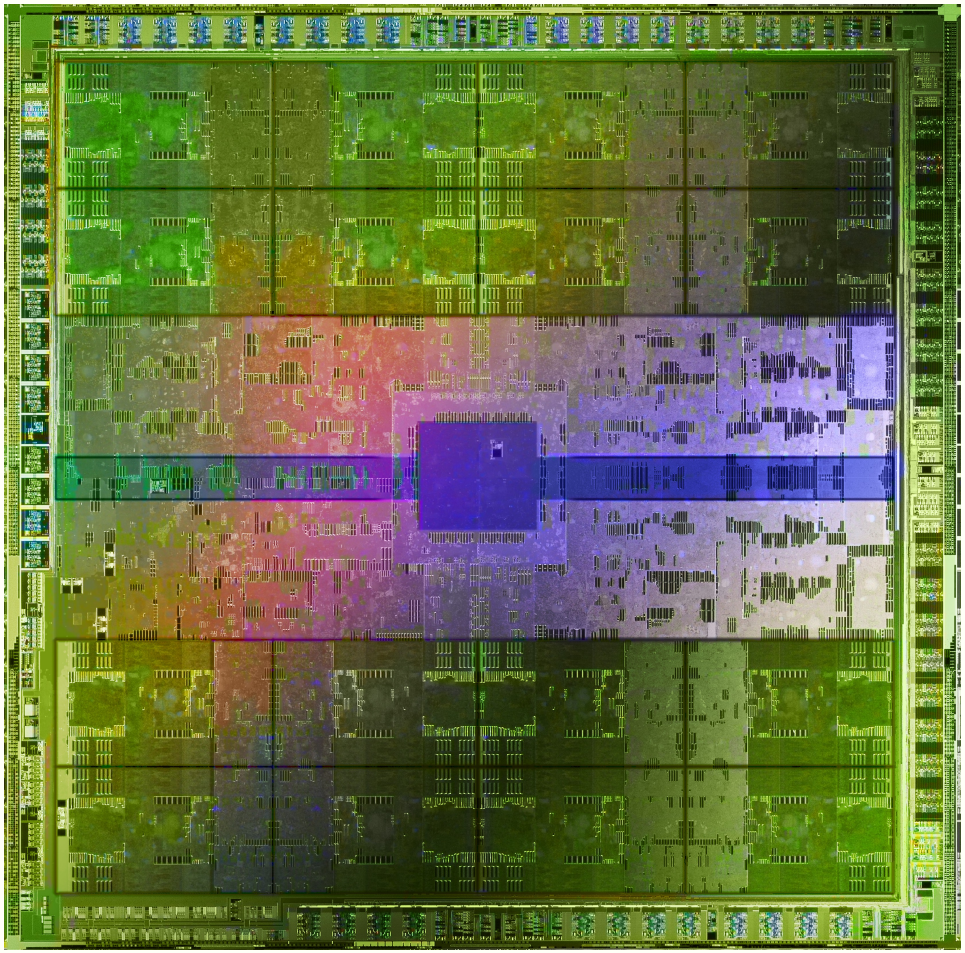




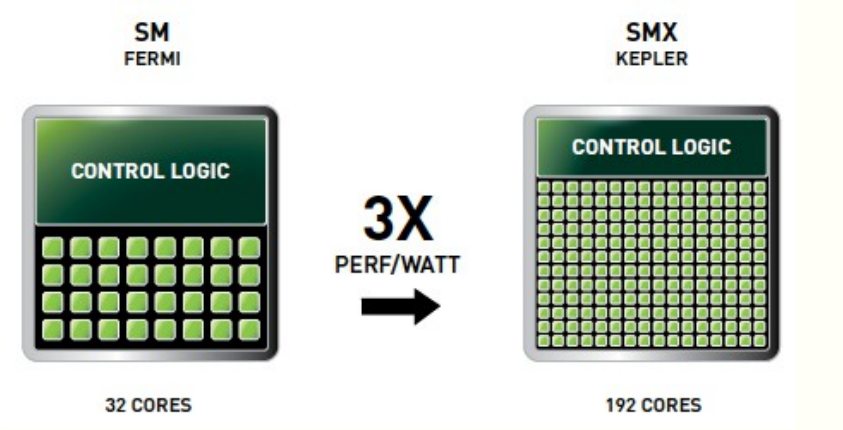
Images: Courtesy of NVIDIA and David Kirk/NVIDIA and Wen-mei W. Hwu

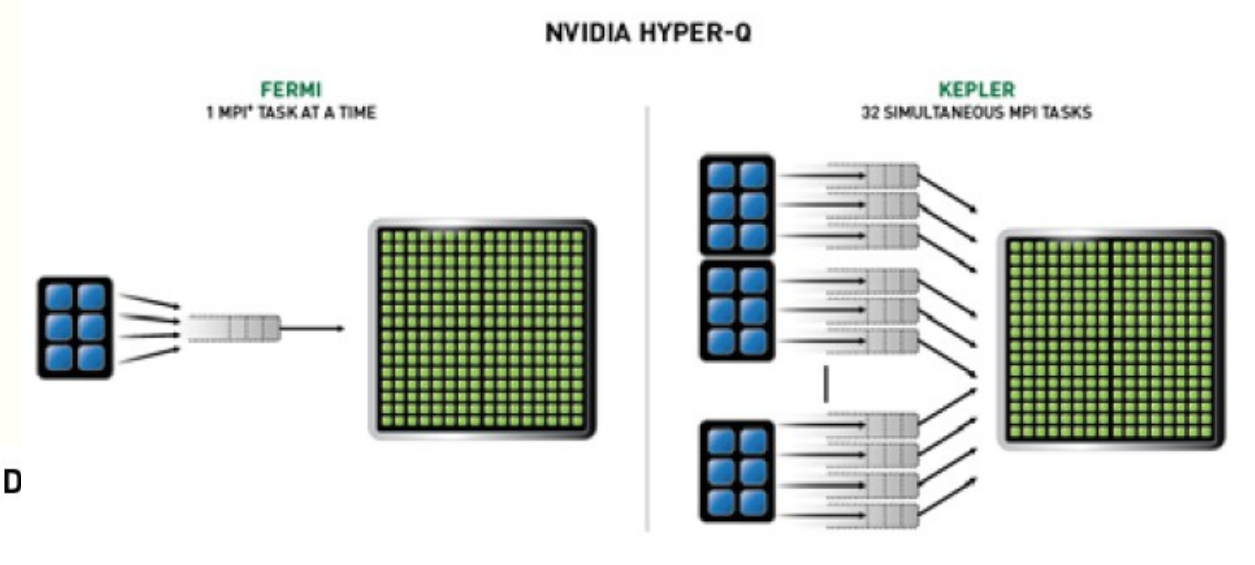
Fermi Processor



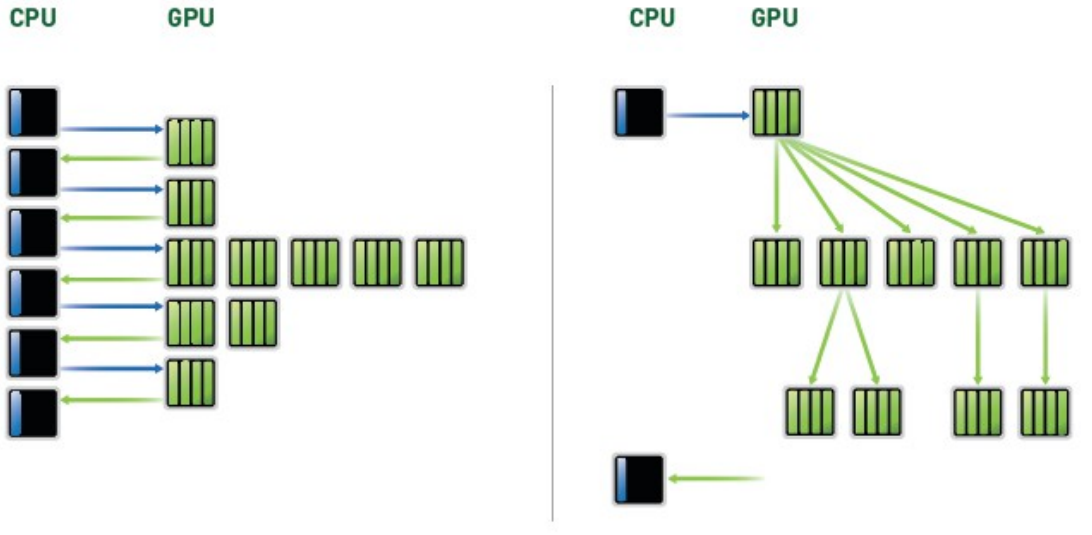


Outlook Kepler Architecture I





D

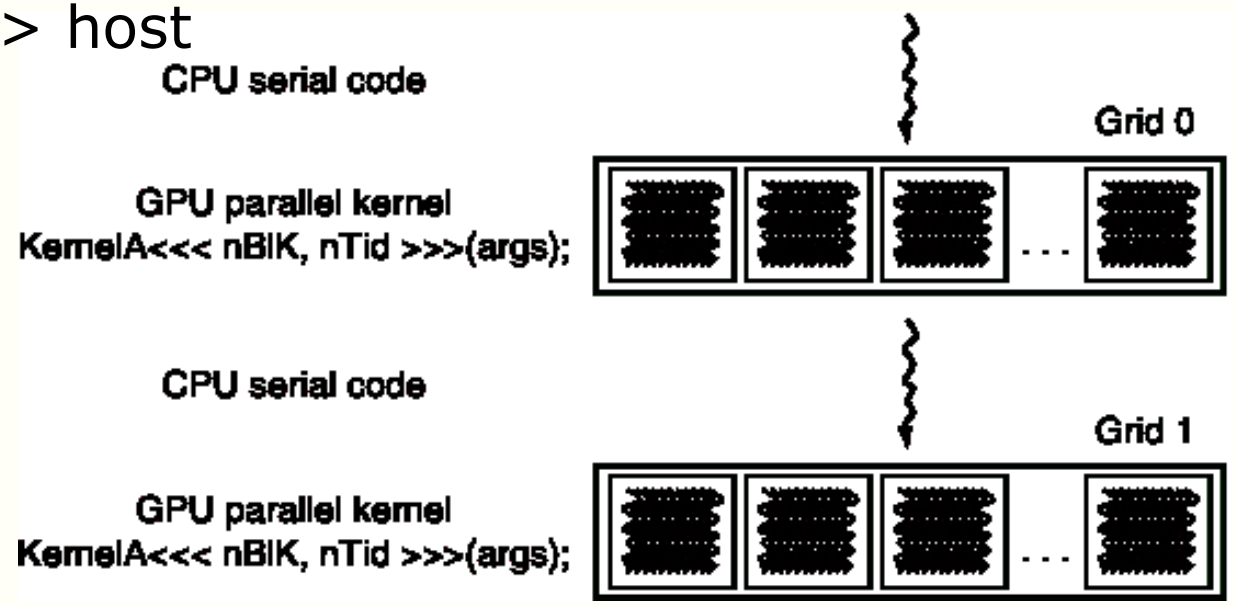


Images: Courtesy of NVIDIA

2. CUDA Threads

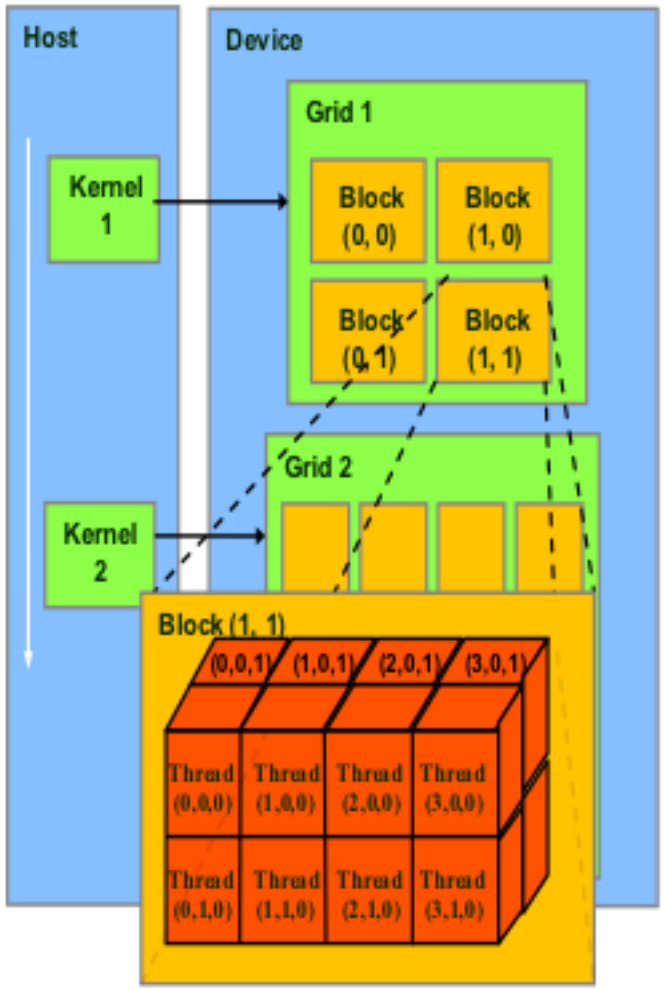
Program Execution Scheme

1. Initialization
2. Alloacate host- and device-memory
3. While „Something to do“
    1. Transfer data host -> device
    2. Execute computational kernels on the device
    3. Transfer data device -> host
4. Deallocate memory
5. End.



Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

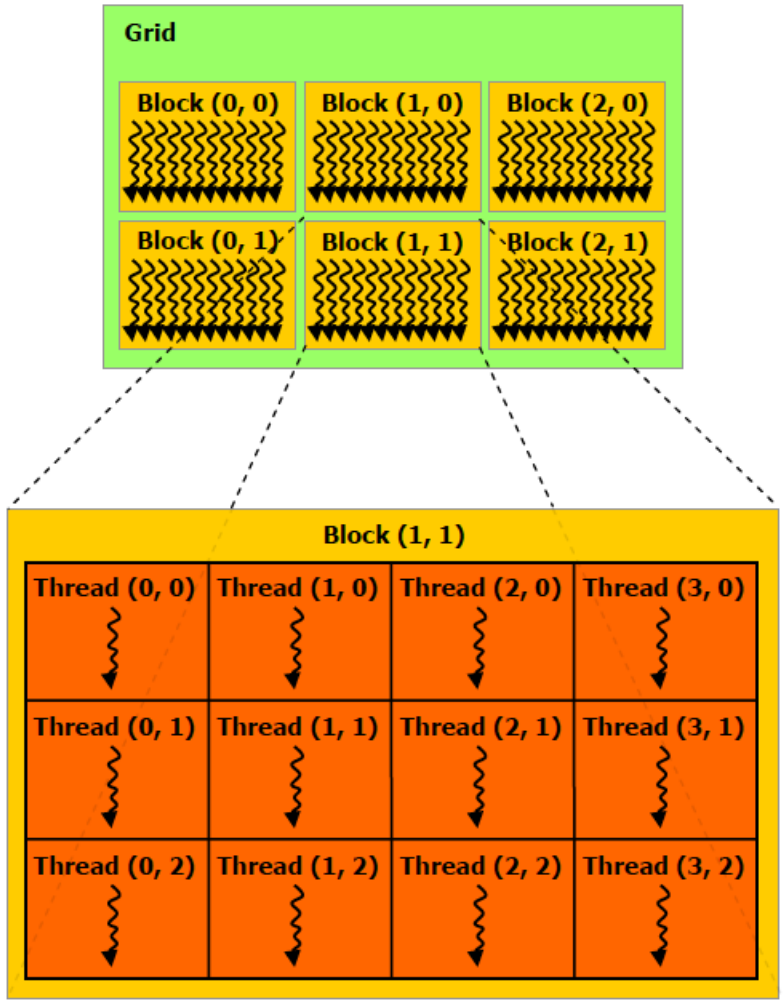
Grids, Blocks, Threads



Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

Threads and Blocks

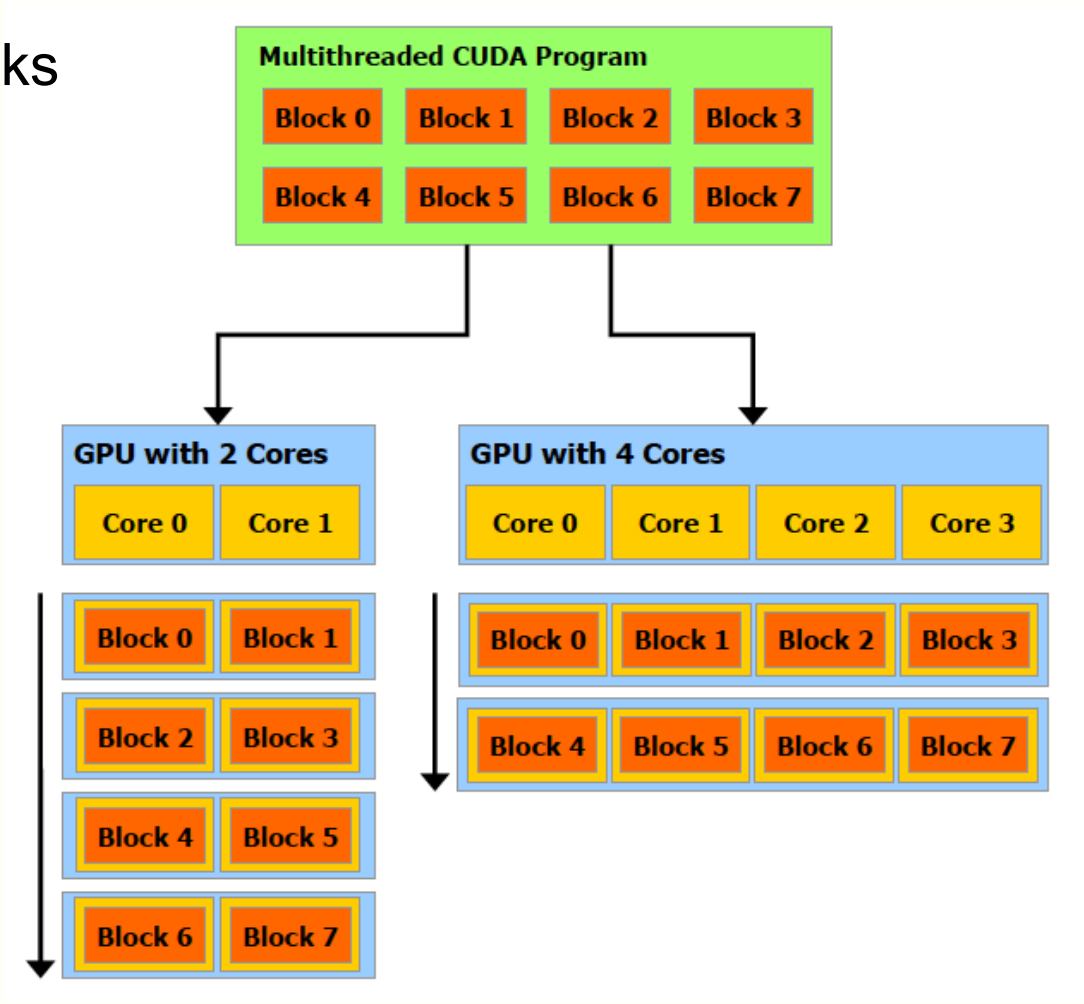
- Thread
  - Independent thread of execution
  - Real thread (on GPU), virtual (on host)
- Block
  - correspond to parallel tasks
  - must be independent
    - Execution order not determined, sequential or parallel
    - Coordination possible



Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

Scalable Execution

Independence of blocks guarantees scalable program execution



Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

CUDA
Compute Unified Device Architecture

- C extensions
  - FORTRAN support mit PGI compiler
  - C++ support improved in Fermi
- Parts with CUDA code must be compiled by „nvcc“
  - Calls other compilers like gcc
  - Links the runtime

CUDA
Compute Unified Device Architecture

- C extensions
  - FORTRAN support mit PGI compiler
  - C++ support improved in Fermi
- Parts with CUDA code must be compiled by „nvcc“
  - Calls other compilers like gcc
  - Links the runtime

CUDA Hello World

```
__global__ void kernel( void ) {
}

int main( int argc, char *argv[] ) {
   kernel<<<1,1>>>();
   printf("Hello, world!\n");
   return 0;
}
```

Function Modifier

	Execution	Callable from
__device__ float deviceFunc()	device	device
__global__ void kernelFunc()	device	host
__host__ float hostFunc()	host	host

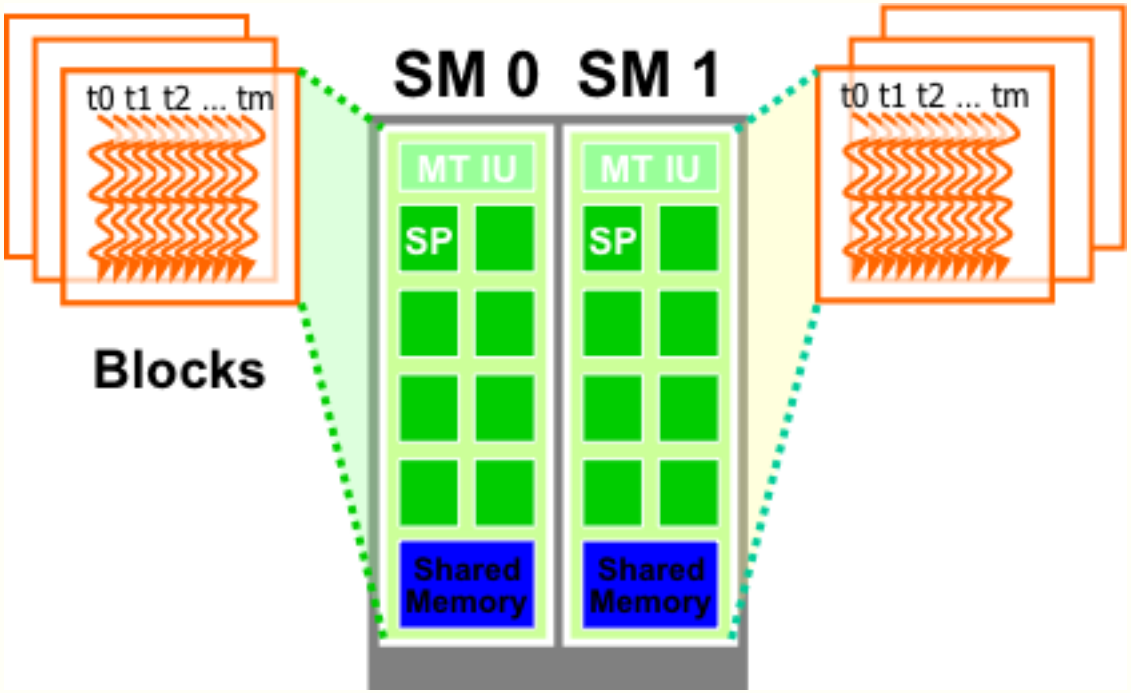
Specification of Host and Device Code

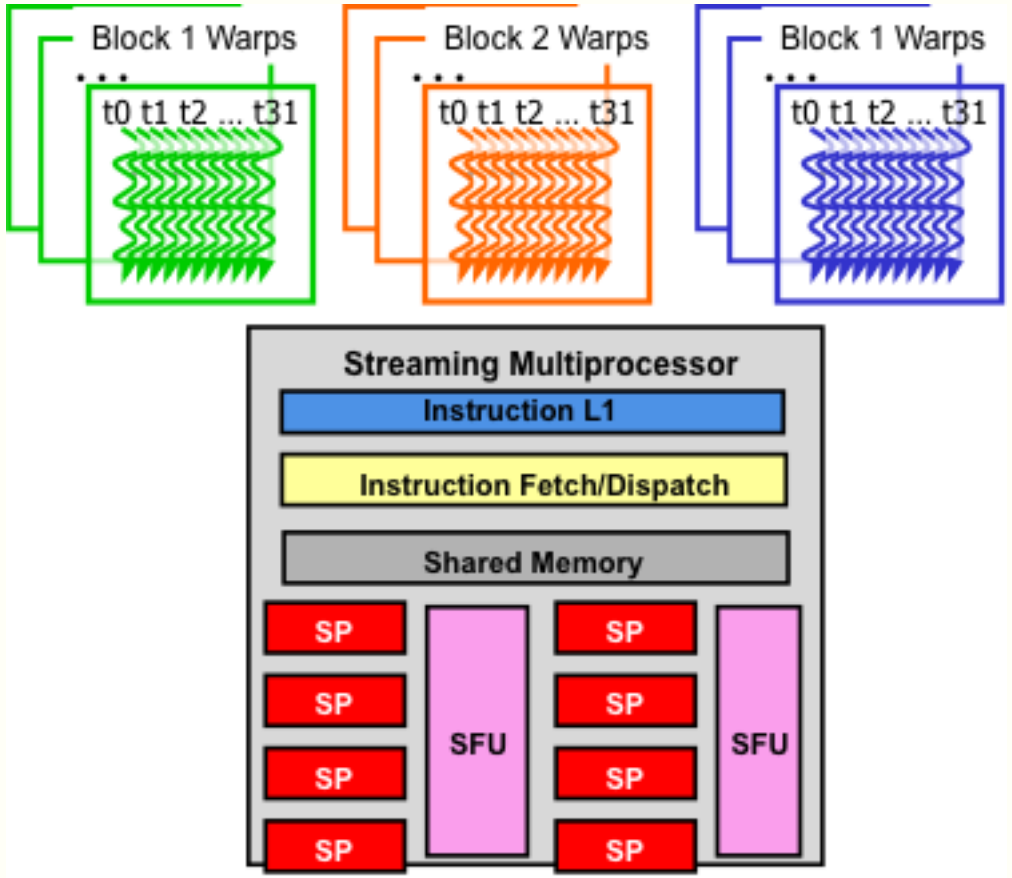
- Execution Kernel

```
function_name<<<nBlock, nThread>>>(args);
```

  - `nBlock` – number of blocks
  - `nThread` – number of threads
  - args – pointer to data etc.
- Thread learns its position via
  - `blockIdx, threadIdx` – global variables for current Position
  - `gridDim, blockDim` – global variables for extensions of the indices

Warp Scheduling





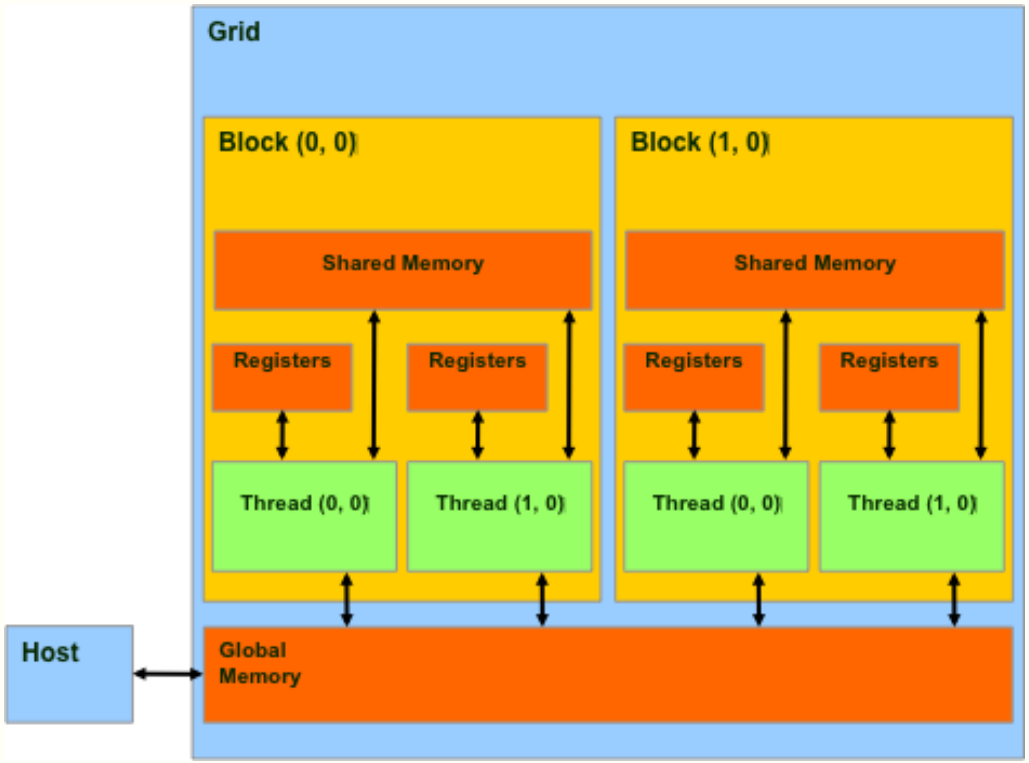

```
void __syncthreads();
```

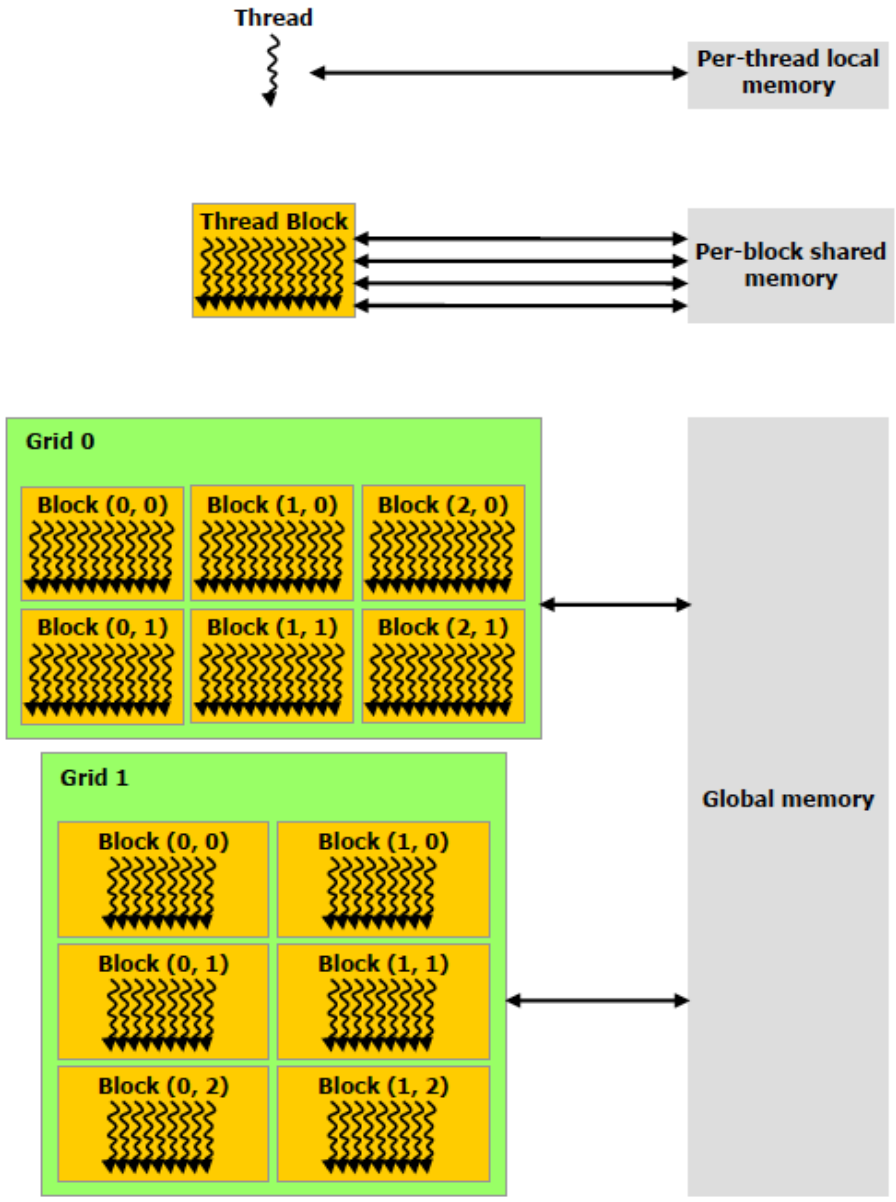
Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

3. CUDA Memories

Memory access

Simplified View to the execution of a block

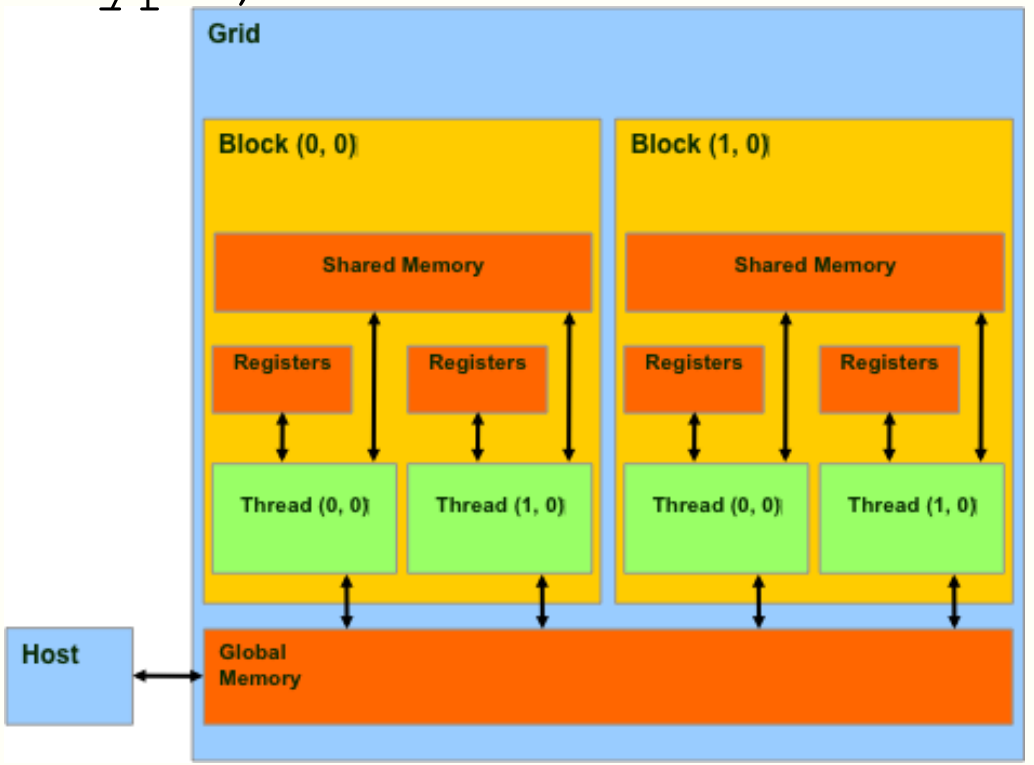




Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

Memory Allocation and Transfer

- `cudaMalloc(pointer, size)`
- `cudaFree(pointer)`
- `cudaMemcpy(dest, src, type)`
  - Host -> Host
  - Host -> Device
  - Device -> Host
  - Device -> Device



ImageCourtesy of: David Kirk/NVIDIA and Wen-mei W. Hwu

Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

Declaration of Variables

Declaration	Memory	Scope	Lifetime
__device__ __local__ int LocalVar	local	thread	thread
__device__ __shared__ int SharedVar	shared	block	block
__device__ int GlobalVar	global	grid	application
__device__ __constant__ int ConstantVar	constant	grid	application

- Automatic Variables
  - In register
  - Arrays: local memory

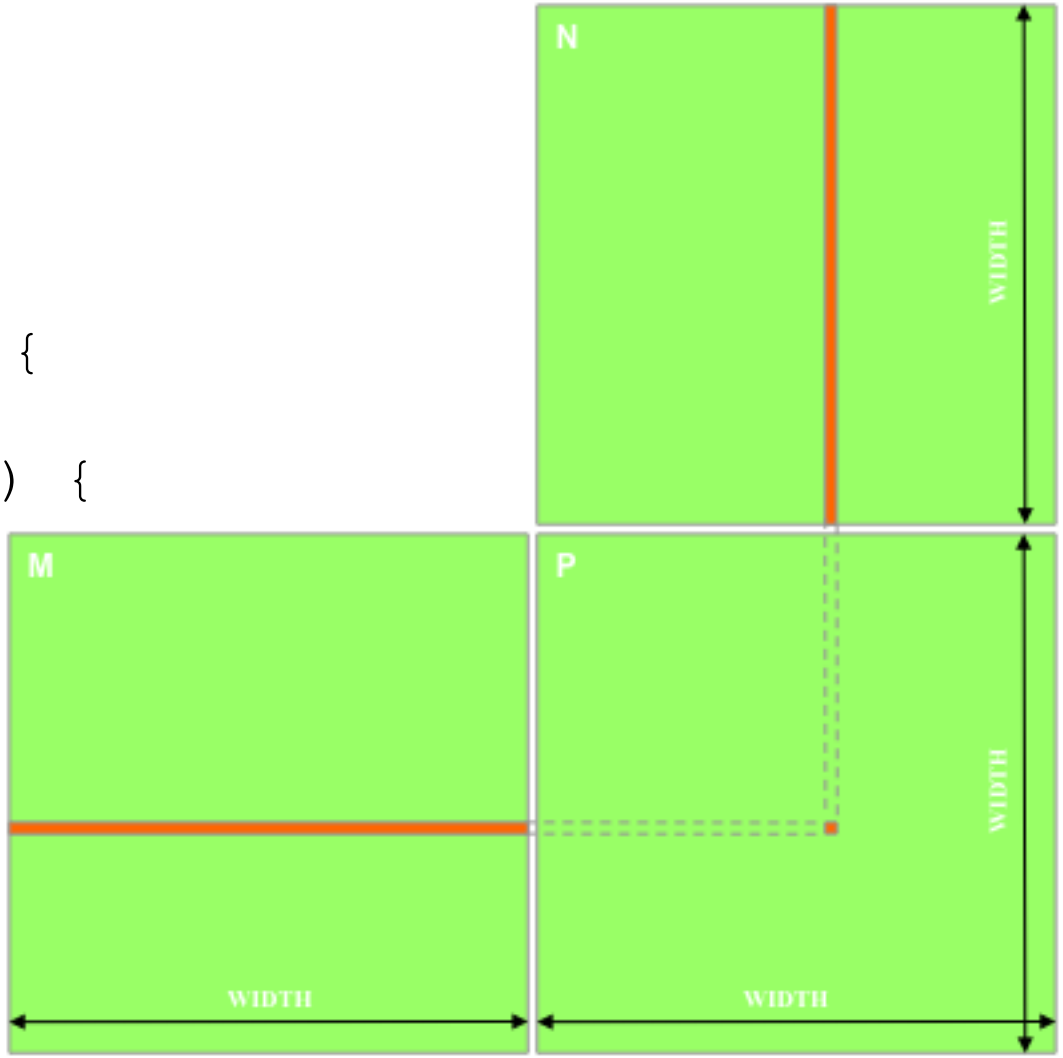
Memory characteristics

- Global memory slow
- Shared memory advantageous for computation
- Constant memory – global memory but very efficient for read-only use
- Texture memory can be used too

4. Matrix Multiplication

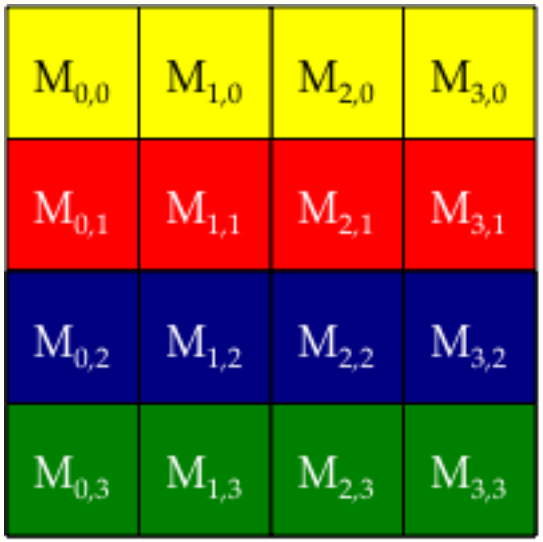
Matrix multiplication
Base algorithm

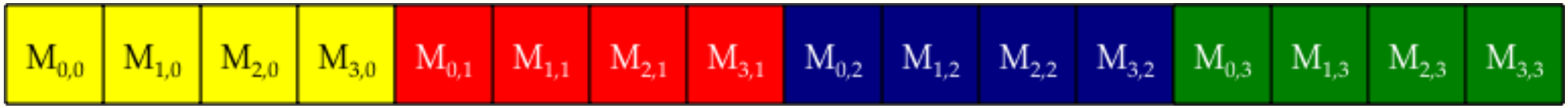
```
float M[WIDTH][WIDTH];
float N[WIDTH][WIDTH];
float P[WIDTH][WIDTH];

for (int i = 0; i < WIDTH; ++i)
  for (int j = 0; j < WIDTH; ++j) {
    float sum = 0.;
    for (int k = 0; k < WIDTH; ++k) {
      float a = M[i*WIDTH + k];
      float b = N[k*WIDTH + j];
      sum += a * b;
    }
    P[i * Width + j] = sum;
  }
}
```



Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

Matrix multiplication
Memory Layout of Matrices





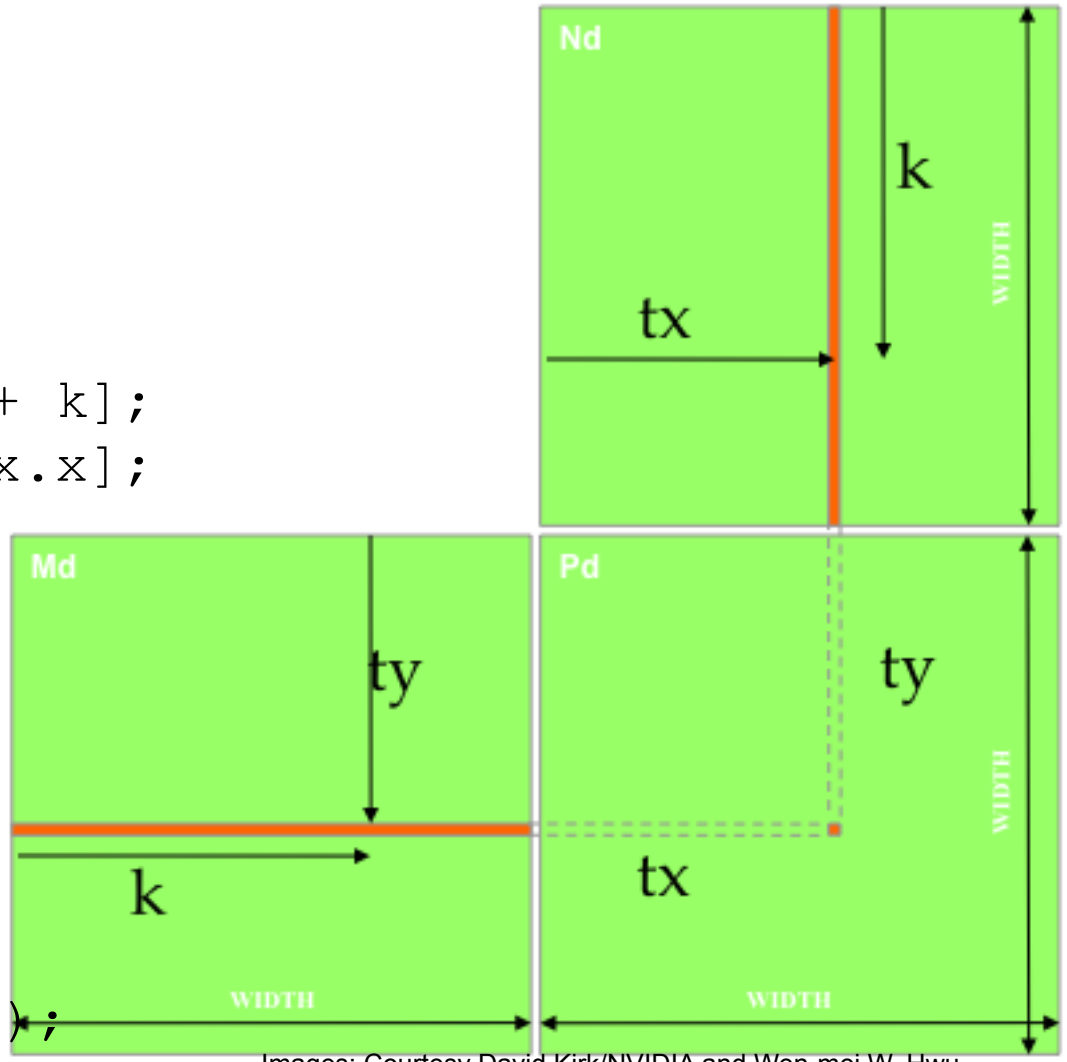
Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

Matrix multiplication
Compute Kernel Fragment

```
float Md[WIDTH*WIDTH];
float Nd[WIDTH*WIDTH];
float Pd[WIDTH*WIDTH];

float sum = 0.;
for (int k = 0; k < WIDTH; ++k) {
  float a = Md[threadIdx.y*WIDTH + k];
  float b = Nd[k*WIDTH + threadIdx.x];
  sum += a * b;
}
Pd[threadIdx.y*WIDTH
      + threadIdx.x] = sum;
```

Kernel activation

```
dim3 dGrid(1,1),
      dBlock(WIDTH, WIDTH);
kernelfunc<<<dGrid, dBlock>>>(...);
```



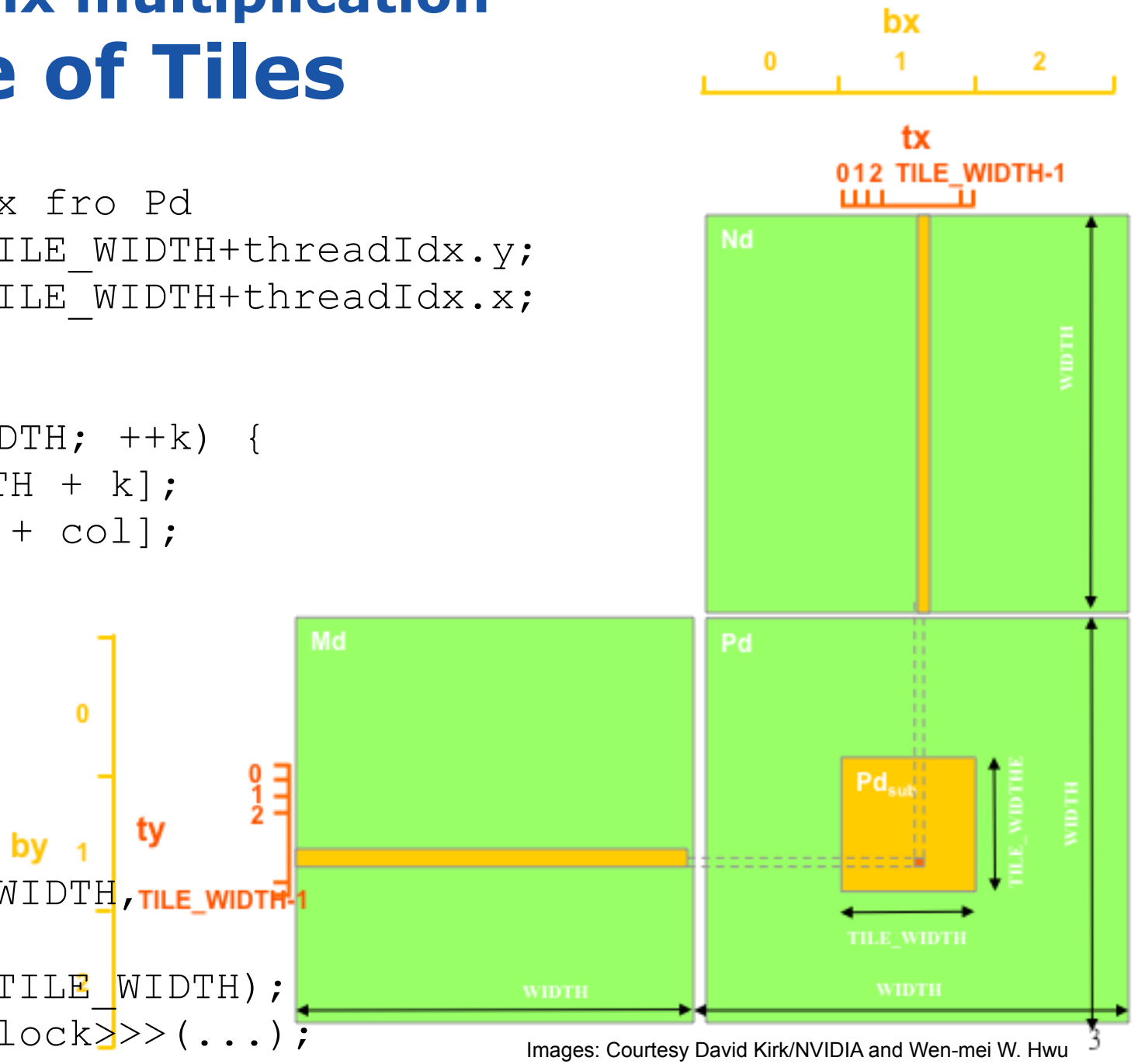
Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

Matrix multiplication
Use of Tiles

```
// Row and column index fro Pd
int row = blockIdx.y*TILE_WIDTH+threadIdx.y;
int col = blockIdx.x*TILE_WIDTH+threadIdx.x;

float sum = 0.;
for (int k = 0; k < WIDTH; ++k) {
  float a = Md[row*WIDTH + k];
  float b = Nd[k*WIDTH + col];
  sum += a * b;
}
Pd[row*WIDTH
    + col] = sum;
```

Kernel activation

```
dim3 dGrid(WIDTH/TILE_WIDTH,
   WIDTH/TILE_WIDTH),
   dBlock(TILE_WIDTH, TILE_WIDTH);
kernelfunc<<<dGrid, dBlock>>>(...);
```



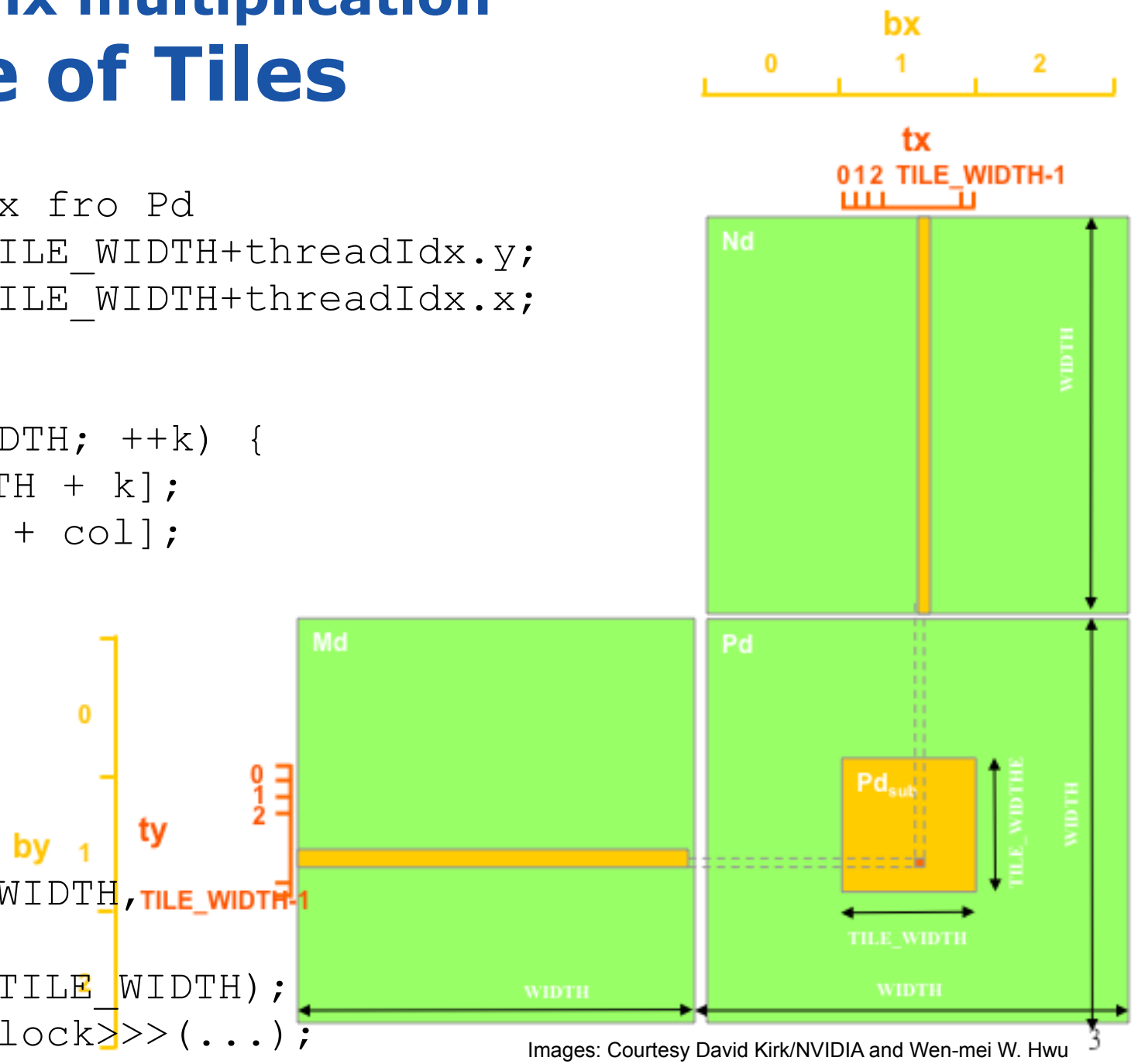
Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

Matrix multiplication
Use of Tiles

```
// Row and column index fro Pd
int row = blockIdx.y*TILE_WIDTH+threadIdx.y;
int col = blockIdx.x*TILE_WIDTH+threadIdx.x;

float sum = 0.;
for (int k = 0; k < WIDTH; ++k) {
  float a = Md[row*WIDTH + k];
  float b = Nd[k*WIDTH + col];
  sum += a * b;
}
Pd[row*WIDTH
    + col] = sum;
```

<u>Kernel activation</u>

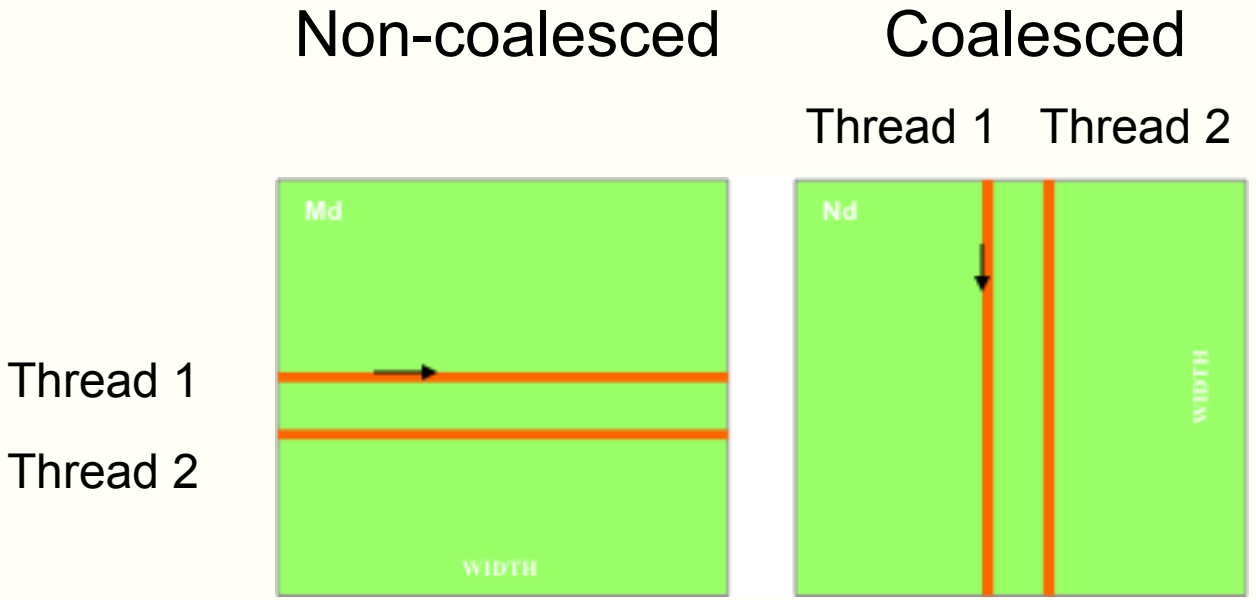
```
dim3 dGrid(WIDTH/TILE_WIDTH,
   WIDTH/TILE_WIDTH),
   dBlock(TILE_WIDTH, TILE_WIDTH);
kernelfunc<<<dGrid, dBlock>>>(...);
```



Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

Memory Coalescing

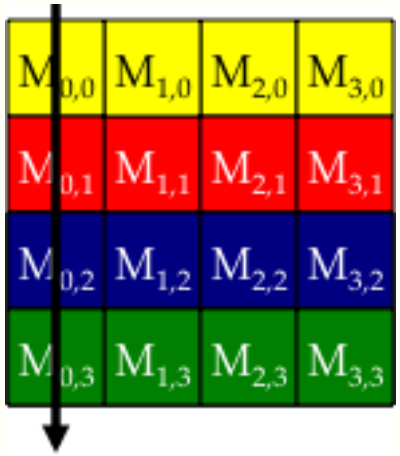
- Accessign consecutive memory locations in a warp increases performance



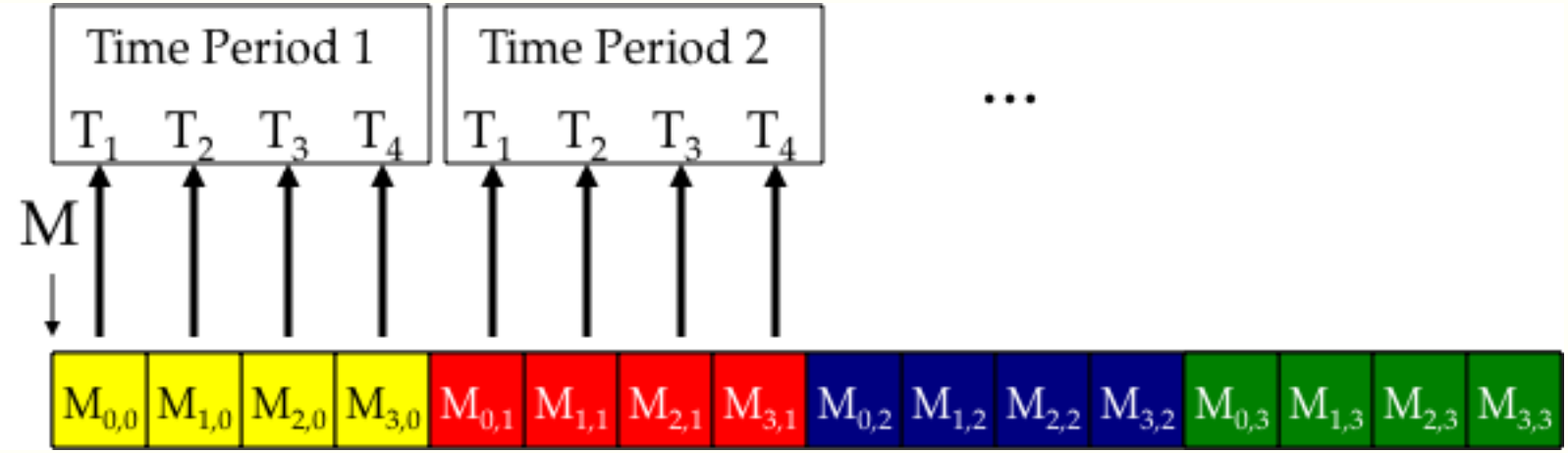
Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

Memory Coalescing II

Access order in the algorithm



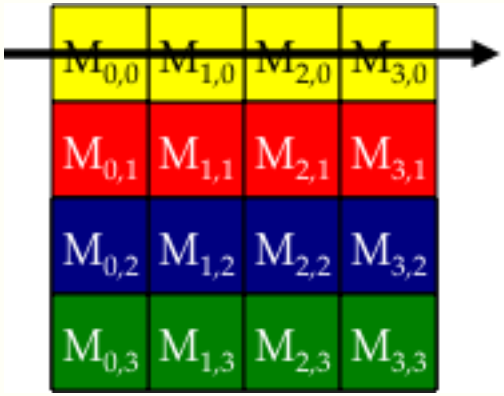
Device memory access



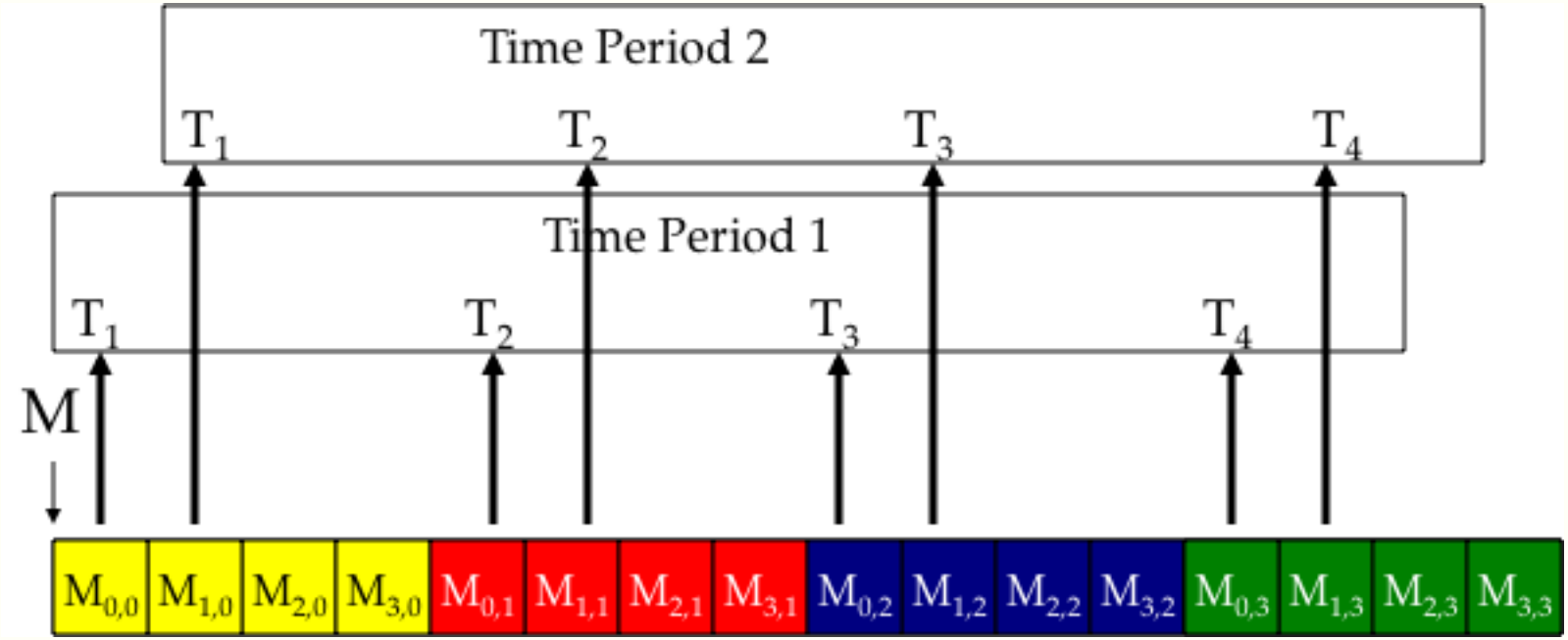
Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

Memory Coalescing III

Access order in the algorithm

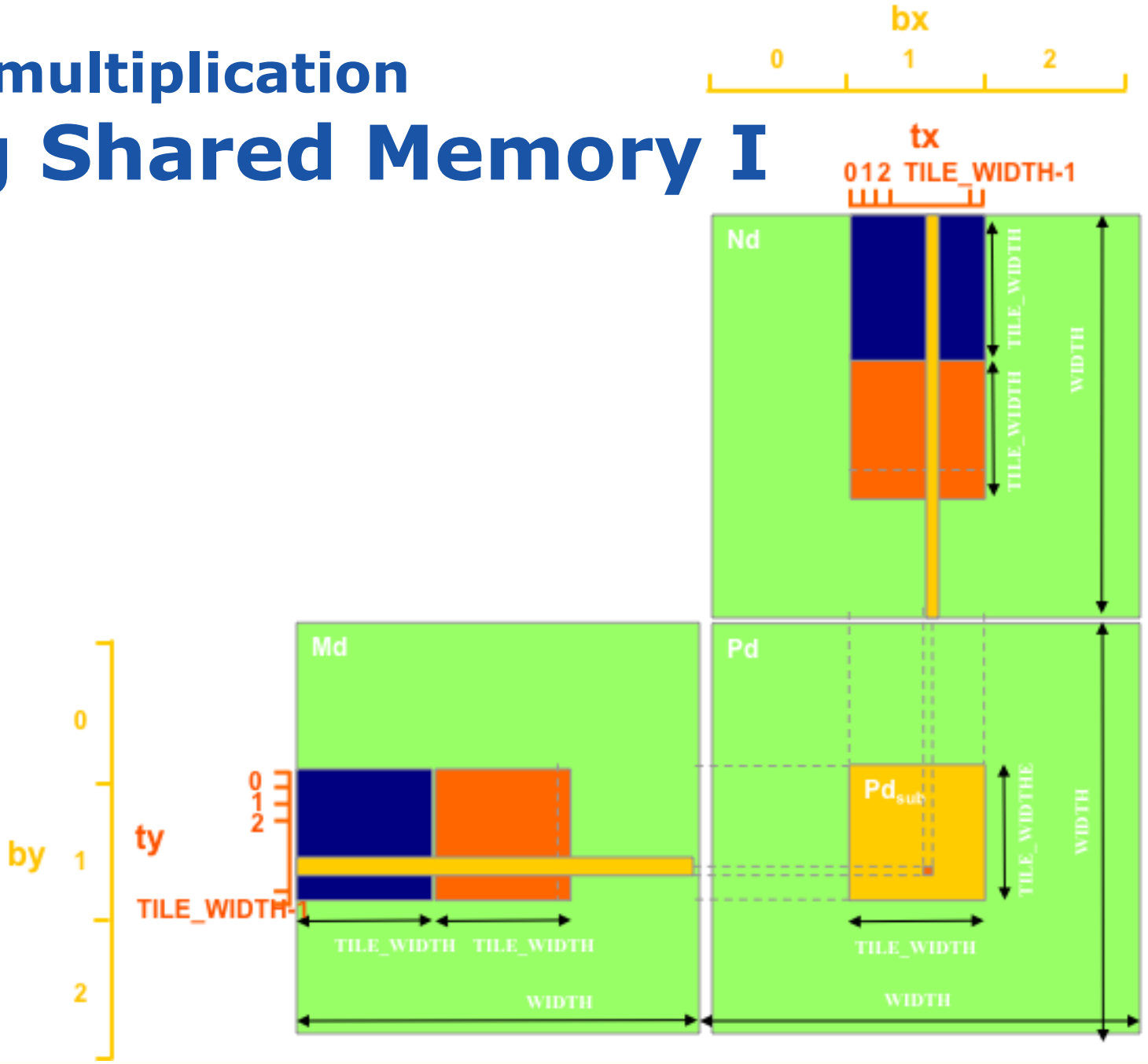


Device memory access



Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

Matrix multiplication
Using Shared Memory I



Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

Matrix multiplication

Using Shared Memory II

```
__shared __float Mds[TILE_WIDTH][TILE_WIDTH];
__shared __float Nds[TILE_WIDTH][TILE_WIDTH];
int bx = blockIdx.x;  int by = blockIdx.y;
int tx = threadIdx.x; int ty = threadIdx.y;
int Row = by*TILE_WIDTH + ty;
int Col = bx*TILE_WIDTH + tx;
float sum = 0;

for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
    Nds[ty][tx] = Nd[(m*TILE_WIDTH + ty)*Width + Col];
    __syncthreads();
    float sum;
    for (int k = 0; k < TILE_WIDTH; ++k)
      sum += Mds[ty][k] * Nds[k][tx];
    __syncthreads();
}
Pd[Row*Width + Col] = sum;
```