Advanced GPU Parallel Programming (Part II)

CS240A. 2017

Tao Yang

Slides adopted from CIS565 Patrick Cozzi @UPenn and Derek Hower @AMD

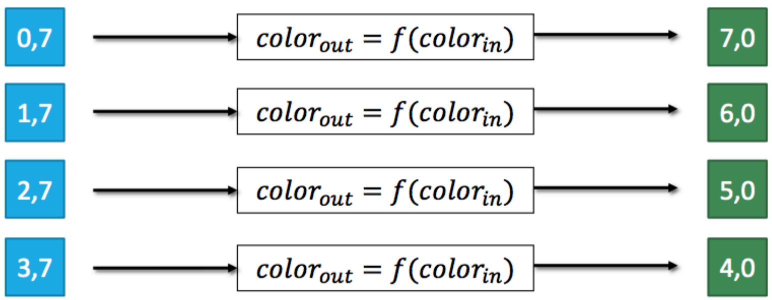
UCSB

Table of Content

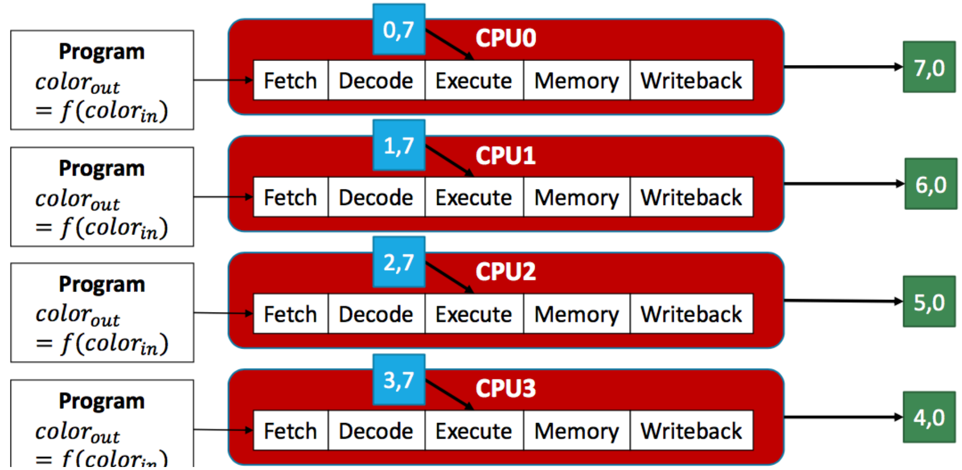
- Review of SPMD, SIMD, SIMT
  - A comparison of CPU vs GPU
  - Why GPU executes so many threads
- Thread Scheduling in GPU
- Application thread naming
- Memory model of GPU and impact
- Matrix multiplication example
  - Optimization with block shared memory

Running SPMD on MIMD and SIMD

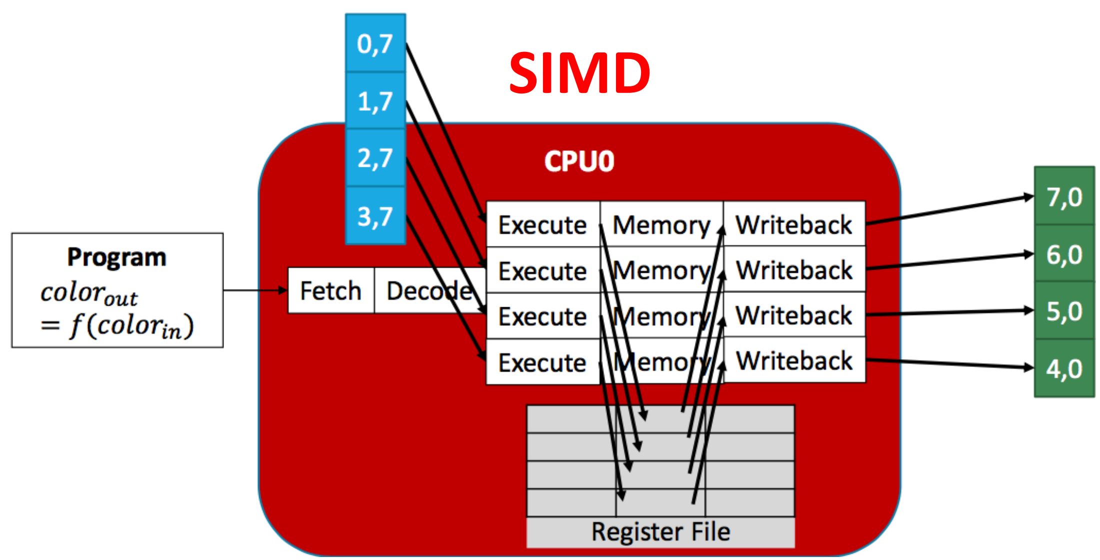
SPMD (Single Program Multiple Data)



MIMD



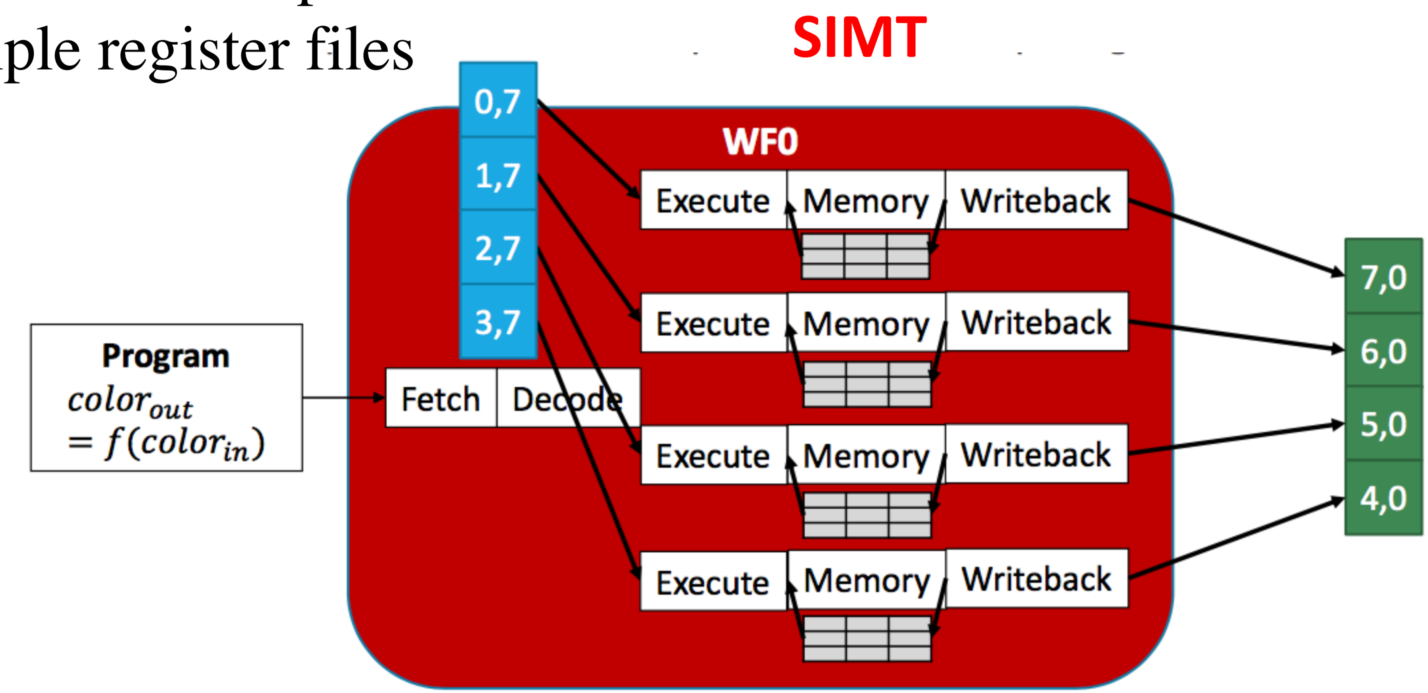
SIMD:
Single instruction multiple data
One Thread + Data Parallel Ops→Single PC, single register file



Derek Howe@AMD

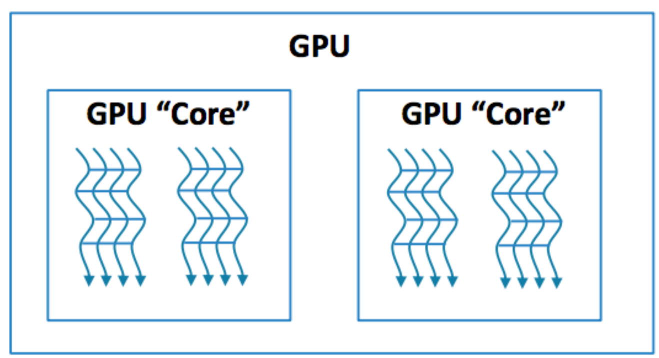
SIMT: Single Instruction Multiple Thread

Multiple Threads + Scalar Ops
→One PC, Multiple register files



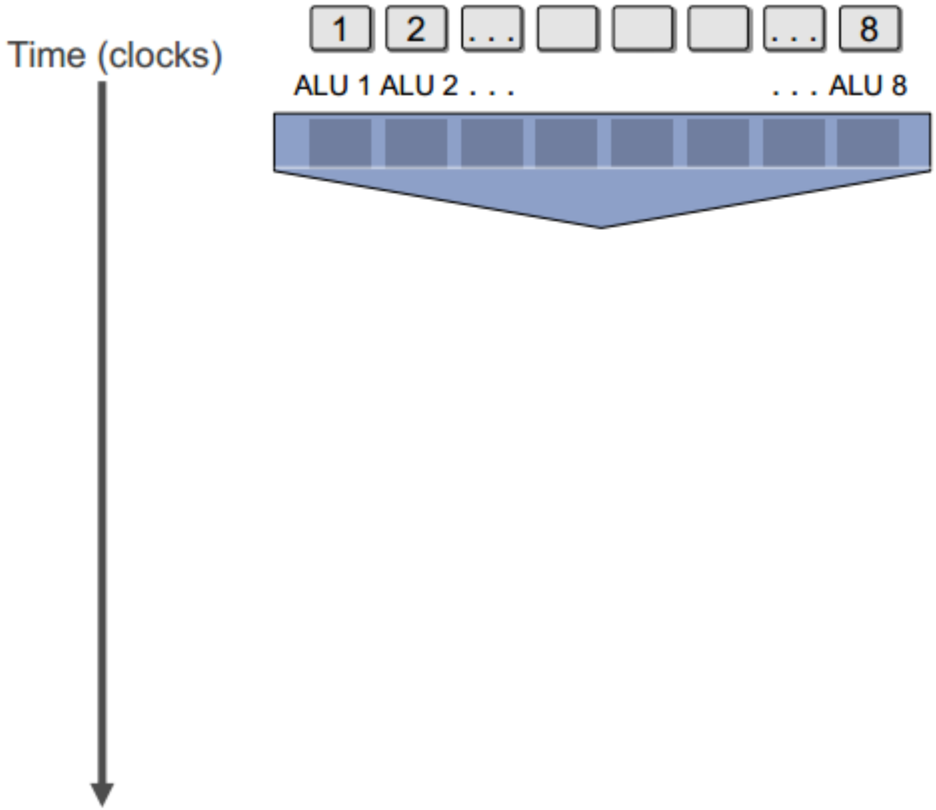
Multicore Multithreaded SIMT

Many SIMT "threads" grouped together into GPU "Core"



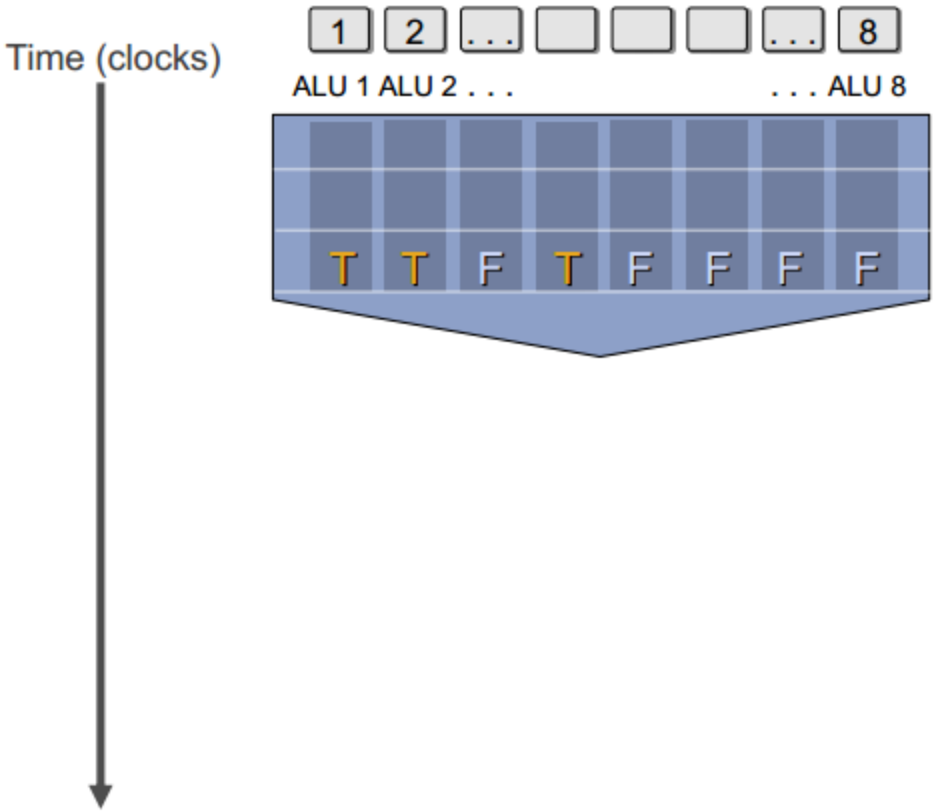
UCSB

But what about branches?




```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
<resume unconditional
 shader code>
```

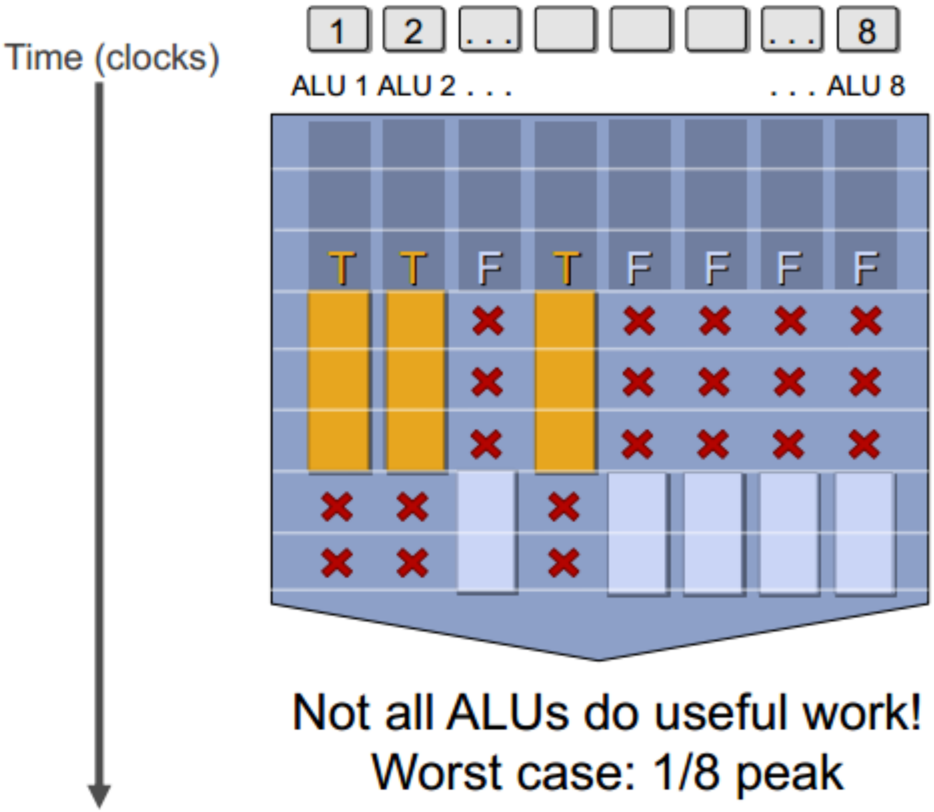
But what about branches?




```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
 shader code>
```

Branching divergence

But what about branches?




```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
 shader code>
```

Not all ALUs do useful work!
Worst case: 1/8 peak performance

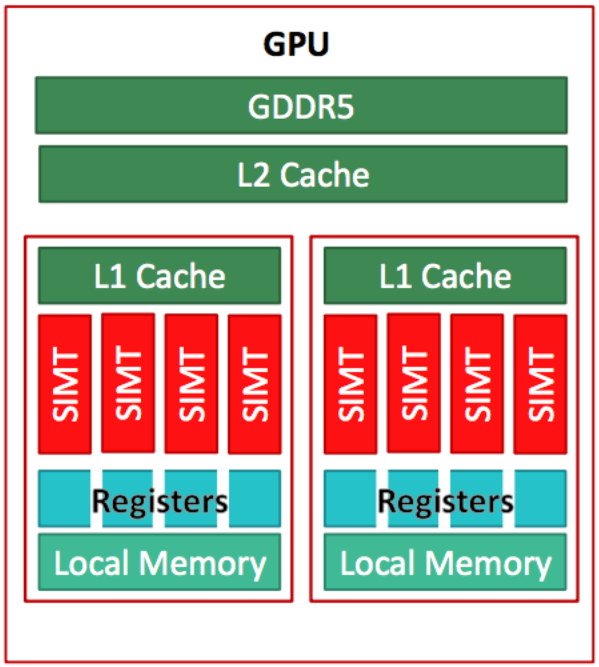
Clarification

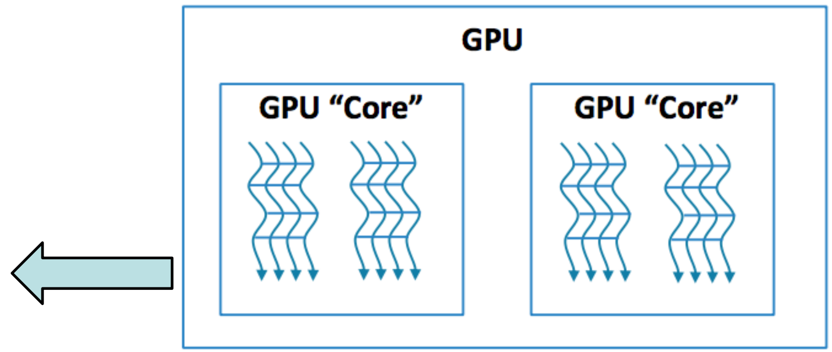
SIMD processing does not imply SIMD instructions

- Option 1: explicit vector instructions
  - x86 SSE, AVX, Intel Larrabee
- Option 2: scalar instructions, implicit HW vectorization
  - HW determines instruction stream sharing across ALUs (amount of sharing hidden from software)
  - NVIDIA GeForce (“SIMT” warps), ATI Radeon architectures (“wavefronts”)

In practice: 16 to 64 fragments share an instruction stream.

GPU Architecture and Execution Model

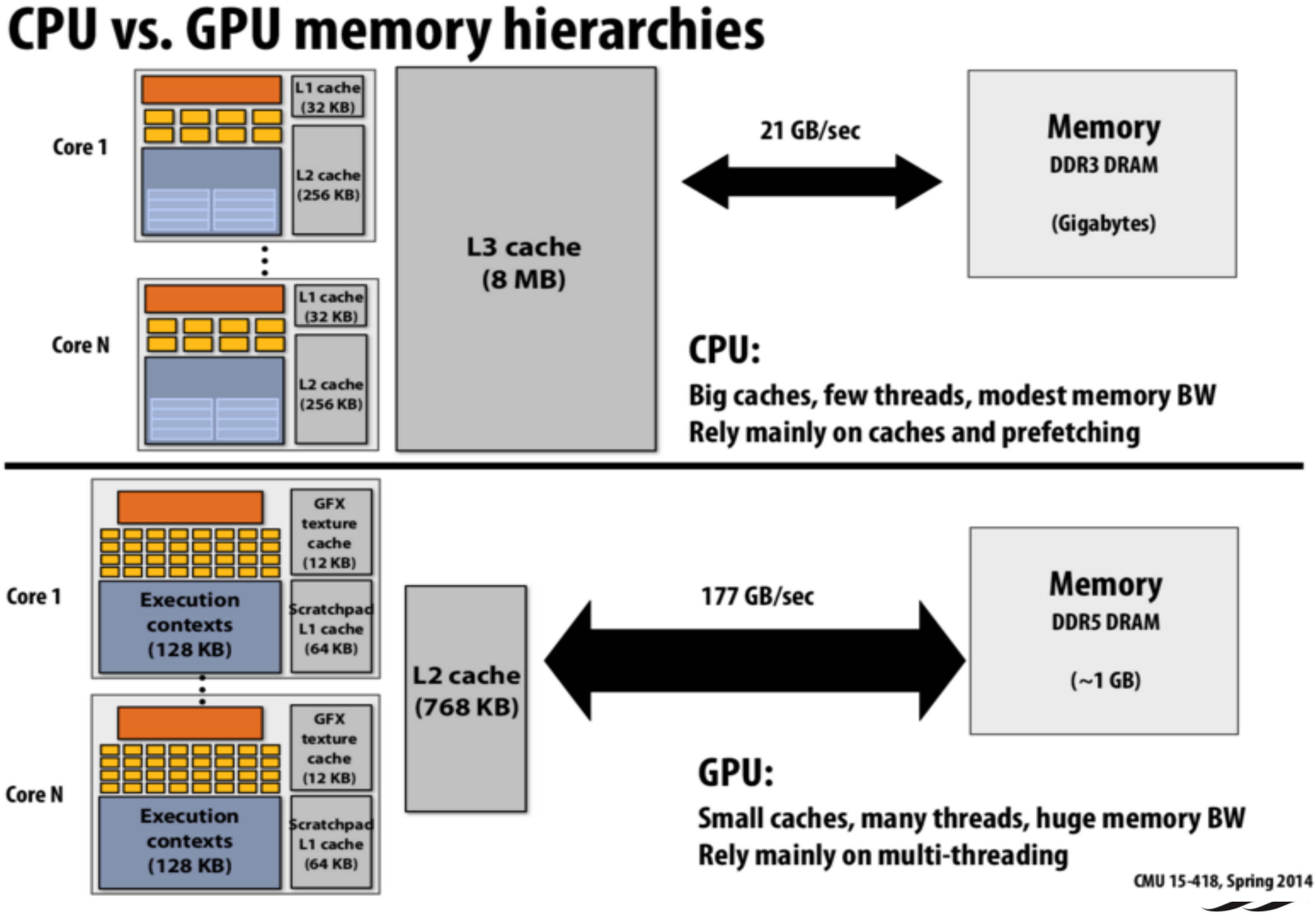




Execution Model Comparison

	MIMD/SPMD	SIMD/Vector	SIMT
Example Architecture	Multicore CPUs	x86 SSE/AVX	GPUs
Pros	More general: supports TLP	Can mix sequential & parallel code	Easier to program Gather/Scatter operations
Cons	Inefficient for data parallelism	Gather/Scatter can be awkward	Divergence kills performance

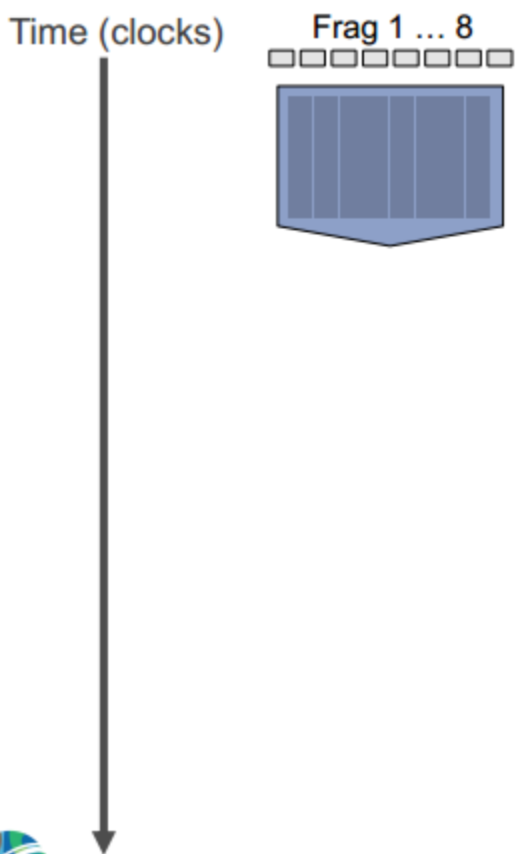
CPU vs. GPU Architecture

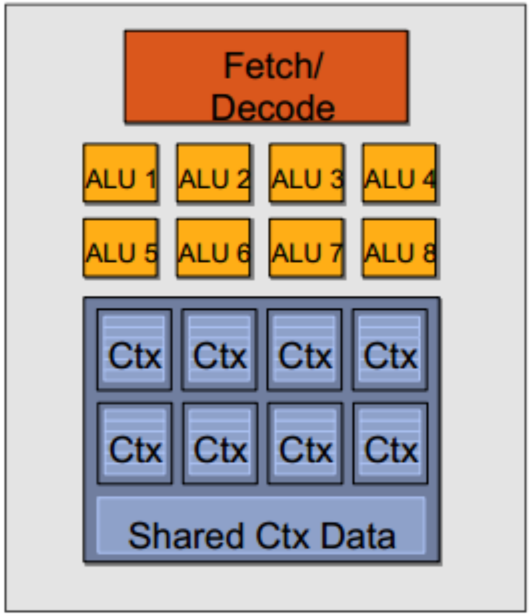


Hide slow memory latency: Multi-threading in addition to caching

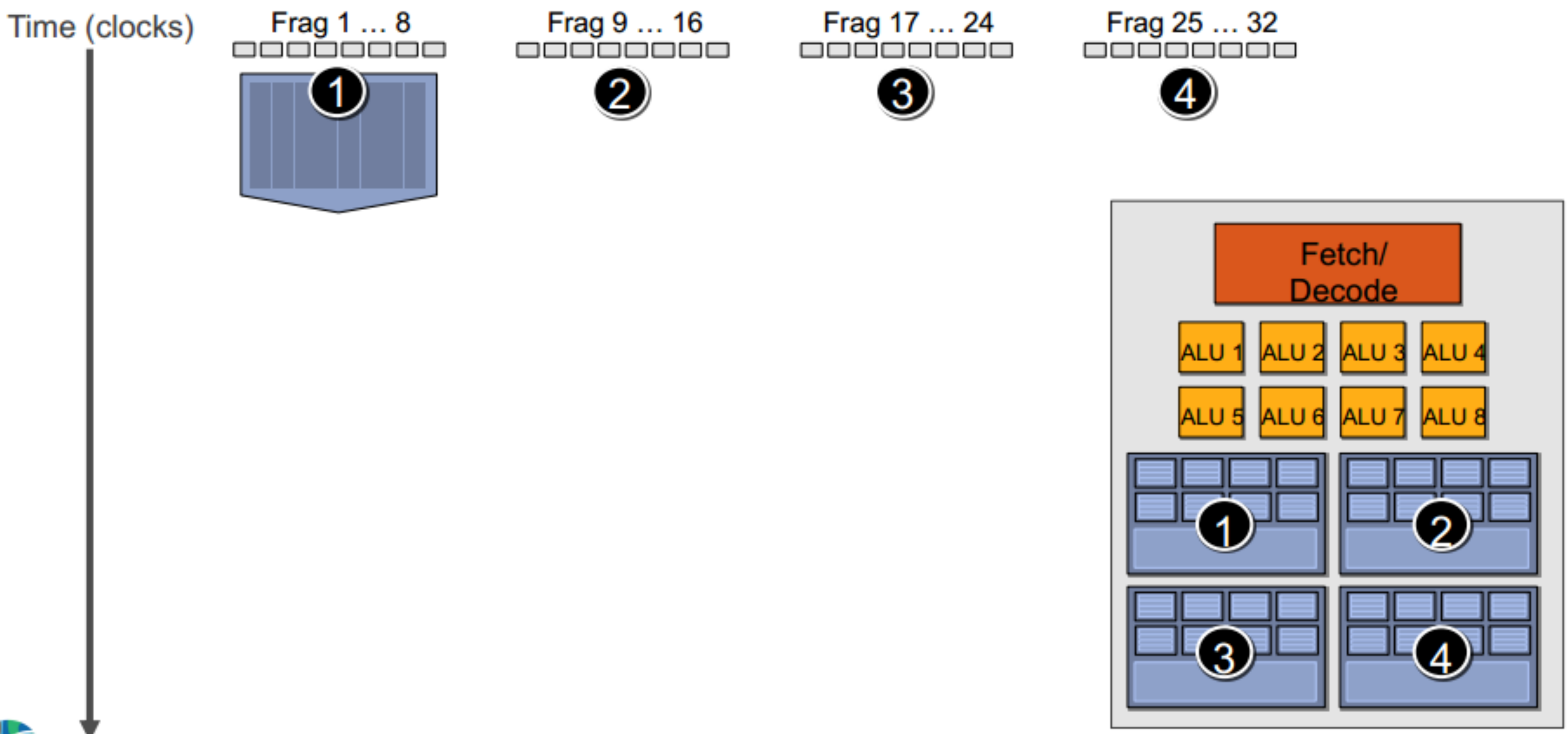
- **Shader = GPU scalar processors**

Hiding shader stalls

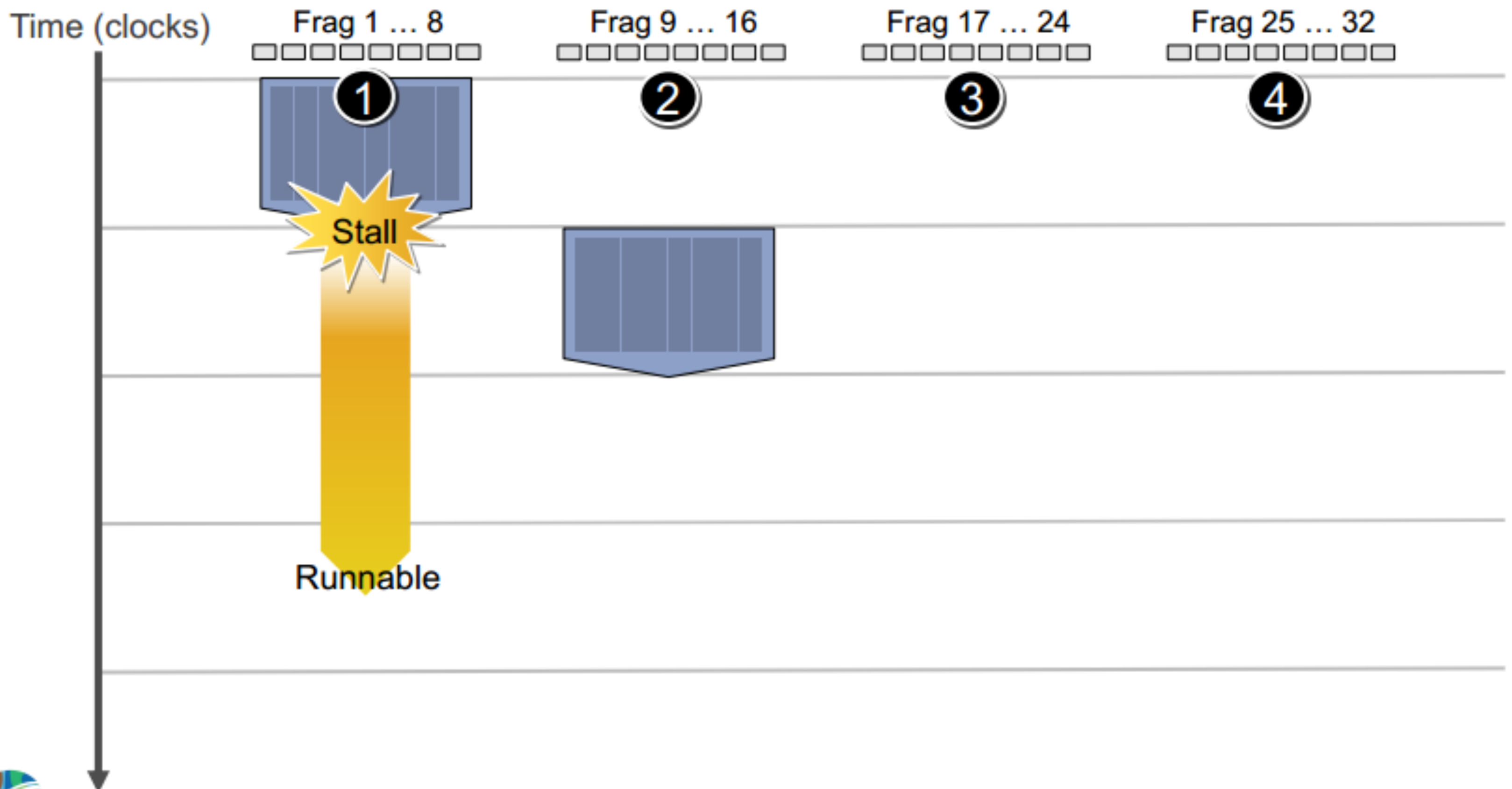




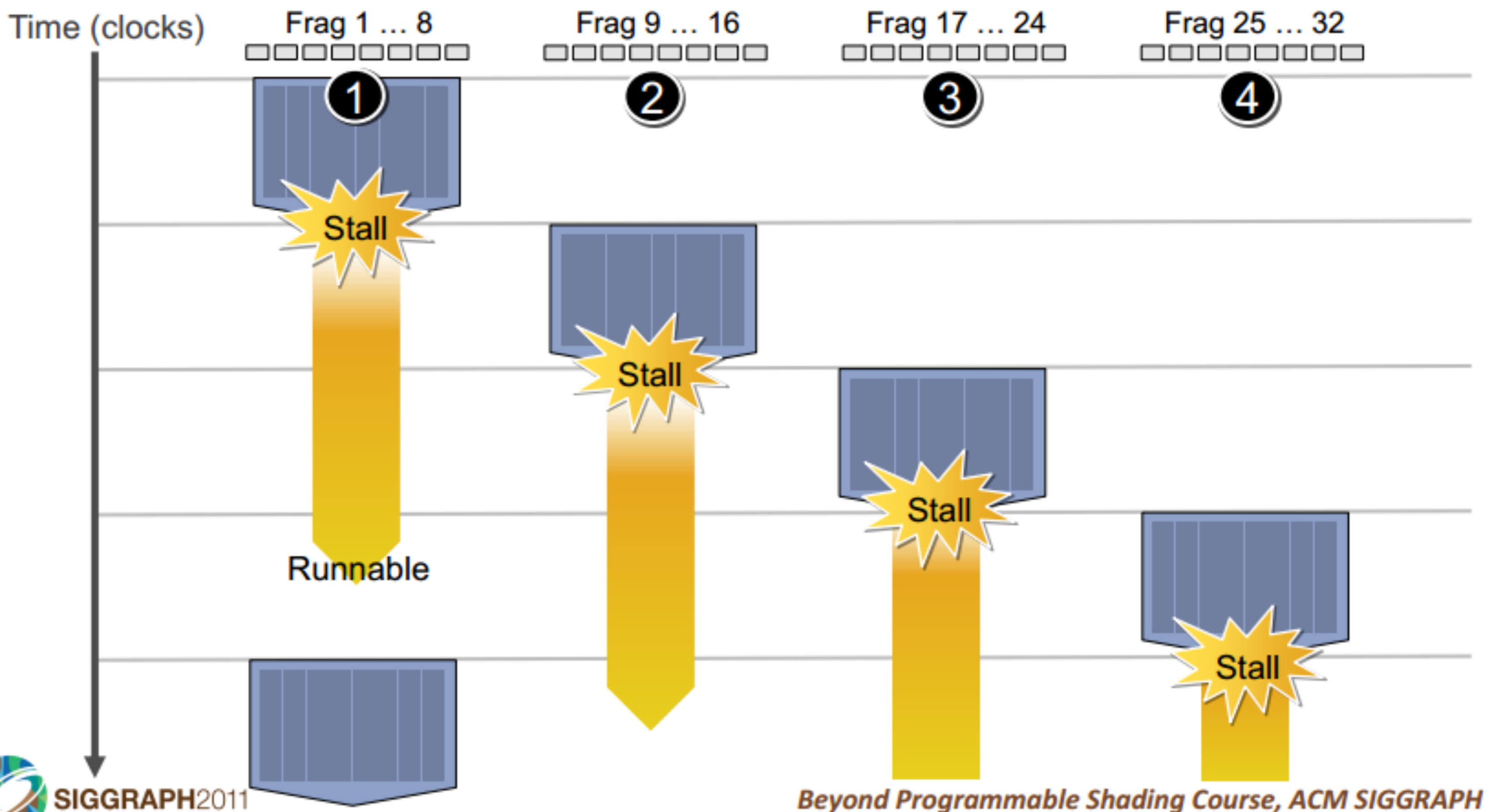
Hiding shader stalls



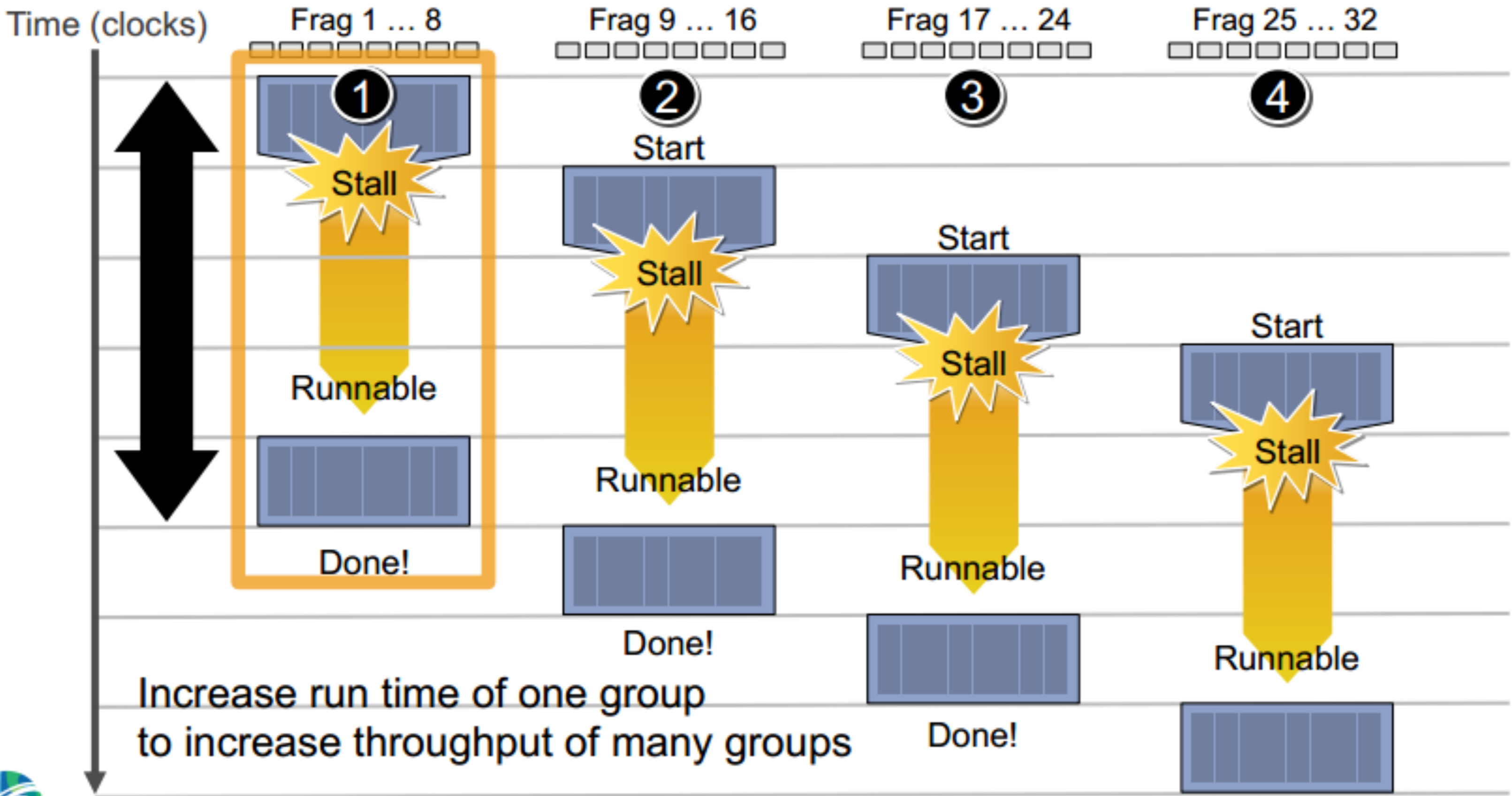
Hiding shader stalls



Hiding shader stalls



Throughput!

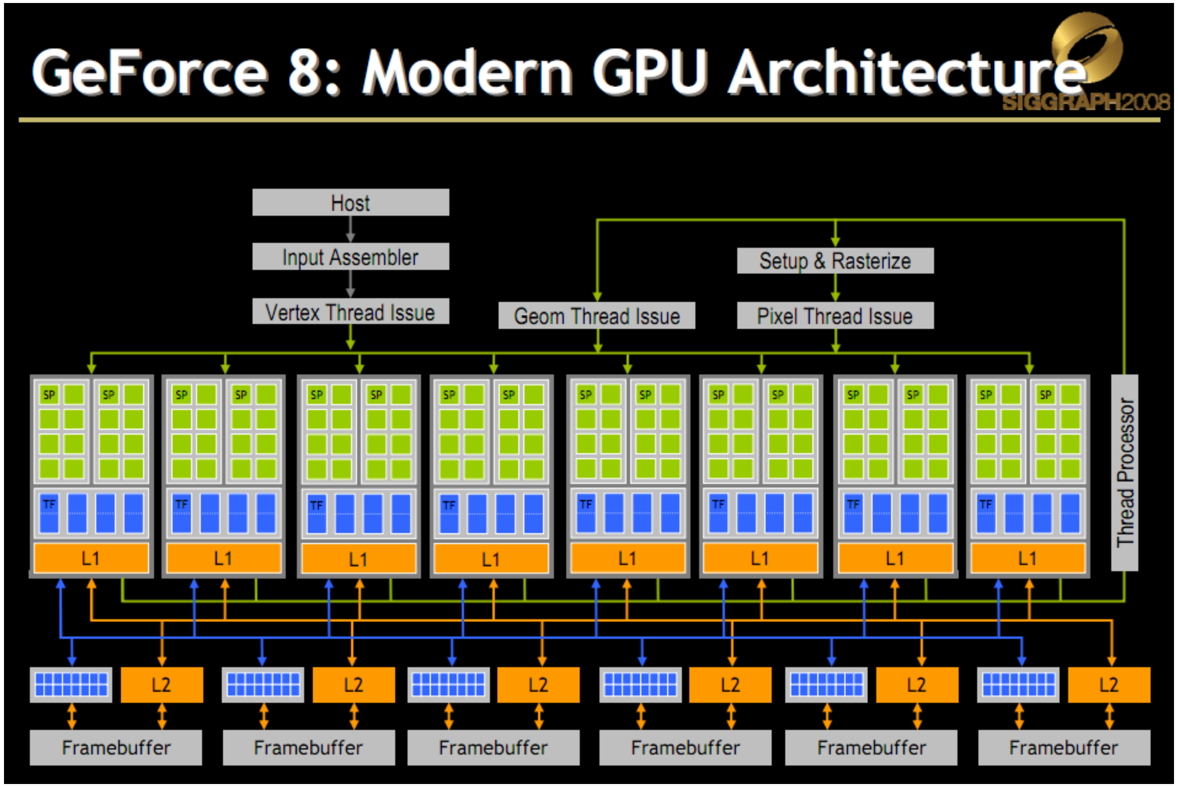


Increase run time of one group
to increase throughput of many groups

Scheduling Threads

Example with G80

- 16 SMs (Stream multiprocessors)
- Each with 8 SPs
  - Scalar processors
  - 128 total SPs
- Each SM hosts up to 768 threads
- Up to 12,288 threads in flight

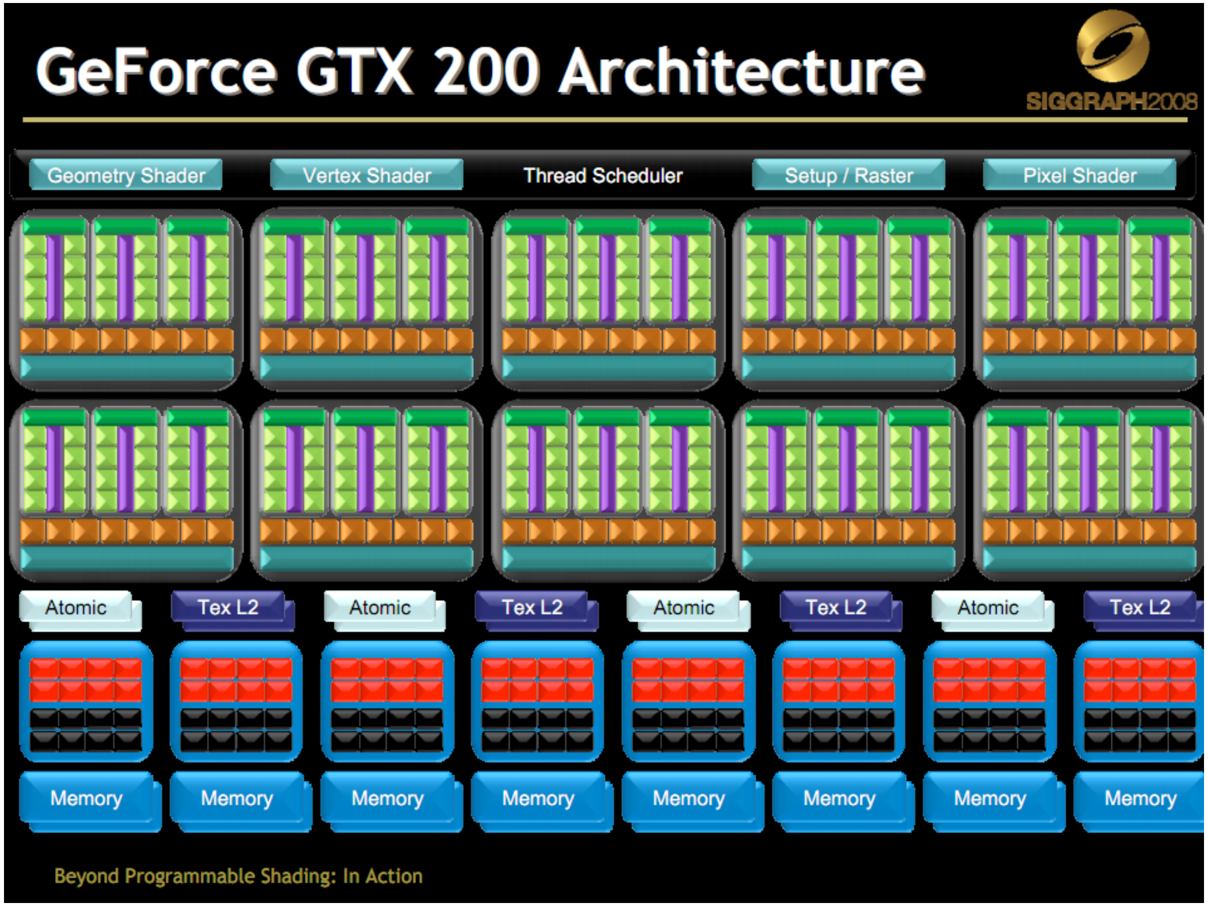


Slide from David Luebke: http://s08.idav.ucdavis.edu/luebke-nvidia-gpu-architecture.pdf

Scheduling Threads in GT200

GT200

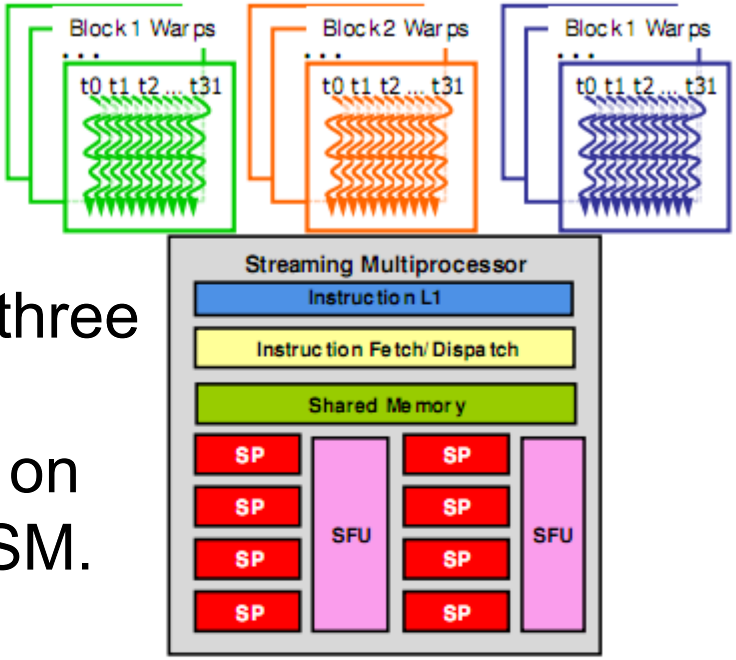
- 30 SMs
- Each with 8 SPs
  - 240 total SPs
- Each SM hosts up to
  - 8 blocks, or
  - 1024 threads
- In flight, up to
  - 240 blocks, or
  - 30,720 threads



Slide from David Luebke: http://s08.idav.ucdavis.edu/luebke-nvidia-gpu-architecture.pdf

Scheduling Warp by Warp of Threads

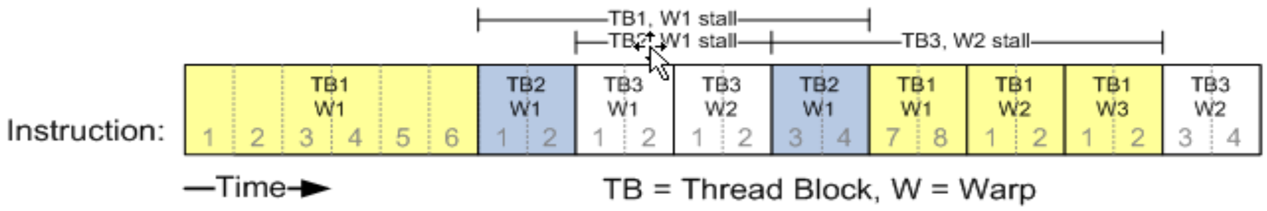
- *Warp* – threads from a block
  - G80 / GT200 – *32* threads
  - Run on the same SM
  - Unit of thread scheduling
  - Consecutive `threadIdx` values
  - An implementation detail – in theory
    - `warpSize`



- Warps for three blocks scheduled on the same SM.

Warp Scheduling

- SM implements zero-overhead warp scheduling
  - At any time, only one of the warps is executed by SM
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a warp execute the same instruction when selected



Slide from: http://courses.engr.illinois.edu/ece498/al/Syllabus.html

4 Cycles for G80 to Dispatch a Wrap

- G80: 32 threads per warp but 8 SPs per SM.
- When an SM schedules a warp:
  - Its instruction is ready
  - 8 threads enter the SPs on the 1^{st} cycle
  - 8 more on the 2^{nd}, 3^{rd}, and 4^{th} cycles
  - Therefore, 4 cycles are required to dispatch a warp

What happens if branches in a warp diverge?

Remember this:

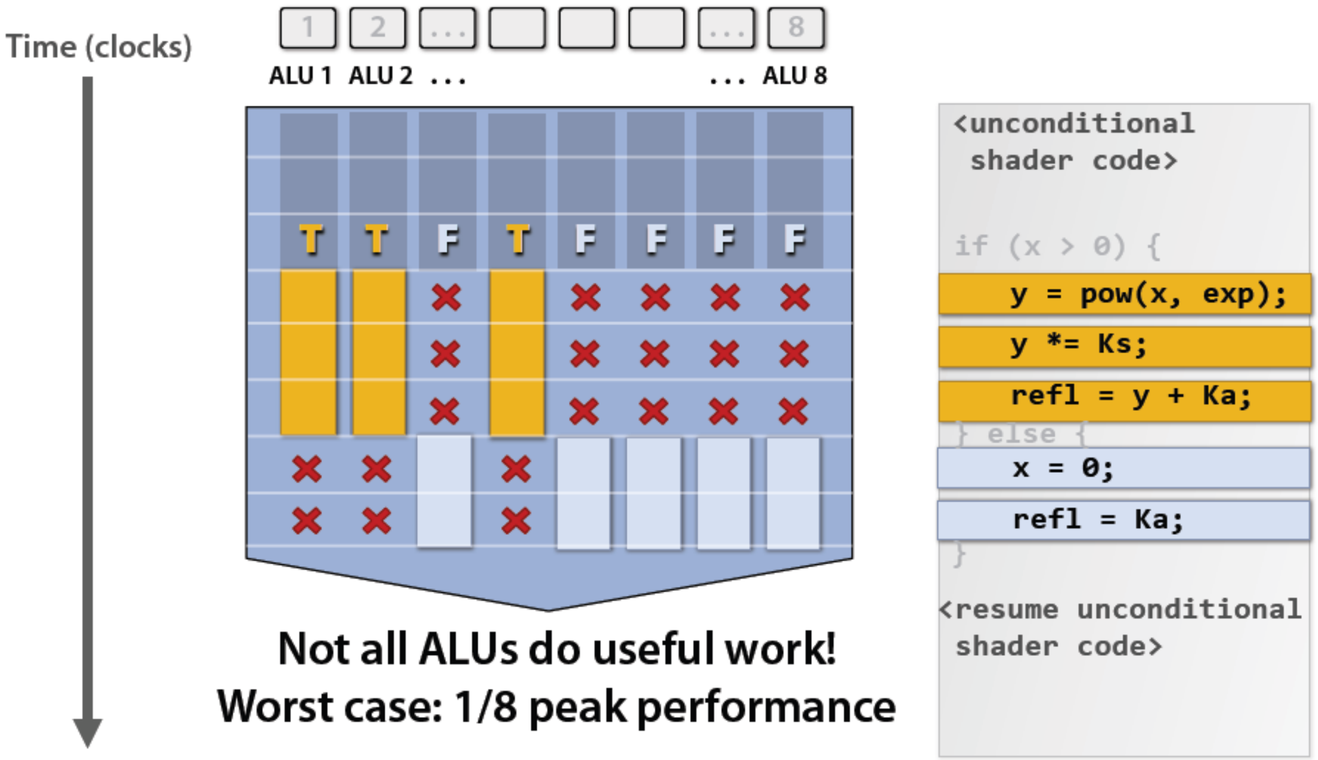


Image from: http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf

UCSB

Question on Scheduling Threads

- A SM on GT200 can host up to 1024 threads, how many warps is that?
- If 3 blocks are assigned to an SM and each block has 256 threads, how many warps are there?

  - 1024 / 32 = 32 warps
  - (3 * 256) / 32 = 24 warps

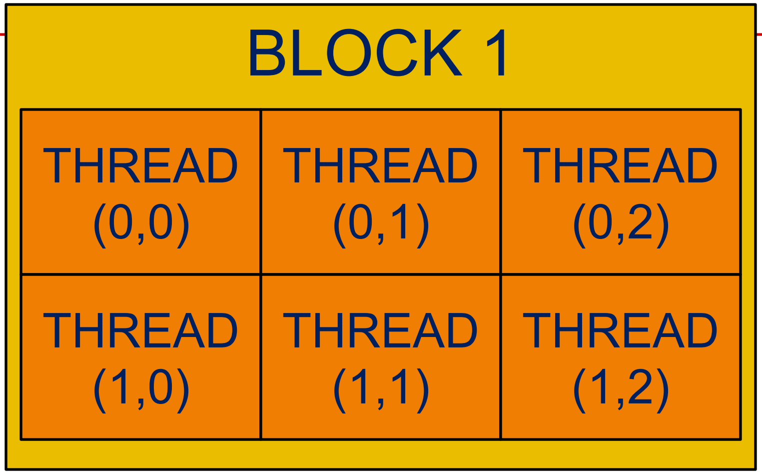
Question on memory latency hiding

- Question
  - A thread kernel has
    - 1 global memory read (200 cycles)
    - 16 cycles of computation
  - How many warps are required to hide the memory latency?

- Answer
  - We need to cover 200 cycles
    - $200 / 16 = 12.5$
  - 13 warps are required

Application Threads

- **Threads grouped in thread blocks**
  - 128, 192 or 256 threads in a block
- **Blocks are further grouped as a grid**



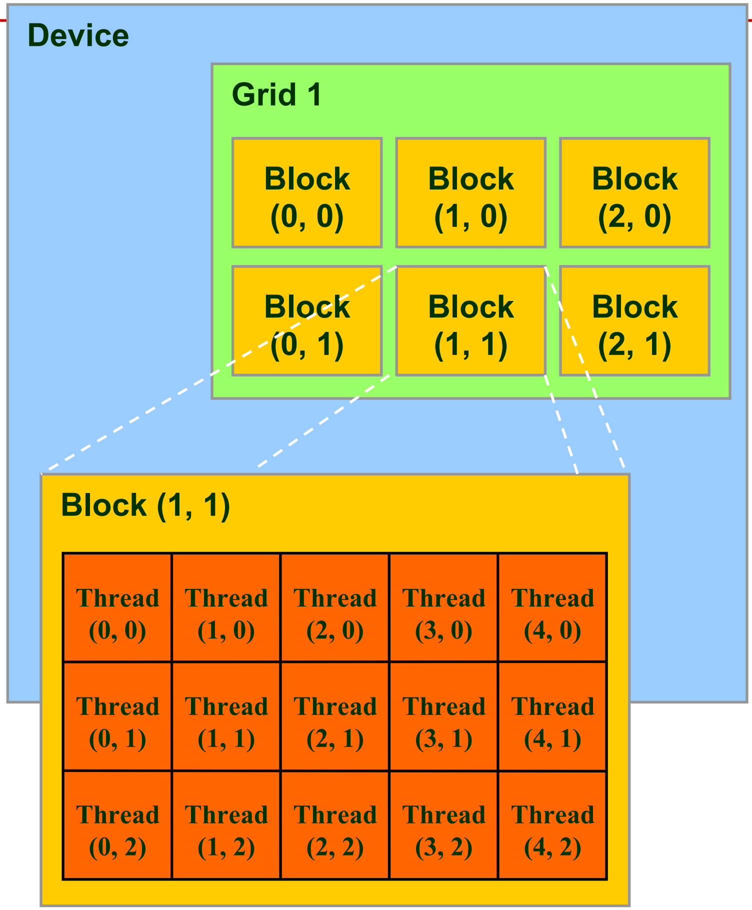
- <u>One thread block executes on one SM</u>
  - All threads sharing the group local memory (called 'shared memory' in NVIDIA GPU)
  - 32 threads are fetched and executed simultaneously ('warp')

UCSB

Block and Thread IDs at CUDA

- **Threads and blocks have IDs**
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D (blockIdx.x, blockIdx.y)
  - Thread ID: 1D, 2D, or 3D (threadIdx.{x,y,z})

Special registers

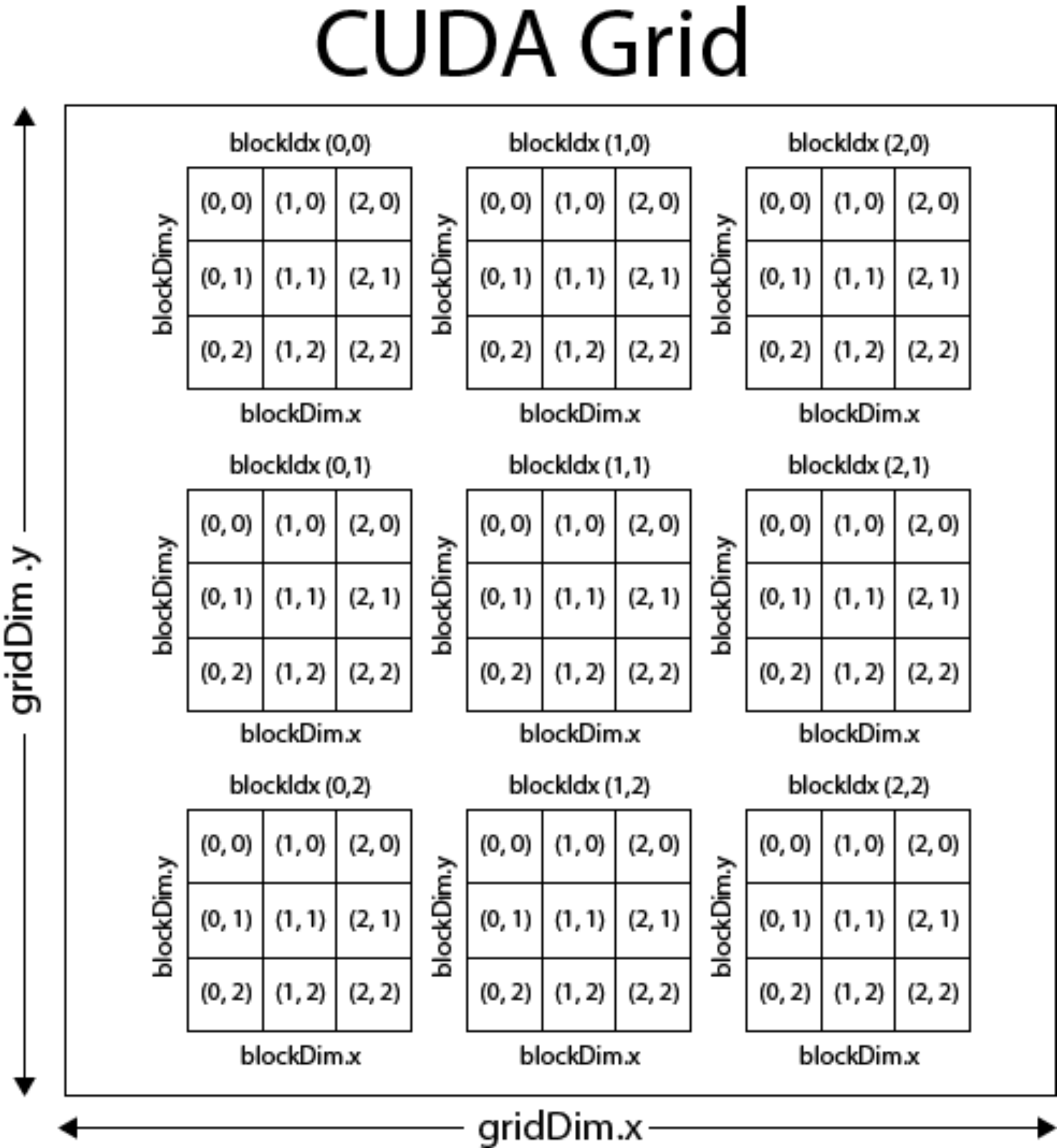


Courtesy: NDVIA

2D Grid and 2D block

Declare:

dim3 dimGrid(3, 3, 1);
dim3 dimBlock(3,3, 1);



2D Grid and 2D block

Host Declaration:

dim3 dimGrid(3, 3, 1);
dim3 dimBlock(3,3, 1);

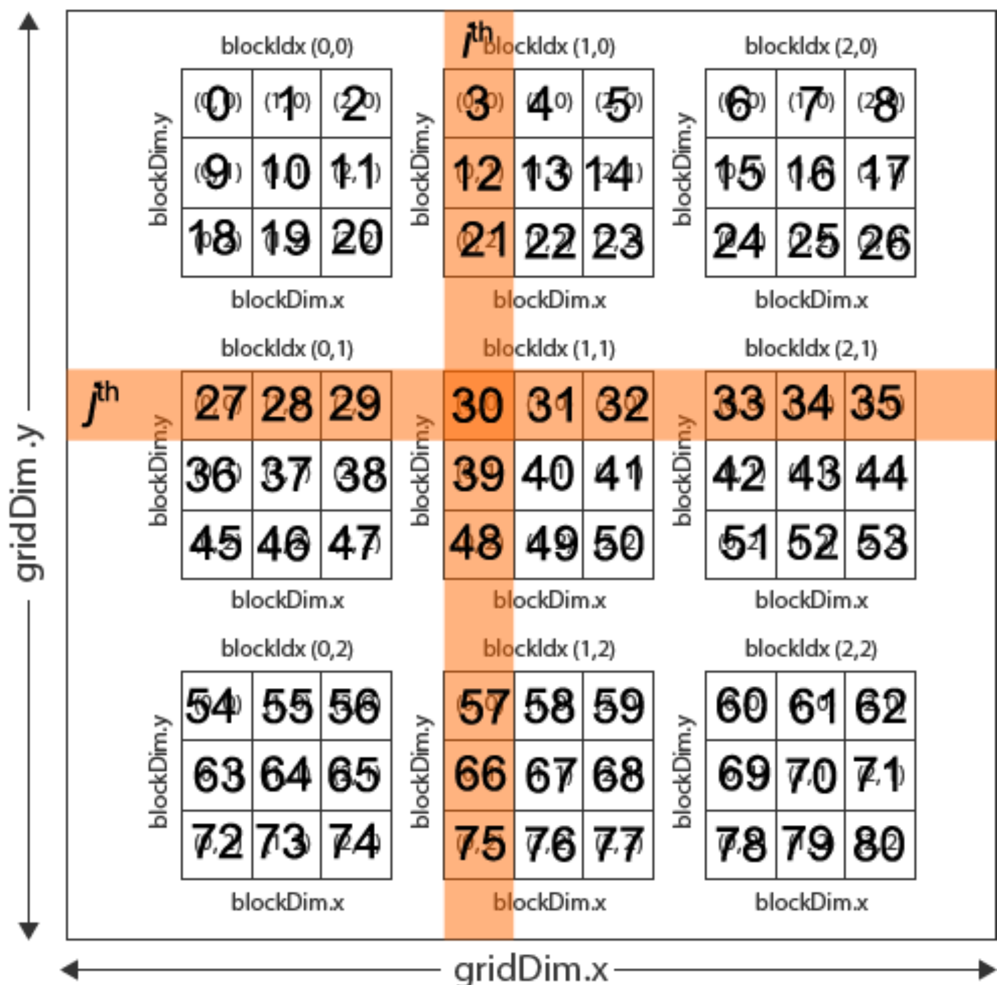
Host Call:

MatrixAccessDevice<<< dimGrid, dimBlock>>>(…);

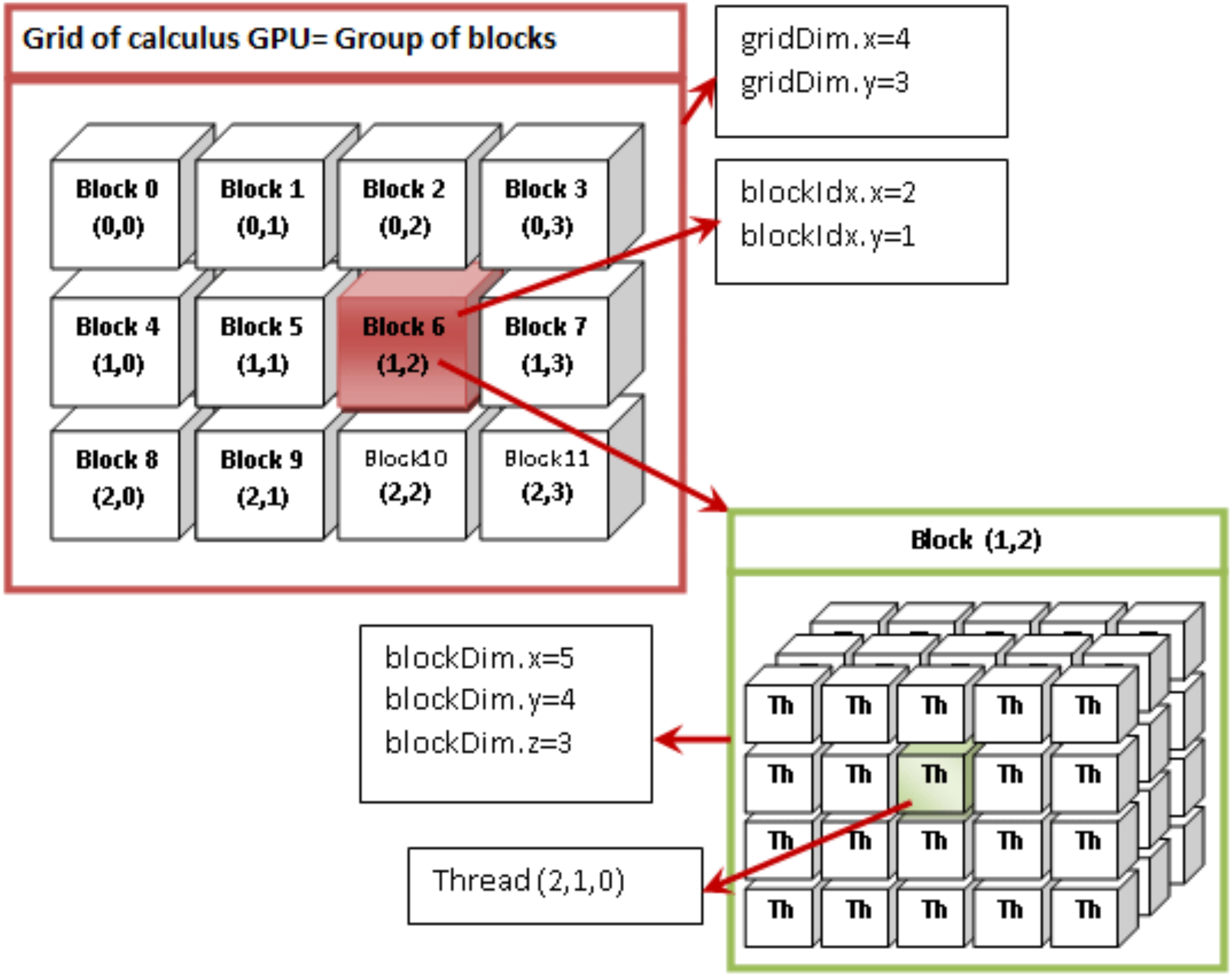
Device Access:

int i=blockIdx.x * 3+threadIdx.x;
int j=blockIdx.y * 3+threadIdx.y;

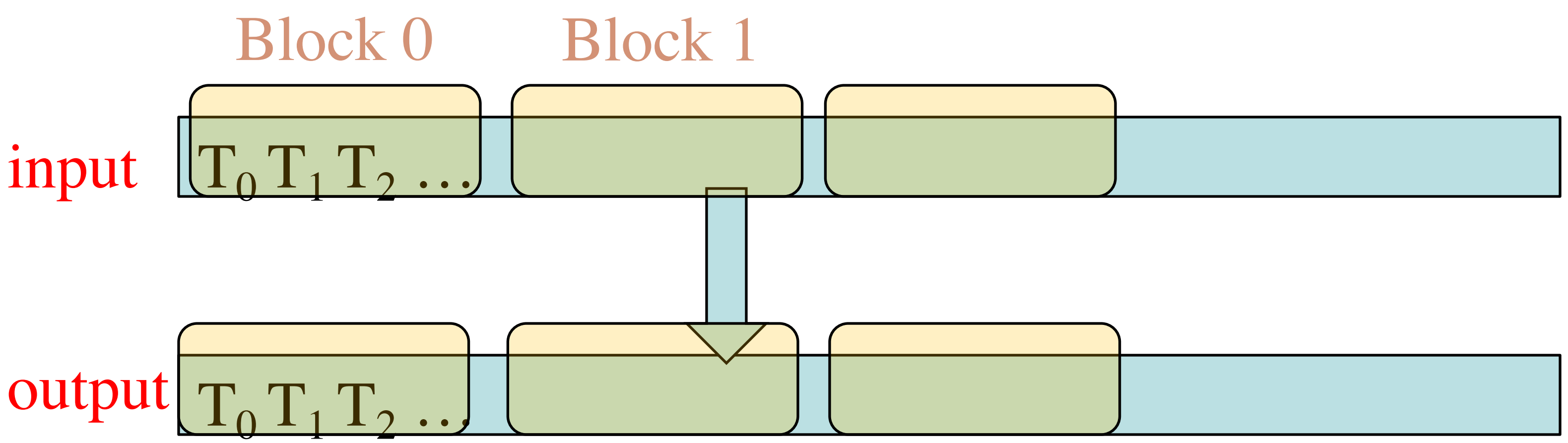
CUDA Grid



More Example: 2D block, 3D threads

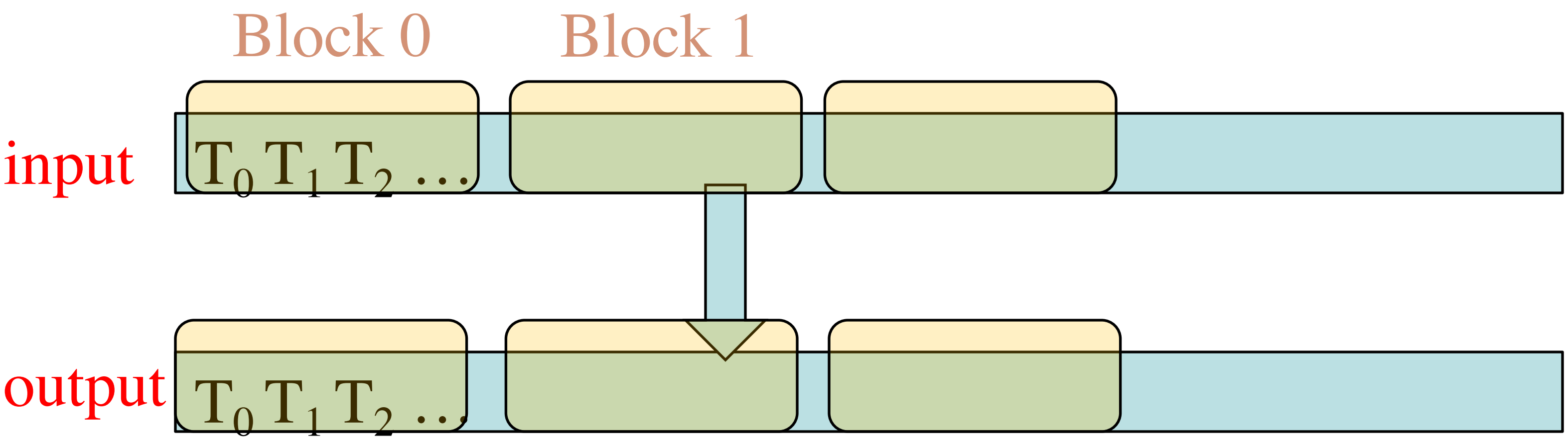


CUDA Review: Thread Hierarchies




```
int threadID = blockIdx.x *
  blockDim.x + threadIdx.x;
float x = input[threadID];
float y = func(x);
output[threadID] = y;
```

Cuda Review: Thread Hierarchies

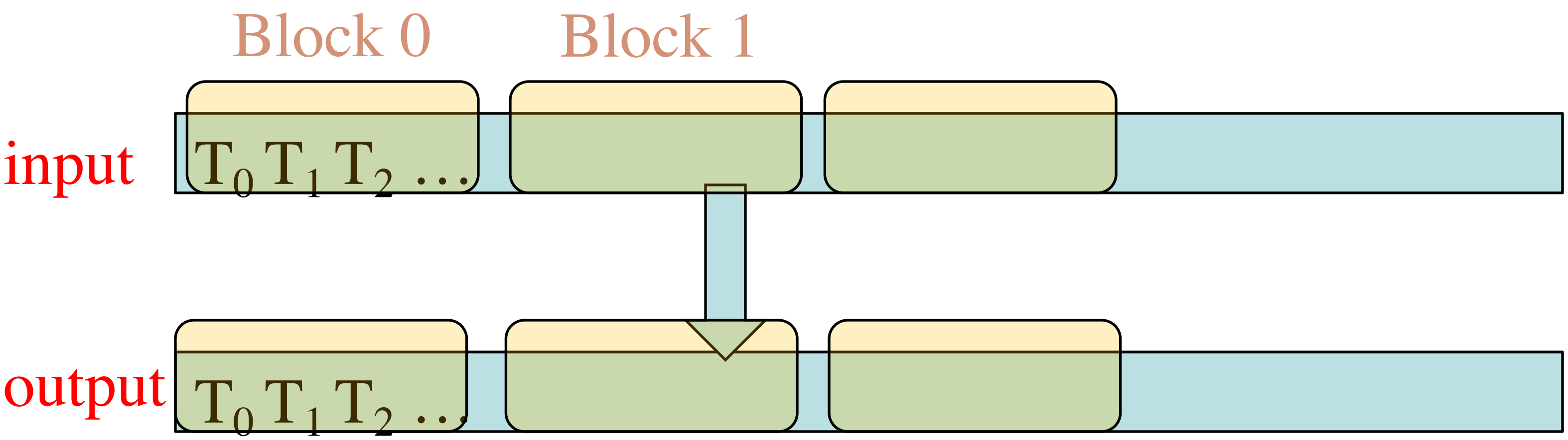



```
int threadID = blockIdx.x *
  blockDim.x + threadIdx.x;
float x = input[threadID];
float y = func(x);
output[threadID] = y;
```

Use grid and block position to compute a thread id

UCSB

Cuda Review: Thread Hierarchies

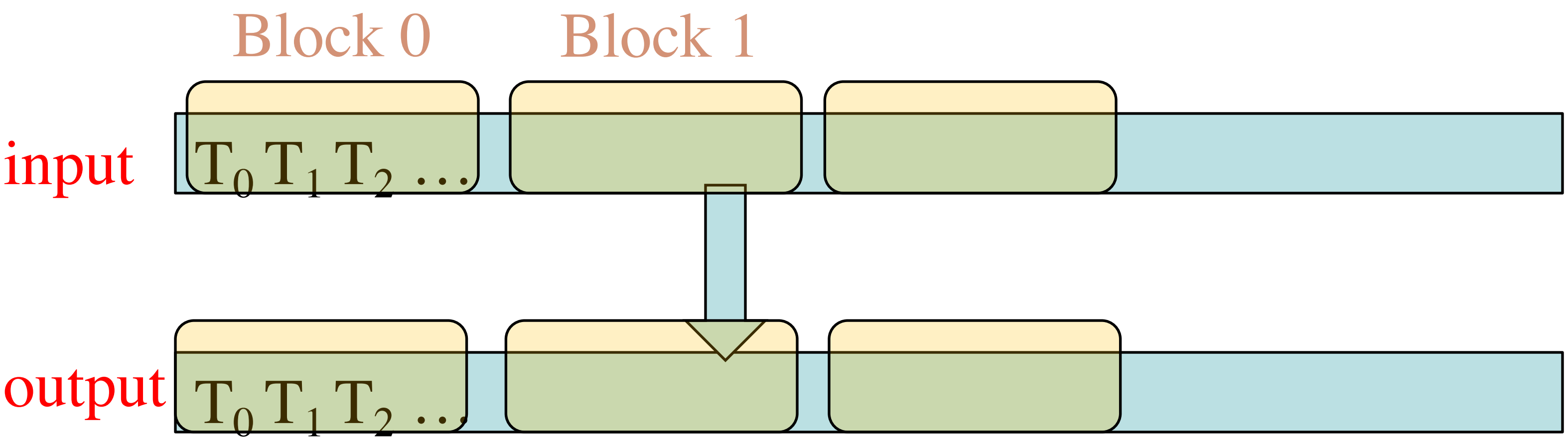



```
int threadID = blockIdx.x *
  blockDim.x + threadIdx.x;
float x = input[threadID];
float y = func(x);
output[threadID] = y;
```

Use thread id to read from input

UCSB

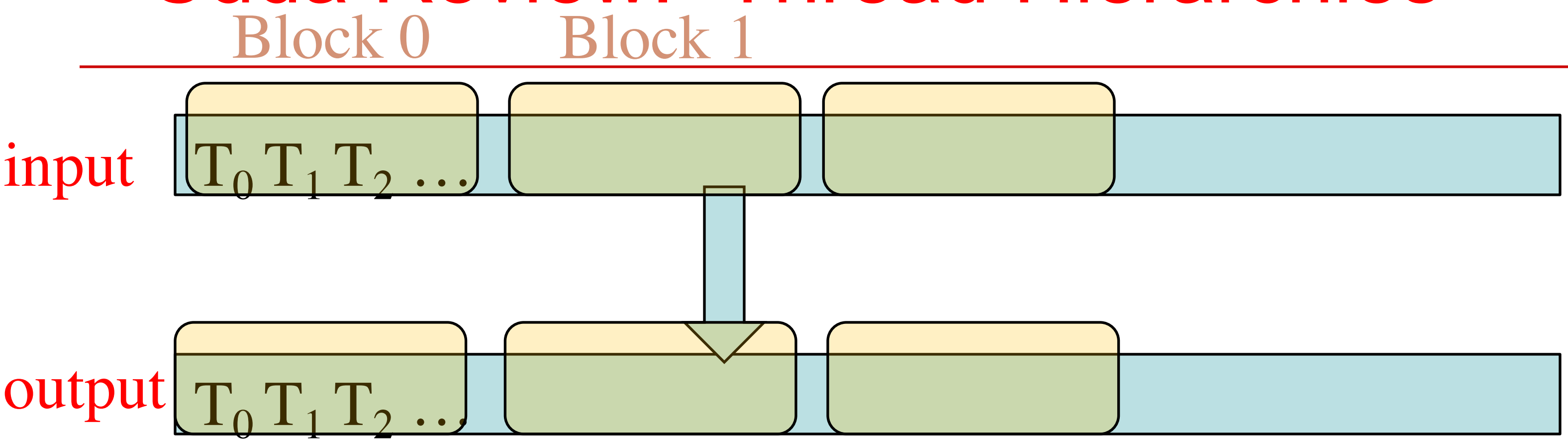
Cuda Review: Thread Hierarchies




```
int threadID = blockIdx.x *
  blockDim.x + threadIdx.x;
float x = input[threadID];
float y = func(x);
output[threadID] = y;
```

Run function on input: data-parallel!

Cuda Review: Thread Hierarchies

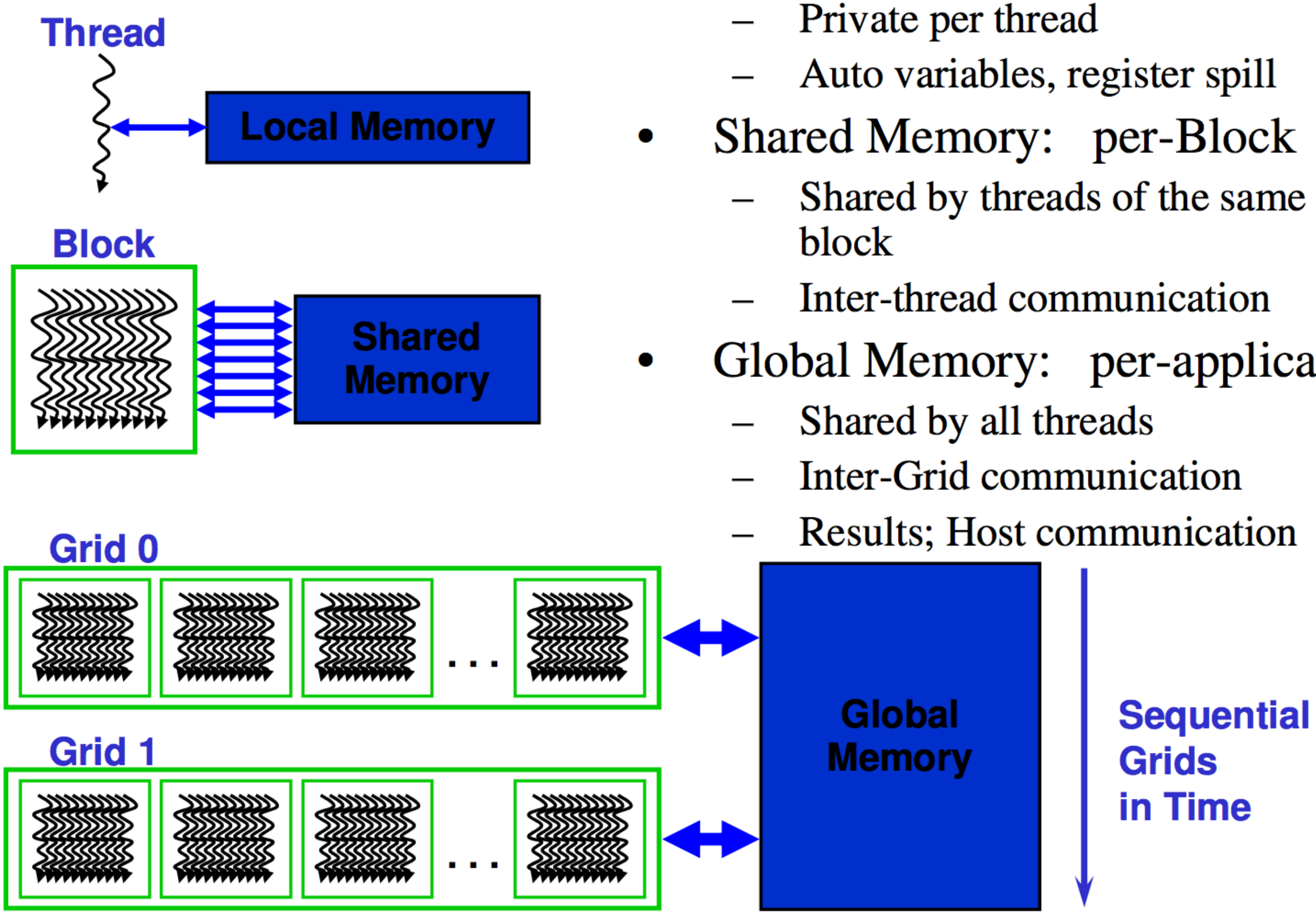



```
int threadID = blockIdx.x *
  blockDim.x + threadIdx.x;
float x = input[threadID];
float y = func(x);
output[threadID] = y;
```

Use thread id to output result

UCSB

Memory Access during Thread Execution



- Local Memory: per-thread
  - Private per thread
  - Auto variables, register spill
- Shared Memory: per-Block
  - Shared by threads of the same block
  - Inter-thread communication
- Global Memory: per-application
  - Shared by all threads
  - Inter-Grid communication
  - Results; Host communication

Cuda Memory Model

- Device code can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
- Host code can
  - R/W per grid global and constant memories

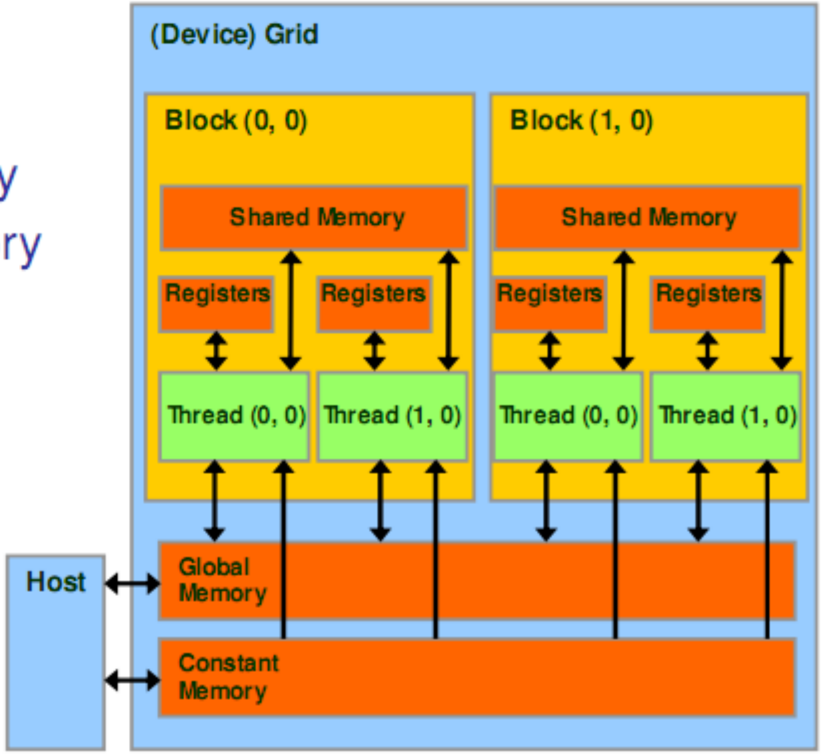
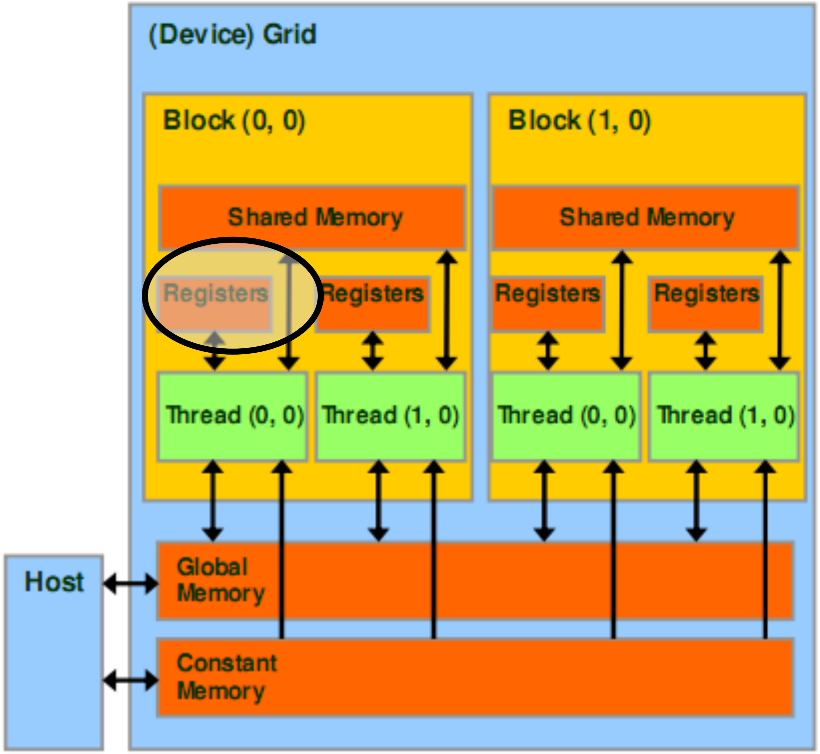


Image from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf

Cuda Memory Model

- Registers
  - Per thread
  - Fast, on-chip, read/write access
  - Increasing the number of registers used by a kernel has what affect?



Potentially decrease the number of threads (well, blocks) available to that SM for scheduling.

UCSB

Impact of register number

- Per SM
  - Up to 768 threads
  - 8K registers
- How many registers per thread?

- 8K / 768 = 10 registers per thread

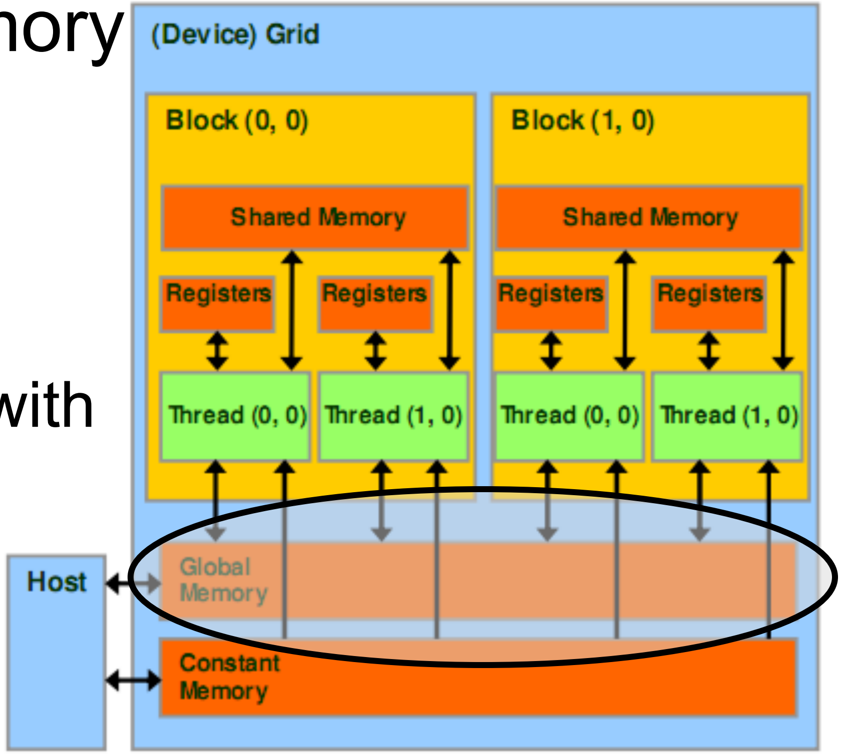
- **Exceeding limit reduces threads by the block**
  - Example: Each thread uses 11 registers, and each block has 256 threads
    - How many threads can a SM host?
    - How many warps can a SM host?

8K/11/256= 2 blocks. Host 512 threads

512 / 32 = 16 warps. Less threads available for hiding memory latency

Memory Model

- Local Memory
  - Stored in global memory
    - Copy per thread
  - Used for automatic arrays
    - Unless all accessed with only constant indices

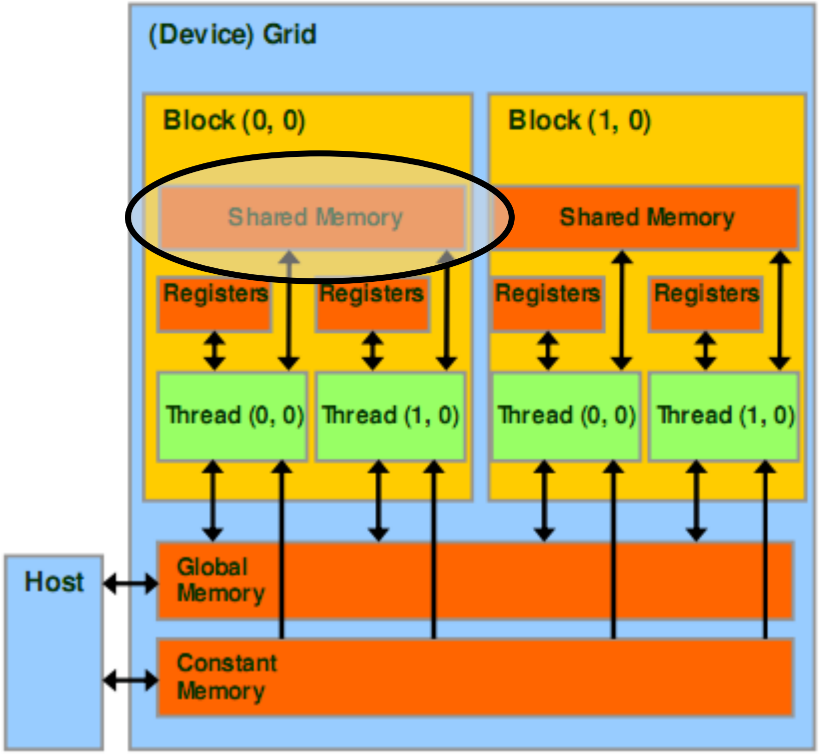


Memory Model

- **Special Registers**
  - Used for built-in variables
  - threadIdx
  - blockIdx
  - blockDim

Memory Model

- Shared Memory
  - Per block
  - Fast, on-chip, read/write access
  - Full speed random access



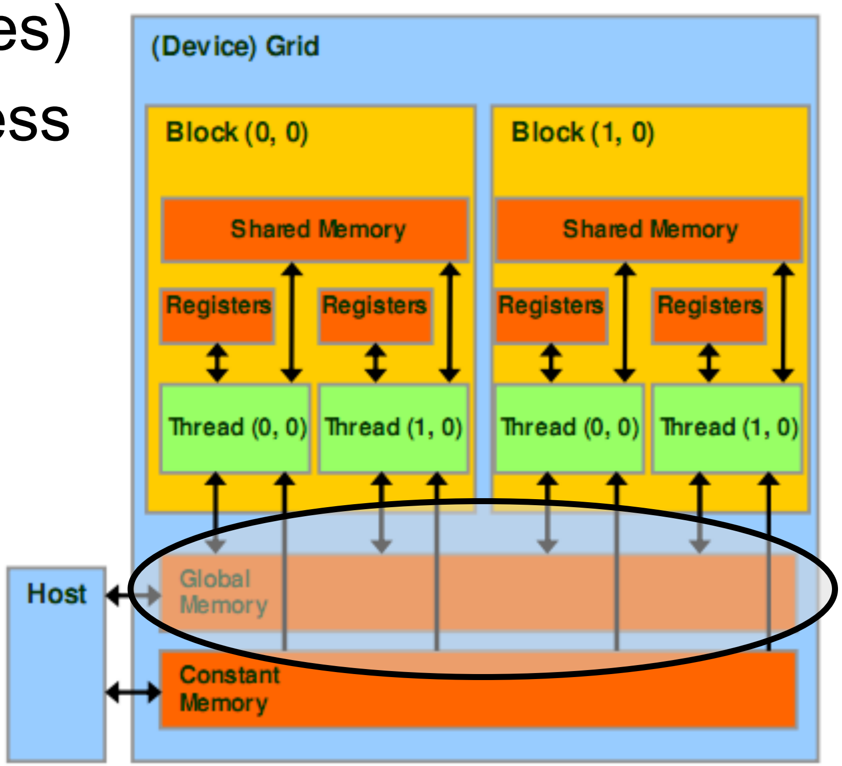
Cuda Memory Model

- Shared Memory – G80
  - Per SM
    - Up to 8 blocks
    - 16 KB
  - How many KB per block
  - 16 KB / 8 = 2 KB per block
- Example
  - If each block uses 5 KB, how many blocks can a SM host?

16 KB / 5 KB = 3 blocks per SM

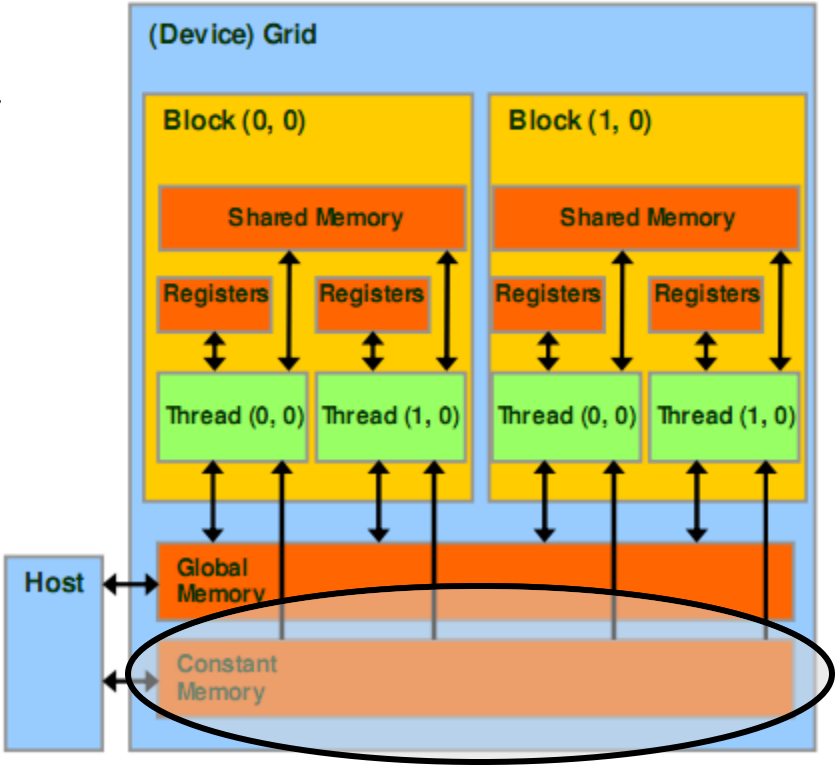
Memory Model

- Global Memory
  - Long latency (100s cycles)
  - Off-chip, read/write access
  - Random access causes performance hit
  - Host can read/write
  - GT200
    - 150 GB/s
    - Up to 4 GB
  - G80 – 86.4 GB/s



Memory Model

- Constant Memory
  - Short latency, high bandwidth, read only access when all threads access the same location
  - Stored in global memory but cached
  - Host can read/write
  - Up to 64 KB



Memory Model in CUDA

Variable Declaration	Memory	Scope	Lifetime
`Automatic variables other than arrays`	register	thread	kernel
`Automatic array variables`	local	thread	kernel
`__shared__ int sharedVar;`	shared	block	kernel
`__device__ int globalVar;`	global	grid	application
`__constant__ int constantVar;`	constant	grid	application

See Appendix B.2 in the NVIDIA CUDA C Programming Guide for more details

Memory Model

- Global and constant variables
  - Host can access with
    - `cudaGetSymbolAddress()`
    - `cudaGetSymbolSize()`
    - `cudaMemcpyToSymbol()`
    - `cudaMemcpyFromSymbol()`
  - Constants must be declared outside of a function body

```
__constant__ float constData[256]; // global scope
float data[256];
cudaMemcpyToSymbol(constData, data, sizeof(data));
cudaMemcpyFromSymbol(data, constData, sizeof(data));
```

Thread Synchronization

- Threads in a block can synchronize
  - call `__syncthreads` to create a barrier
  - A thread waits at this call until all threads in the block reach it, then all threads continue

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i + 1]);
```

Thread Synchronization

Thread 0

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

Thread 1

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

Thread 2

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

Thread 3

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

Time: 0

Thread Synchronization

Thread 0

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

Thread 1

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

Thread 2

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

Thread 3

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

Time: 1

Thread Synchronization

Thread 0

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

Thread 1

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

Thread 2

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

Thread 3

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

Threads 0 and 1 are blocked at barrier

Time: 1

Thread Synchronization

Thread 0

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

Thread 1

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

Thread 2

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

Thread 3

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

Time: 2

Thread Synchronization

Thread 0

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

Thread 1

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

Thread 2

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

Thread 3

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

Time: 3

Thread Synchronization

Thread 0

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

Thread 1

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

Thread 2

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

Thread 3

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

All threads in block have reached barrier, any thread can continue

Time: 3

 UCSB

Thread Synchronization

Thread 0

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

Thread 1

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

Thread 2

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

Thread 3

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

Time: 4

Thread Synchronization

Thread 0

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

Thread 1

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

Thread 2

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

Thread 3

```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i+1]);
```

Time: 5

Questions on Thread Synchronization

- Why is it important that execution time be similar among threads?
- Why does it only synchronize within a block?

Similar execution time – load balance

Synchronize within a block – runtime can schedule threads in any order

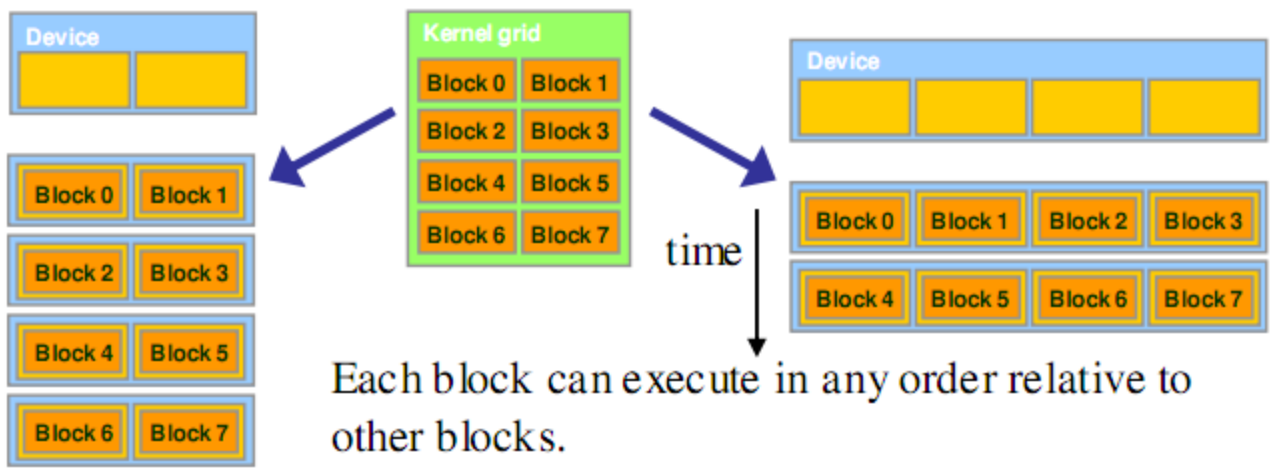


Figure 3.5 Lack of synchronization across blocks enables transparent scalability of CUDA programs

Errors in Thread Synchronization

- Can `__syncthreads()` cause a thread to hang?

```
if (someFunc())
{
  __syncthreads();
}
// ...
```

```
if (someFunc())
{
  __syncthreads();
}
else
{
  __syncthreads();
}
```

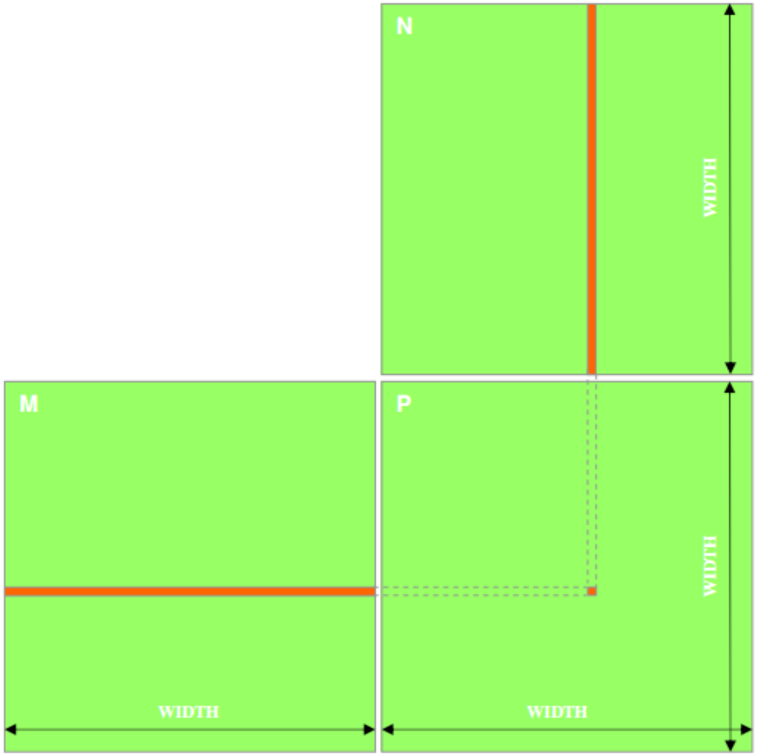
Matrix Multiplication with GPU

Matrix Multiplication P=M*N with 1D partitioning

Implements P=M*N

```
for i = 1 to n
      for j = 1 to n
            for k = 1 to n
                  P(i,j) +=M(i,k) * N(k,j)
```

$T_{i,j}$ reads Row M_i and Column N_j to produce element $P_{i,j}$



Task graph: n^2 independent tasks:

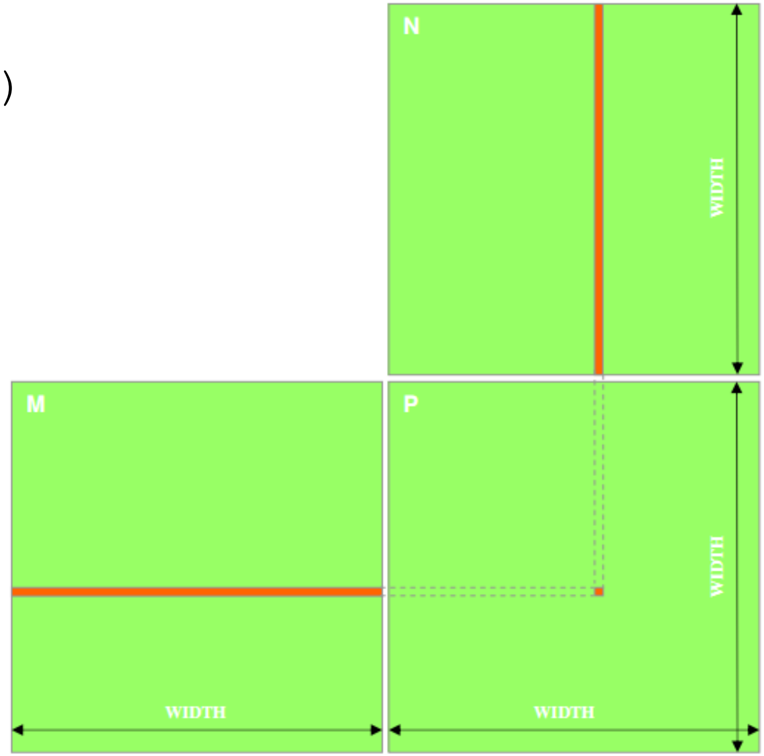
$$T_{1,1} \quad T_{1,2} \quad \cdots T_{1,n}$$
$$T_{2,1} \quad T_{2,2} \quad \cdots T_{2,n}$$
$$\cdots$$
$$T_{n,1} \quad T_{n,2} \quad \cdots T_{n,n}$$

UCSB

Matrix Multiplication P=M*N CPU code

```
void MatrixMulOnHost(float* M, float* N, float* P, int 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;
    }
}
```

Use 1D representation of 2D matrix

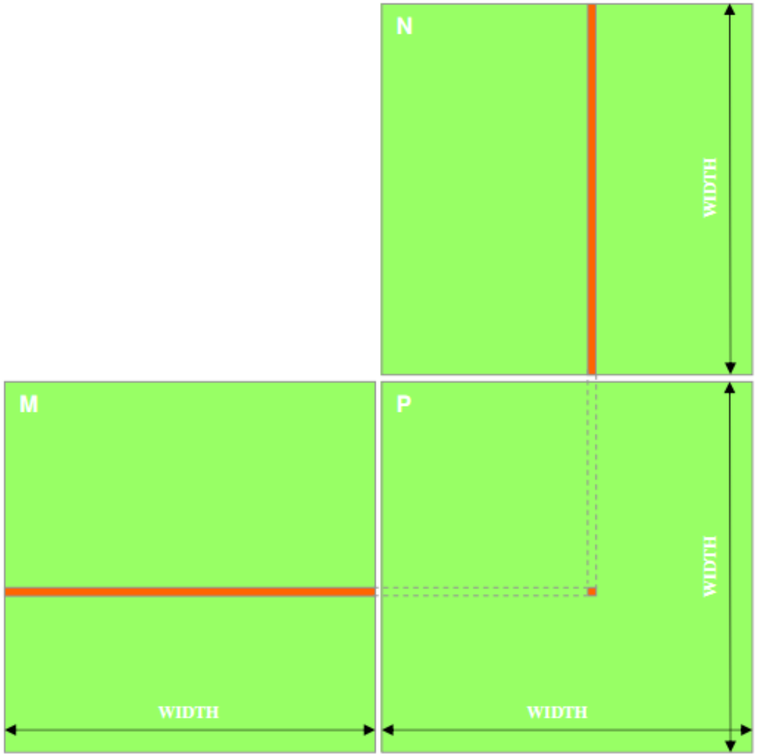


Code from: http://courses.engr.illinois.edu/ece498/al/lectures/lecture3%20cuda%20threads%20spring%202010.ppt

UCSB

Matrix Multiply: CUDA Kernel

```
__global__ void MatrixMultiplyKernel(const float* devM, const float* devN,
                                     float* devP, const int width)
{
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Initialize accumulator to 0
    float pValue = 0;

    // Multiply and add
    for (int k = 0; k < width; k++) {
        float m = devM[ty * width + k];
        float n = devN[k * width + tx];
        pValue += m * n;
    }

    // Write value to device memory - each
    // thread has unique index to write to
    devP[ty * width + tx] = pValue;
}
```



Thread $T_{y,x}$ reads Row M_y and Column N_x to produce element $P_{y,x}$

Matrix Multiply: CUDA Kernel

```
__global__ void MatrixMultiplyKernel(const float* devM, const float* devN,
                                     float* devP, const int width)
{
      int tx = threadIdx.x;
      int ty = threadIdx.y;

      // Initialize accumulator to 0
      float pValue = 0;

      // Multiply and add
      for (int k = 0; k < width; k++) {
           float m = devM[ty * width + k];
           float n = devN[k * width + tx];
           pValue += m * n;
      }

      // Write value to device memory - each thread has unique index to write to
      devP[ty * width + tx] = pValue;
}
```

Always void

Specify const buffers for read-only

No const for buffers that will be written to

Kernel specifier

UCSB

Matrix Multiply: CUDA Kernel

```
__global__ void MatrixMultiplyKernel(const float* devM, const float* devN,
                                     float* devP, const int width)
{
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Initialize accumulator to 0
    float pValue = 0;

    // Multiply and add
    for (int k = 0; k < width; k++) {
        float m = devM[ty * width + k];
        float n = devN[k * width + tx];
        pValue += m * n;
    }

    // Write value to device memory - each thread has unique index to write to
    devP[ty * width + tx] = pValue;
}
```

Accessing a Matrix, use 2D threads

Matrix Multiply: CUDA Kernel

```
__global__ void MatrixMultiplyKernel(const float* devM, const float* devN,
                                      float* devP, const int width)
{
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Initialize accumulator to 0
    float pValue = 0;

    // Multiply and add
    for (int k = 0; k < width; k++) {
        float m = devM[ty * width + k];
        float n = devN[k * width + tx];
        pValue += m * n;
    }

    // Write value to device memory - each thread has unique index to write to
    devP[ty * width + tx] = pValue;
}
```

Each kernel thread computes one location of the matrix

UCSB

Matrix Multiply: CUDA Kernel

```
__global__ void MatrixMultiplyKernel(const float* devM, const float* devN,
                                     float* devP, const int width)
{
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Initialize accumulator to 0
    float pValue = 0;

    // Multiply and add
    for (int k = 0; k < width; k++) {
        float m = devM[ty * width + k];
        float n = devN[k * width + tx];
        pValue += m * n;
    }

    // Write value to device memory - each thread has unique index to write to
    devP[ty * width + tx] = pValue;
}
```

What happened to the 2 outer loops?

Matrix Multiply: CUDA Kernel

```
__global__ void MatrixMultiplyKernel(const float* devM, const float* devN,
                                     float* devP, const int width)
{
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Initialize accumulator to 0
    float pValue = 0;

    // Multiply and add
    for (int k = 0; k < width; k++) {
        float m = devM[ty * width + k];
        float n = devN[k * width + tx];
        pValue += m * n;
    }

    // Write value to device memory - each thread has unique index to write to
    devP[ty * width + tx] = pValue;
}
```

2D Matrix but 1D access

Matrix Multiply: CUDA Kernel

```
__global__ void MatrixMultiplyKernel(const float* devM, const float* devN,
                                     float* devP, const int width)
{
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Initialize accumulator to 0
    float pValue = 0;

    // Multiply and add
    for (int k = 0; k < width; k++) {
        float m = devM[ty * width + k];
        float n = devN[k * width + tx];
        pValue += m * n;
    }

    // Write value to device memory - each thread has unique index to write to
    devP[ty * width + tx] = pValue;
}
```

Are we missing synchronization? Why?

Matrix Multiply: CUDA Kernel

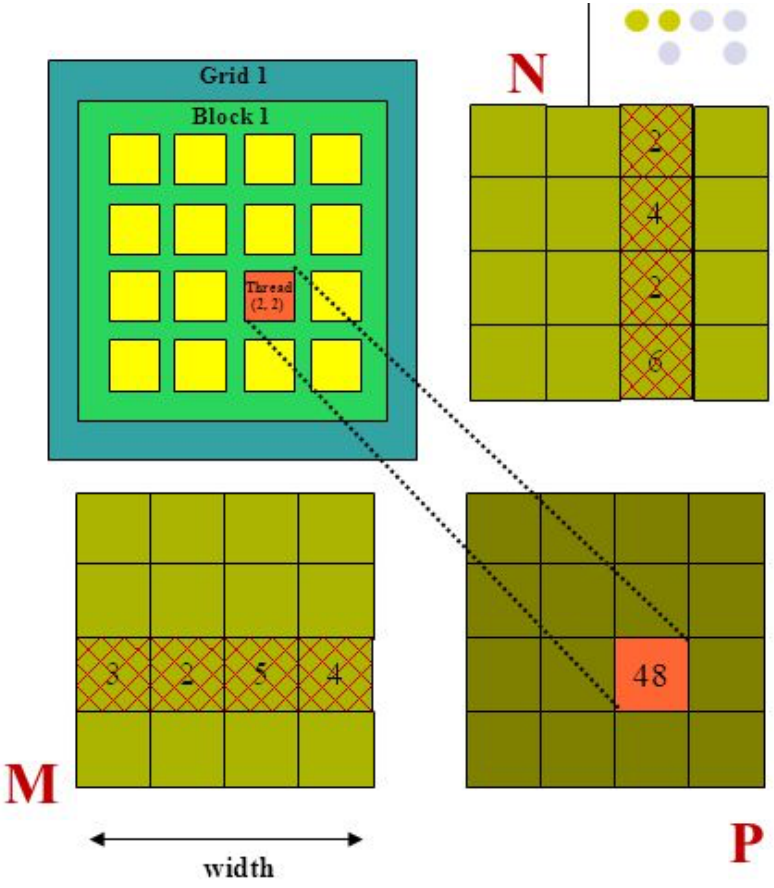
```
__global__ void MatrixMultiplyKernel(const float* devM, const float* devN,
                                     float* devP, const int width)
{
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Initialize accumulator to 0
    float pValue = 0;

    // Multiply and add
    for (int k = 0; k < width; k++) {
        float m = devM[ty * width + k];
        float n = devN[k * width + tx];
        pValue += m * n;
    }

    // Write value to device memory - each thread has unique index to write to
    devP[ty * width + tx] = pValue;
}
```

Write the value to the correct output index.

UCSB

Problems with Previous Design

- Limited matrix size
  - Only uses one thread block
  - G80 and GT200 – up to 512 threads per block
- Lots of global memory access



Matrix Multiply with 2D Tiling

- Remove size limitation
  - Break `Pd` matrix into tiles
  - Assign each tile to a block
  - Use `threadIdx` and `blockIdx` for indexing

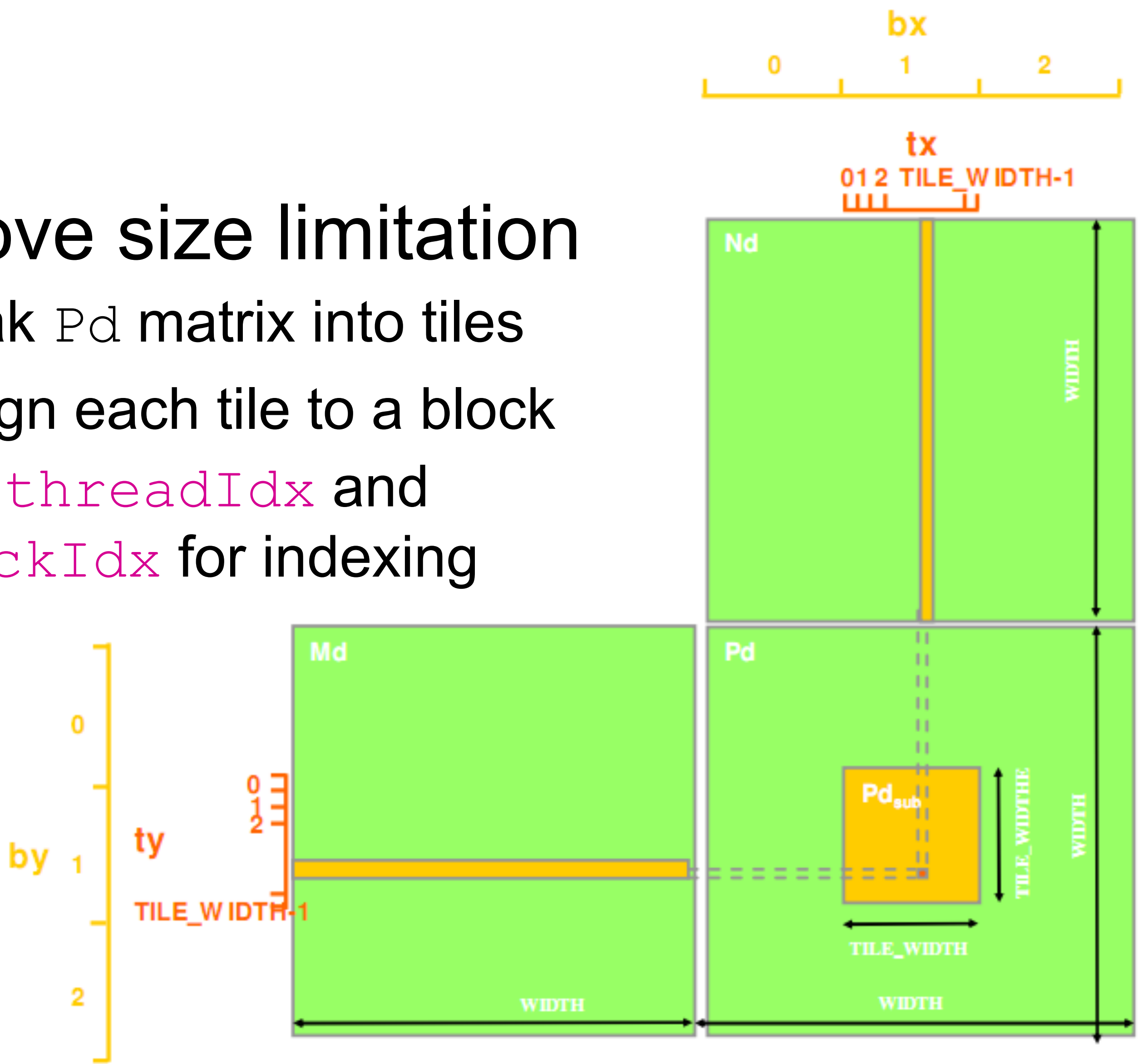


Image from http://courses.engr.illinois.edu/ece498/al/textbook/Chapter3-CudaThreadingModel.pdf

Matrix Multiply with 2D Tiling

- Example
  - Matrix: 4x4
  - `TILE_WIDTH` = 2
  - Block size: 2x2

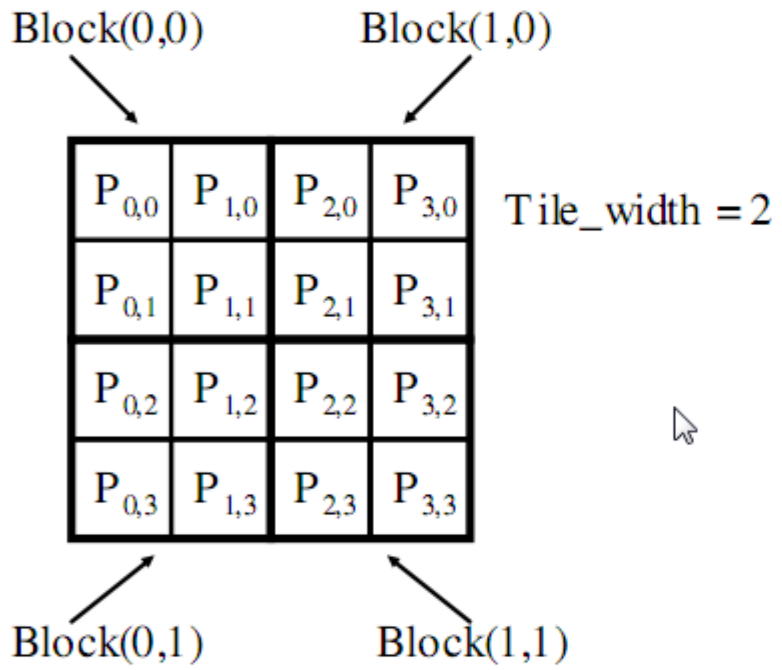


Image from http://courses.engr.illinois.edu/ece498/al/textbook/Chapter3-CudaThreadingModel.pdf

Matrix Multiply with 2D Tiling

- Example
  - Matrix: 4x4
  - `TILE_WIDTH` = 2
  - Block size: 2x2

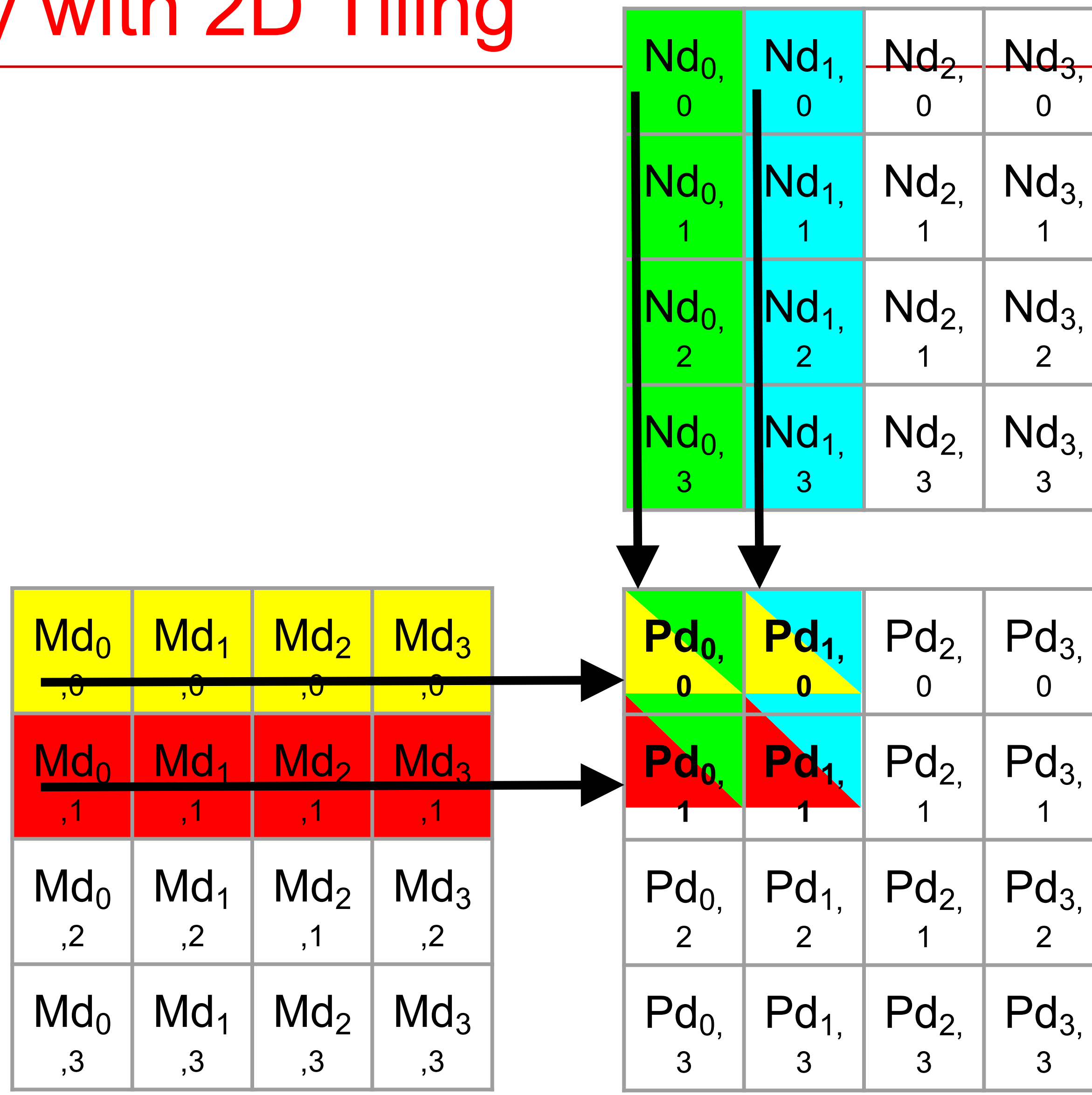


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

2D matrix partitioning: 2D Grid and 2D block

Host Declaration:

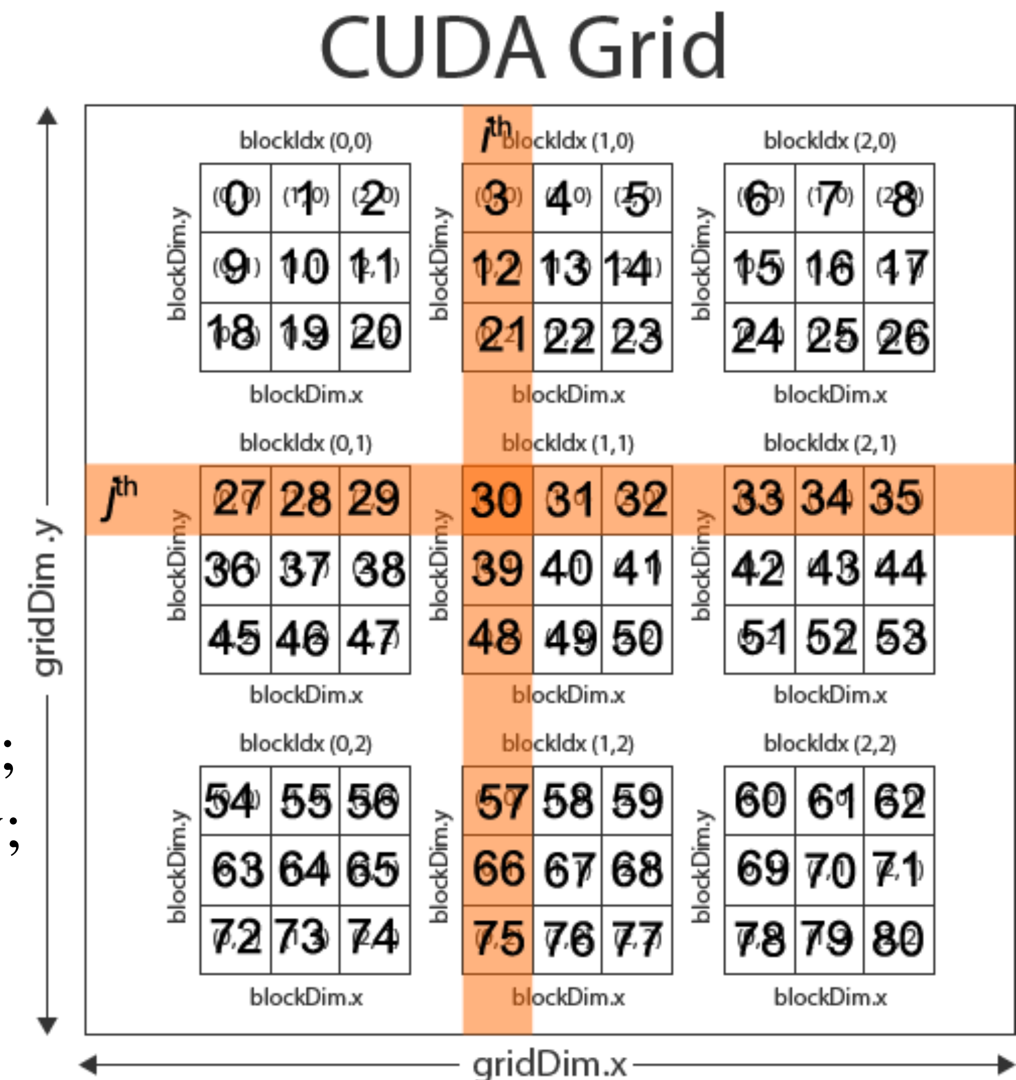
dim3 dimGrid(3, 3, 1);
dim3 dimBlock(3,3, 1);

Host Call:

`MatrixMultiplyKernel`<<< dimGrid, dimBlock>>>(…);

Device Access:

col=i=blockIdx.x * 3+threadIdx.x;
row=j=blockIdx.y * 3+threadIdx.y;



Matrix Multiply with 2D Tiling

```
__global__ void MatrixMultiplyKernel(const float* devM, const
float* devN, float* devP, const int width)
{
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    // Initialize accumulator to 0
    float pValue = 0;
    // Multiply and add
    for (int k = 0; k < width; k++) {
        float m = ;
        float n = ;
        pValue += devM[row * width + k] * devN[k * width +
        col];
    }
    // Write value to device memory -
     each thread has unique index to write to
    devP[row * width + col] = pValue;
}
```

Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

UCSB

Matrix Multiply with 2D Tiling

```
__global__ void MatrixMultiplyKernel(const float* devM,
 const float* devN,
float* devP, const int width)
{
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    // Initialize accumulator to 0
    float pValue = 0;
    // Multiply and add
    for (int k = 0; k < width; k++) {
       float m = ;
       float n = ;
       pValue += devM[row
       * width + k] * devN[k * width + col];
    }
    // Write value to device memory -
     each thread has unique index to write to
    devP[row * width + col] = pValue;
}
```

Calculate row index of P and M

Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Matrix Multiply with 2D Tiling

```
__global__ void MatrixMultiplyKernel(const float* devM,
const float* devN,
float* devP, const int width)
{
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    // Initialize accumulator to 0
    float pValue = 0;
    // Multiply and add
    for (int k = 0; k < width; k++) {
        float m = ;
        float n = ;
        pValue += devM[row* width+k]*devN[k * width+col];
    }
    // Write value to device memory -
     each thread has unique index to write to
    devP[row * width + col] = pValue;
}
```

Calculate col index of P and N

UCSB

Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Matrix Multiply with 2D Tiling

```
__global__ void MatrixMultiplyKernel(const float* devM,
 const float* devN, float* devP, const int width){
   int col = blockIdx.x * blockDim.x + threadIdx.x;
   int row = blockIdx.y * blockDim.y + threadIdx.y;
   // Initialize accumulator to 0
   float pValue = 0;

   // Multiply and add
   for (int k = 0; k < width; k++) {
      float m = ;
      float n = ;
      pValue += devM[row
      * width + k] * devN[k * width + col];
   }
   // Write value to device memory -
    each thread has unique index to write to
   devP[row * width + col] = pValue;
}
```

Each thread computes 1 element of the P sub-matrix

Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

UCSB

Matrix Multiply

```
__global__ void MatrixMultiplyKernel(const float* devM,
 const float* devN, float* devP, const int width){
   int col = blockIdx.x * blockDim.x + threadIdx.x;
   int row = blockIdx.y * blockDim.y + threadIdx.y;

   // Initialize accumulator to 0
   float pValue = 0;

   // Multiply and add
   for (int k = 0; k < width; k++) {
      pValue += devM[row
      * width + k] * devN[k * width + col];
   }

   // Write value to device memory -
    each thread has unique index to write to
   devP[row * width + col] = pValue;
}
```

Write value to devP

Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Matrix Multiply: Invoke Kernel

```
void MatrixMultiplyOnDevice(float* hostP,
const float* hostM, const float* hostN,
const int width)
{
    ………
    // Allocate P on device
  cudaMalloc((void**)&devP, sizeInBytes);
    // Call the kernel here
    // Copy P matrix from device to host
cudaMemcpy(hostP, devP, sizeInBytes, cuda
MemcpyDeviceToHost);

    ………
// Setup thread/block execution configuration
dim3 dimBlocks(TILE_WIDTH, TILE_WIDTH);
dim3 dimGrid(WIDTH / TILE_WIDTH, WIDTH / TILE_WIDTH);

// Launch the kernel
MatrixMultiplyKernel<<<dimGrid, dimBlocks>>>(devM, devN,
devP, width)
```

Performance Issues with Previous Matrix Multiplication Code

Kernel code

```
for (int k = 0; k < width; k++) {
    pValue += devM[row
    * width + k] * devN[k * width + col];
}
```

- Lots of access to global memory
- Performance limited by global memory bandwidth
  - G80 peak GFLOPS: 346.5
  - Require 1386 GB/s to achieve this
  - G80 memory bandwidth: 86.4 GB/s
    - Limits code to 21.6 GFLOPS
    - In practice, code runs at 15 GFLOPS
  - Must drastically reduce global memory access

UCSB

Matrix Multiply with block shared memory

- Each input element is read by `Width` threads
- Use shared memory to reduce global memory bandwidth

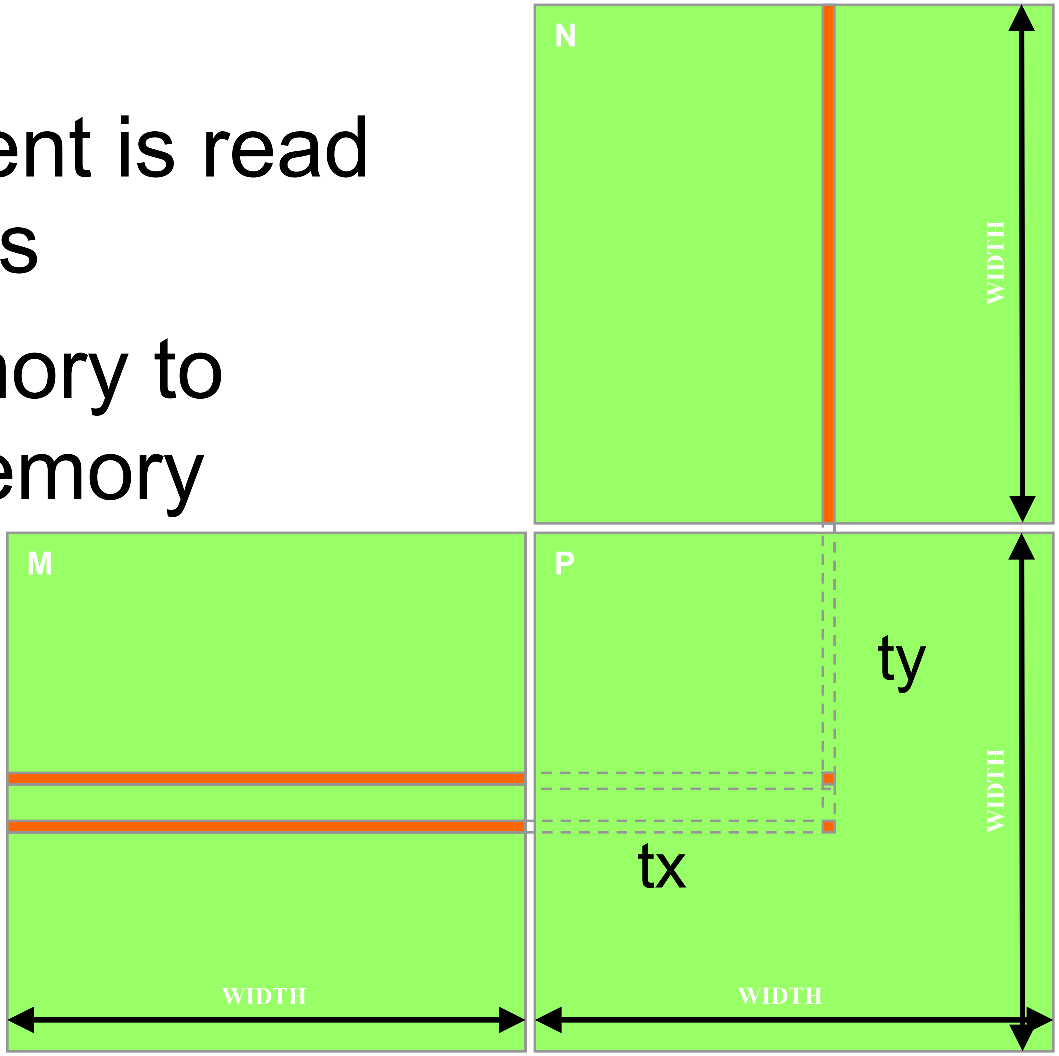


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

UCSB

Matrix Multiply

- Break kernel into phases
  - Each phase accumlates `Pd` using a subset of `Md` and `Nd`
  - Each phase has good data locality

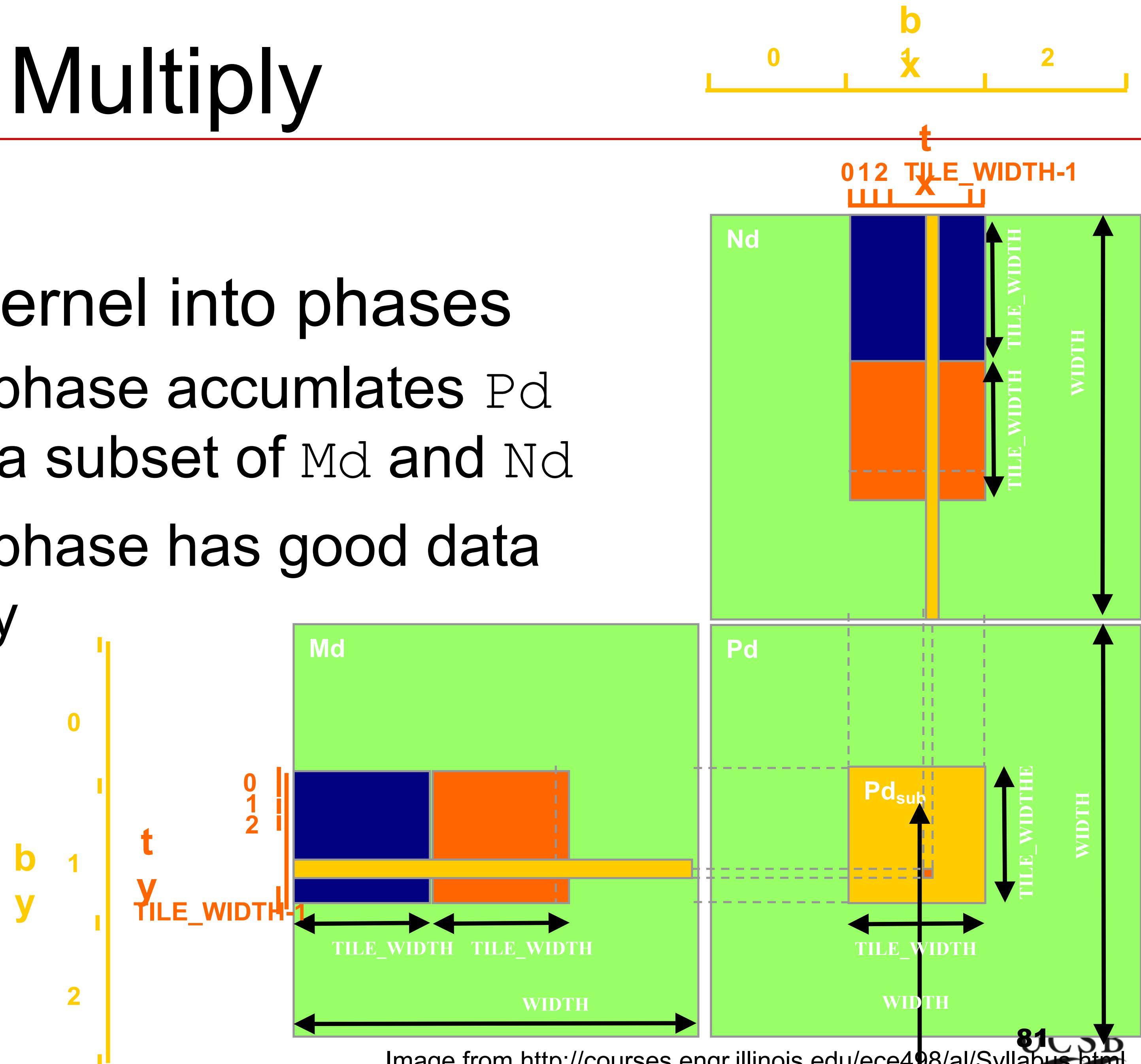
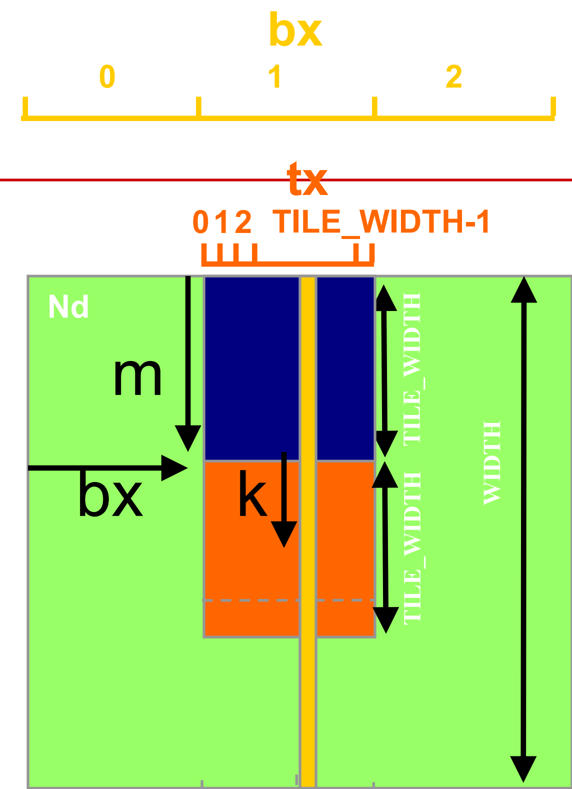


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Matrix Multiply

- Each thread
  - loads one element of `Md` and `Nd` in the tile into shared memory



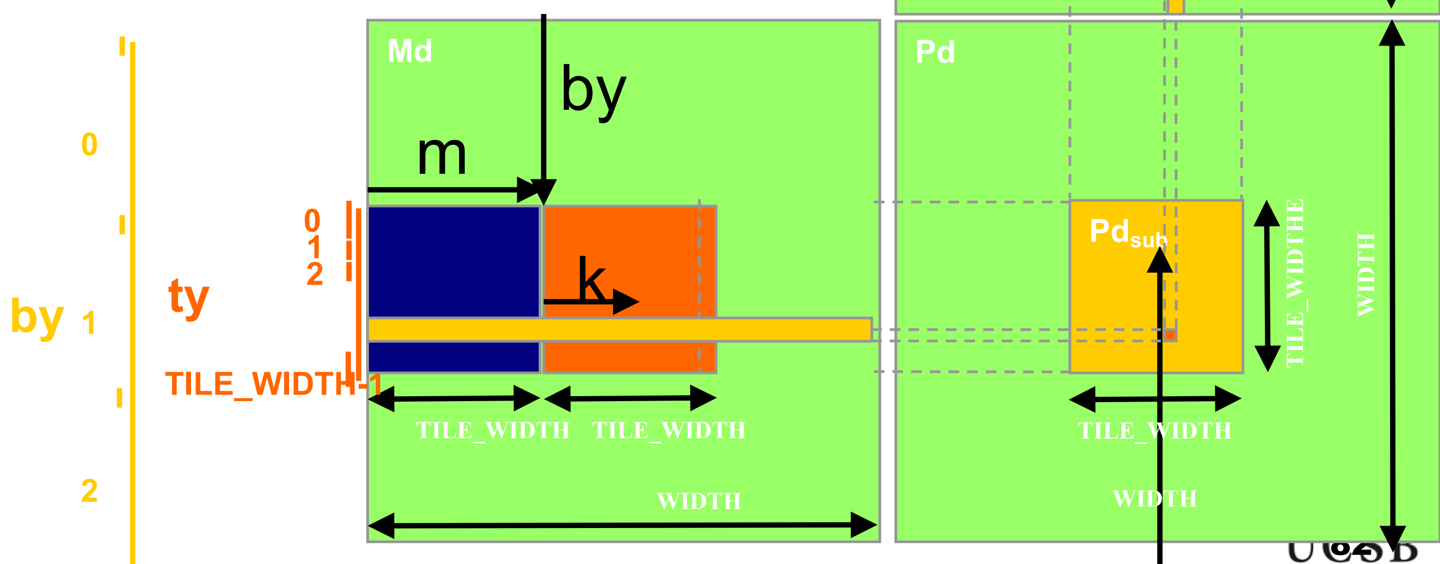


Image from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

```
__global__ void MatrixMultiplyKernel(const float* devM,
 const float* devN,float* devP, const int width){
   __shared__ float sM[TILE_WIDTH][TILE_WIDTH];
   __shared__ float sN[TILE_WIDTH][TILE_WIDTH];
   int bx = blockIdx.x; int by = blockIdx.y;
   int tx = threadIdx.x; int ty = threadIdx.y;
   int col = bx * TILE_WIDTH + bx;
   int row = by * TILE_WIDTH + ty;
   // Initialize accumulator to 0. Then multiply/add
   float pValue = 0;
   for (int m = 0; m < width / TILE_WIDTH; m++) {
     sM[ty][tx] = devM[row *width+(m*TILE_WIDTH + tx)];
     sN[ty][tx] = devN[col+(m *TILE_WIDTH+ty)* Width];
      __synchthreads();
     for (int k = 0; k < TILE_WIDTH; ++k)
            Pvalue += sM[ty][k] * sN[k][tx];
     __synchthreads();
}
devP[row * width + col] = pValue;
}
```

Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Shared memory of M and N submatrices

```
__global__ void MatrixMultiplyKernel(const float* devM,
 const float* devN,float* devP, const int width){
   __shared__ float sM[TILE_WIDTH][TILE_WIDTH];
   __shared__ float sN[TILE_WIDTH][TILE_WIDTH];
   int bx = blockIdx.x; int by = blockIdx.y;
   int tx = threadIdx.x; int ty = threadIdx.y;
   int col = bx * TILE_WIDTH + bx;
   int row = by * TILE_WIDTH + ty;
   // Initialize accumulator to 0. Then multiply/add
   float pValue = 0;
   for (int m = 0; m < width / TILE_WIDTH; m++) {
     sM[ty][tx] = devM[row *width+(m*TILE_WIDTH + tx)];
     sN[ty][tx] = devN[col+(m *TILE_WIDTH+ty)* Width];
     __synchthreads();
     for (int k = 0; k < TILE_WIDTH; ++k)
            Pvalue += sM[ty][k] * sN[k][tx];
     __synchthreads();
  }
  devP[row * width + col] = pValue;
}
```

Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Compute row/column index

```
__global__ void MatrixMultiplyKernel(const float* devM,
 const float* devN,float* devP, const int width){
   __shared__ float sM[TILE_WIDTH][TILE_WIDTH];
   __shared__ float sN[TILE_WIDTH][TILE_WIDTH];
   int bx = blockIdx.x; int by = blockIdx.y;
   int tx = threadIdx.x; int ty = threadIdx.y;
   int col = bx * TILE_WIDTH + bx;
   int row = by * TILE_WIDTH + ty;
   // Initialize accumulator to 0. Then multiply/add
   float pValue = 0;
   for (int m = 0; m < width / TILE_WIDTH; m++) {
     sM[ty][tx] = devM[row *width+(m*TILE_WIDTH + tx)];
     sN[ty][tx] = devN[col+(m *TILE_WIDTH+ty)* Width];
     __synchthreads();
     for (int k = 0; k < TILE_WIDTH; ++k)
           Pvalue += sM[ty][k] * sN[k][tx];
     __synchthreads();
     }
    devP[row * width + col] = pValue;
}
```

Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Bring one element from each devM and devN into shared memory

```
__global__ void MatrixMultiplyKernel(const float* devM,
 const float* devN,float* devP, const int width){
   __shared__ float sM[TILE_WIDTH][TILE_WIDTH];
   __shared__ float sN[TILE_WIDTH][TILE_WIDTH];
   int bx = blockIdx.x; int by = blockIdx.y;
   int tx = threadIdx.x; int ty = threadIdx.y;
   int col = bx * TILE_WIDTH + bx;
   int row = by * TILE_WIDTH + ty;
   // Initialize accumulator to 0. Then multiply/add
   float pValue = 0;
   for (int m = 0; m < width / TILE_WIDTH; m++) {
     sM[ty][tx] = devM[row *width+(m*TILE_WIDTH + tx)];
     sN[ty][tx] = devN[col+(m *TILE_WIDTH+ty)* Width];
     __synchthreads();
     for (int k = 0; k < TILE_WIDTH; ++k)
            Pvalue += sM[ty][k] * sN[k][tx];
     __synchthreads();
    }
   devP[row * width + col] = pValue;
}
```

Wait for every thread in the block, ie. wait for tile to be in submatrix

```
__global__ void MatrixMultiplyKernel(const float* devM,
 const float* devN,float* devP, const int width){
    __shared__ float sM[TILE_WIDTH][TILE_WIDTH];
    __shared__ float sN[TILE_WIDTH][TILE_WIDTH];
    int bx = blockIdx.x; int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;
    int col = bx * TILE_WIDTH + bx;
    int row = by * TILE_WIDTH + ty;
    // Initialize accumulator to 0. Then multiply/add
    float pValue = 0;
    for (int m = 0; m < width / TILE_WIDTH; m++) {
      sM[ty][tx] = devM[row *width+(m*TILE_WIDTH + tx)];
      sN[ty][tx] = devN[col+(m *TILE_WIDTH+ty)* Width];
      __syncthreads();
      for (int k = 0; k < TILE_WIDTH; ++k)
             Pvalue += sM[ty][k] * sN[k][tx];
      __syncthreads();
     }
     devP[row * width + col] = pValue;
}
```

Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

UCSB

Accumulate subset of dot product. After that all threads move on to next phase of shared memory reading and dot product accunulation

```
__global__ void MatrixMultiplyKernel(const float* devM,
 const float* devN,float* devP, const int width){
   __shared__ float sM[TILE_WIDTH][TILE_WIDTH];
   __shared__ float sN[TILE_WIDTH][TILE_WIDTH];
   int bx = blockIdx.x; int by = blockIdx.y;
   int tx = threadIdx.x; int ty = threadIdx.y;
   int col = bx * TILE_WIDTH + bx;
   int row = by * TILE_WIDTH + ty;
   // Initialize accumulator to 0. Then multiply/add
   float pValue = 0;
   for (int m = 0; m < width / TILE_WIDTH; m++) {
     sM[ty][tx] = devM[row *width+(m*TILE_WIDTH + tx)];
     sN[ty][tx] = devN[col+(m *TILE_WIDTH+ty)* Width];
     __synchthreads();
     for (int k = 0; k < TILE_WIDTH; ++k)
           Pvalue += sM[ty][k] * sN[k][tx];
     __synchthreads();
   }
   devP[row * width + col] = pValue;
}
```


UCSB

Write final answer to global memory

```
__global__ void MatrixMultiplyKernel(const float* devM,
 const float* devN,float* devP, const int width){
   __shared__ float sM[TILE_WIDTH][TILE_WIDTH];
   __shared__ float sN[TILE_WIDTH][TILE_WIDTH];
   int bx = blockIdx.x; int by = blockIdx.y;
   int tx = threadIdx.x; int ty = threadIdx.y;
   int col = bx * TILE_WIDTH + bx;
   int row = by * TILE_WIDTH + ty;
   // Initialize accumulator to 0. Then multiply/add
   float pValue = 0;
   for (int m = 0; m < width / TILE_WIDTH; m++) {
     sM[ty][tx] = devM[row *width+(m*TILE_WIDTH + tx)];
     sN[ty][tx] = devN[col+(m *TILE_WIDTH+ty)* Width];
     __synchthreads();
     for (int k = 0; k < TILE_WIDTH; ++k)
           Pvalue += sM[ty][k] * sN[k][tx];
     __synchthreads();
     }
     devP[row * width + col] = pValue;
}
```

Code from http://courses.engr.illinois.edu/ece498/al/Syllabus.html

UCSB

How do you pick `TILE_WIDTH`?

- How can it be too large?
  - By exceeding the maximum number of threads/block
    - G80 and GT200 – 512
    - Fermi & newer – 1024
  - By exceeding the shared memory limitations
    - G80: 16KB per SM and up to 8 blocks per SM
      - 2 KB per block
      - 1 KB for `Nds` and 1 KB for `Mds` (16 * 16 * 4)
      - `TILE_WIDTH` = 16
      - A larger `TILE_WIDTH` will result in less blocks

Size Considerations in G80

- Each thread block should have many threads
  - `TILE_WIDTH` of 16 gives 16*16 = 256 threads
- There should be many thread blocks
  - A 1024*1024 matrix with 16x16 tiles gives 64*64 = 4K Thread Blocks
- Each thread block perform 2*256 = 512 float loads from global memory for 256 * (2*16) = 8K mul/add operations.
  - Memory bandwidth no longer a limiting factor

Shared memory tiling benefits

- Reduces global memory access by a factor of `TILE_WIDTH`
  - 16x16 tiles reduces by a factor of 16
- G80
  - Now new code supports 345.6 GFLOPS
  - Close to maximum of 346.5 GFLOPS