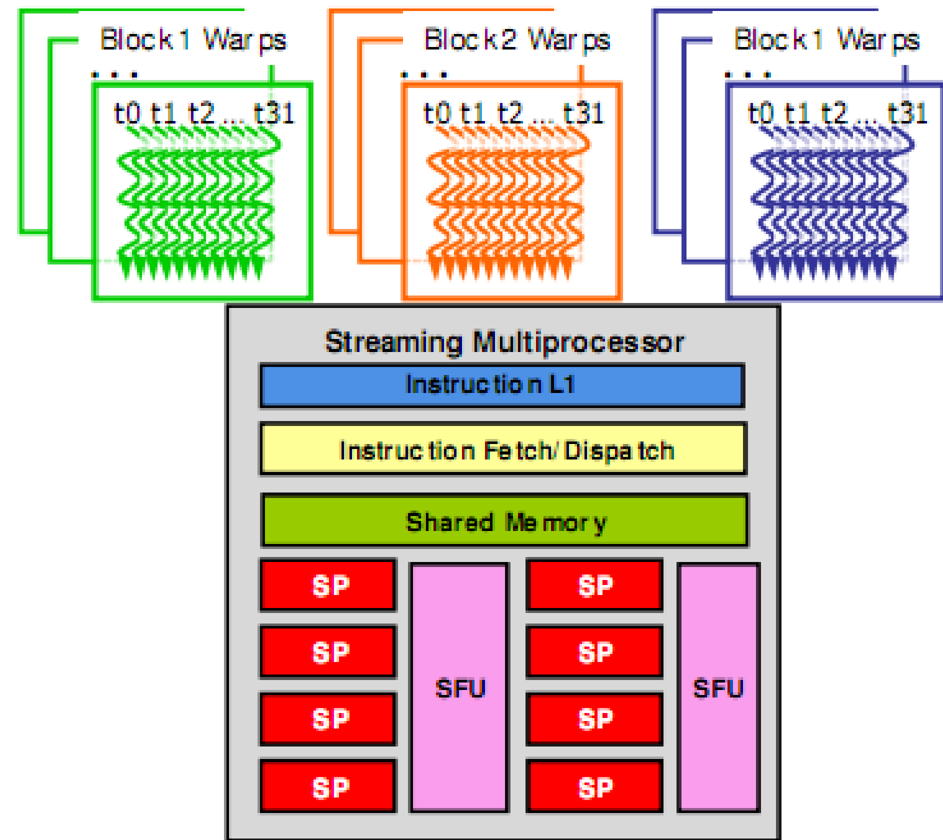
Advanced Topics on GPU Programming

CS140 Tao Yang

UCSB

Scheduling of Threads

- GPU executes threads block by block.
  - Assign a block to an available SM
    - Select a warp of threads (typically 32 threads) within the same block to run.
    - Execute threads warp by warp from the same block
  - With more SMs, blocks of threads can run in parallel also.

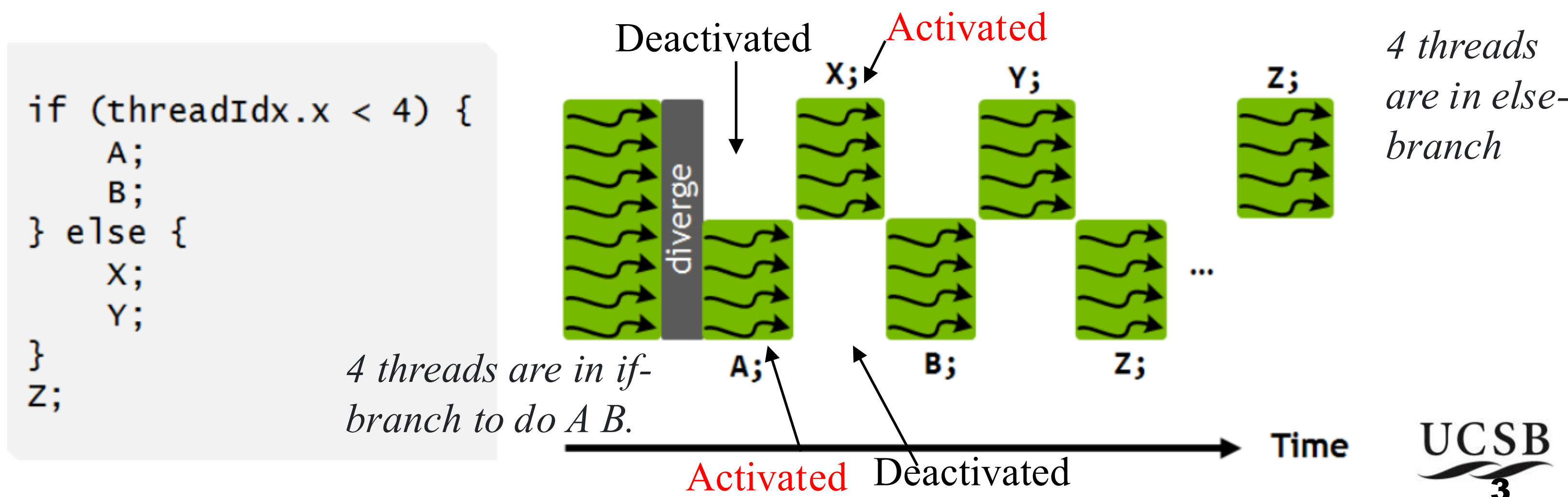


- Example: Warps for three blocks scheduled on the same SM.

UCSB

SIMT Execution in a Warp

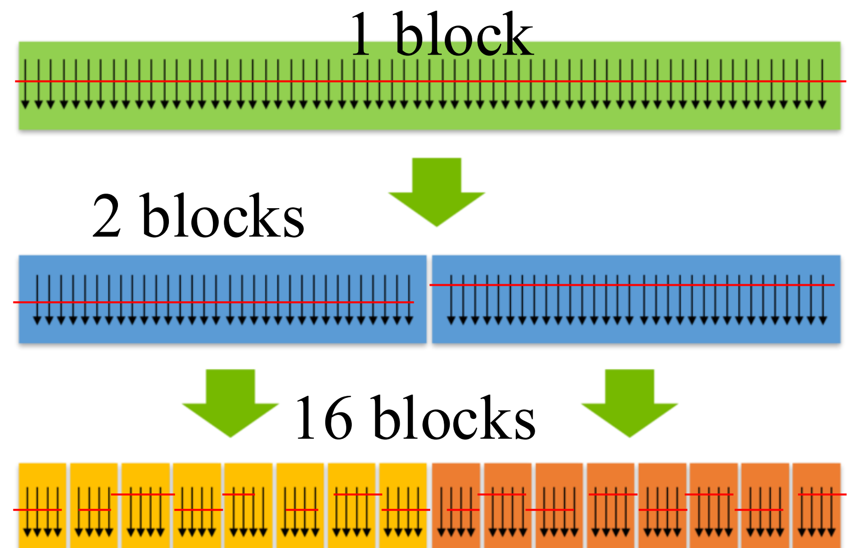
- **Pure SIMD with single program counter (PC):** a single instruction acts upon *all* the data in *exactly* the same way.
- **CUDA**: 32 threads in a warp are executed in **SIMT (Single Instruction Multiple Threads)** style on 32 cores
  - SIMT: Generalize SIMD to support branching: all *activated* threads in if-branch executes the same instruction while deactivated threads in else-branch do nothing.
  - Separate PC per thread now while scheduler tries to group threads at the same PC into a single SIMT instruction for high throughput



Thread Synchronization

- Threads **within 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

```
void kernel_fun(){
…
  __syncthreads();
}
```



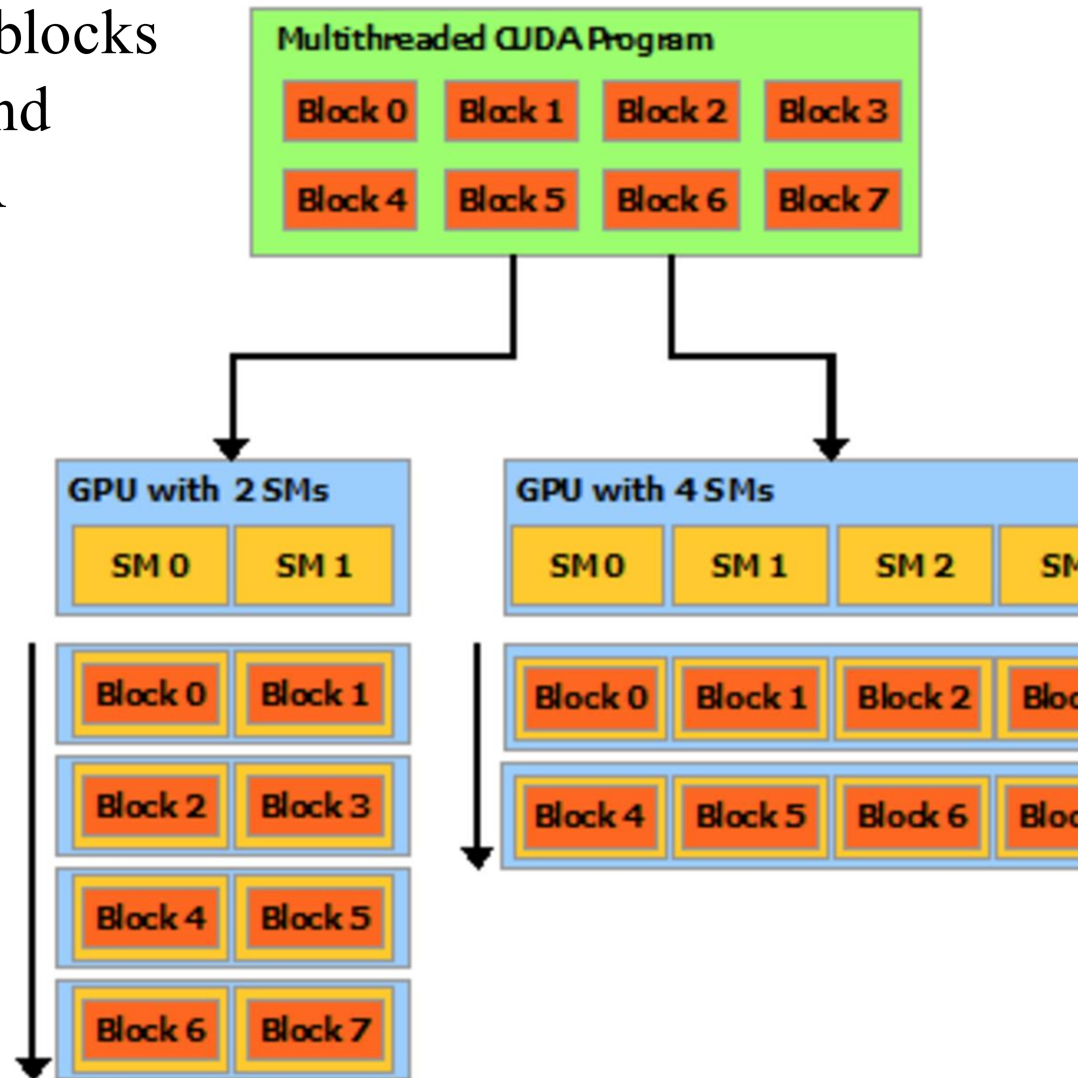
- There is no global barrier across different blocks of threads during execution.
- Atomic primitives for managing a critical section among threads are available

UCSB

Why synchronize only within a block?

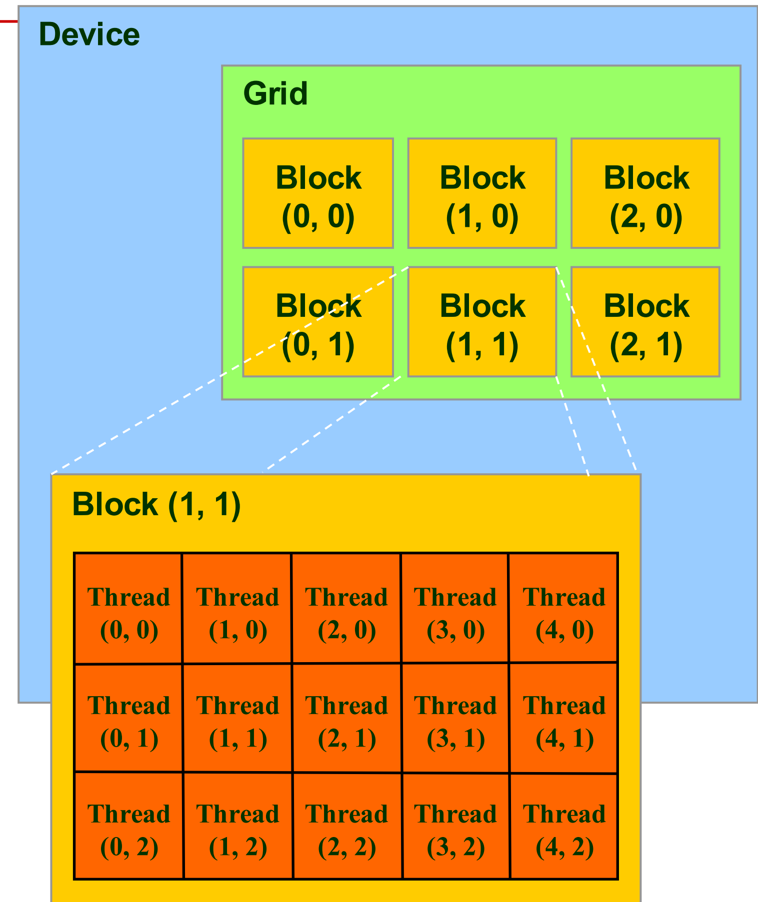
Lack of synchronization across blocks enables scheduling flexibility, and transparent scalability of CUDA programs.

- Nvidia runtime system can schedule blocks of threads in any order on multiple SMs



Revisit Thread Naming

- kernel<<<NoBlocks, NoThreads>>>( )
  - NoBlocks: no of blocks.
    - Use an integer as1D, or 2D or 3D.
  - NoThreads: no of threads per block.
    - Use an integer as1D, or 2D or 3D.
- Why multiple blocks?
  - Total number of threads per block is limited (512 or 1024).
- Multi-dimension naming simplifies coding for multidimensional data



Courtesy: NDVIA

UCSB

Thread Naming: Examples

```
kernel<<<1, 512>>>( )
```

- 1 block. 512 threads

```
kernel<<<10, 512>>>( )
```

- 10 blocks. Each block has 512 threads

```
dim3 dimBlock(3,3,1);
kernel<<<10, dimBlock>>>( );
```

- Threads within each block use 2D naming
- 10 blocks. Each block has 3x3 threads
- A 1D grid of 10 blocks and each block has 2D threads

More complex naming →

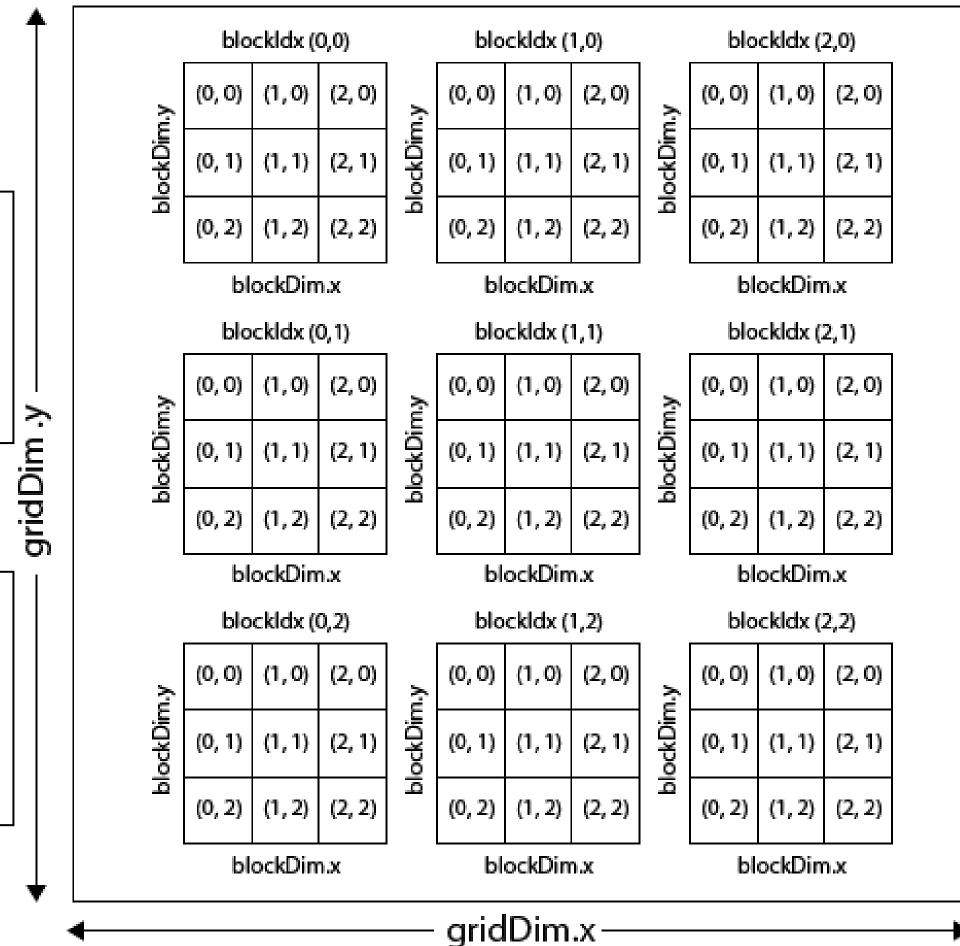
```
kernel<<<2DGrid, 2DBlock>>>();
kernel <<<3DGrid, 3DBlock>>>();
```

Thread Naming with 2D Grid and 2D block

2D grid of blocks.
2D naming for each block

```
dim3 Grid(3, 3, 1);
dim3 Block(3,3, 1);
Kernel<<<Grid, Block>>>();
```

```
dim3 Grid(9, 1, 1);
dim3 Block(9,1, 1);
Kernel<<<Grid, Block>>>();
```




```
# of threads same as Kernel<<<9,9 >>>();
```

How to Locate a Thread in a grid/block name space?

- Use special variables on dimension size and my position within each block
- **Dimension size for <<<Grid, Block>>>**
  - gridDim.x, gridDim.y, gridDim.z;
  - blockDim.x, blockDim.y, blockDim.z;
- **Position in grid/block space:**
  - Block position in grid: blockIdx.x, blockIdx.y, blockIdx.z
  - Thread position within a block: threadIdx.x, threadIdx.y, threadIdx.z

```
Kernel<<<9,10 >>>();
```

Same as:

```
dim3 Grid(9, 1, 1);
dim3 Block(10,1, 1);
Kernel<<<Grid,
Block>>>();
```

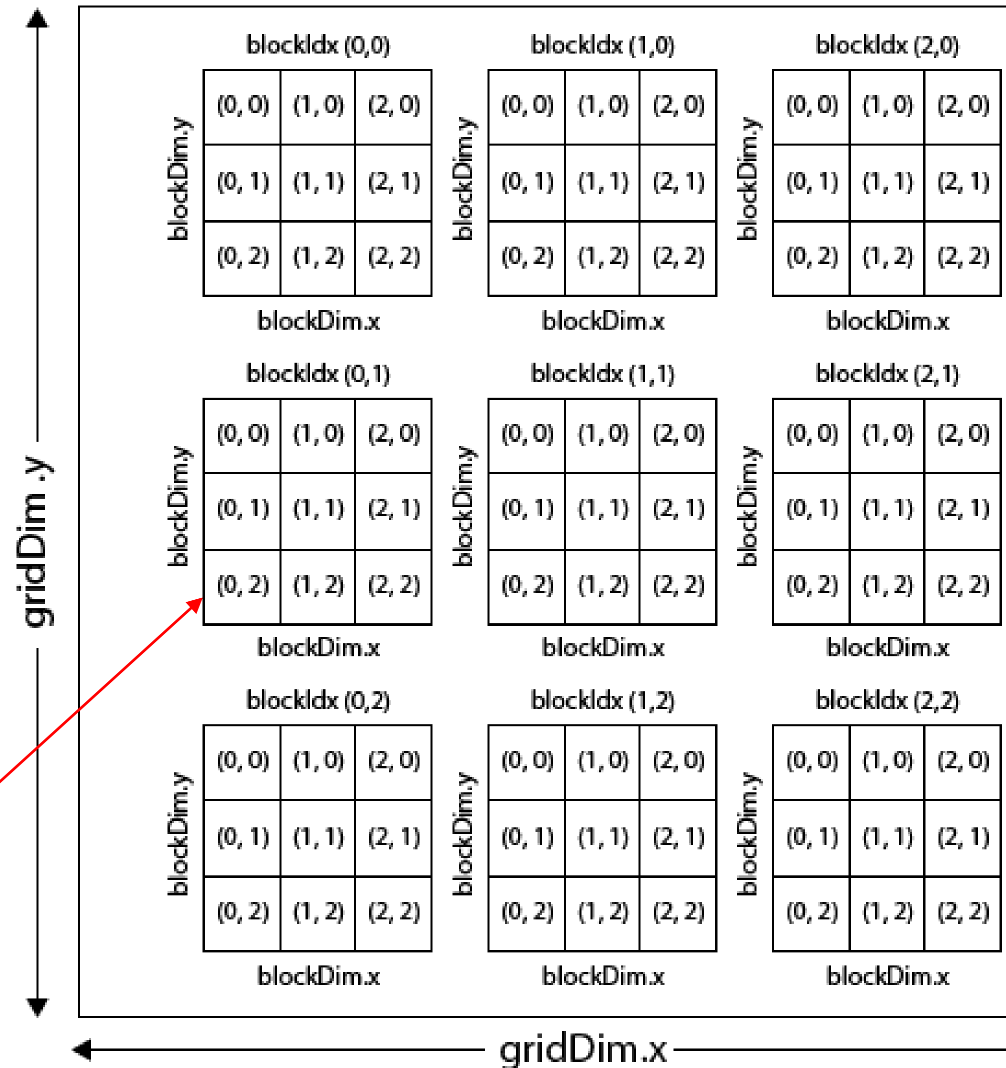
```
gridDim.x=9
gridDim.y=1
gridDim.z=1
blockDim.x=10
blockDim.y=1
blockDim.z=1
```

Example with 2D Grid and 2D Block of Threads

```
dim3 Grid(3, 3, 1);
dim3 Block(3,3, 1);
Kernel<<<Grid,
Block>>>();
```

gridDim.x=3
gridDim.y=3
gridDim.z=1
blockDim.x=3
blockDim.y=3
blockDim.z=1

blockIdx.x=0
blockIdx.y=1
blockIdx.z=0
threadIdx.x=0
threadIdx.y=2
threadIdx.z=0



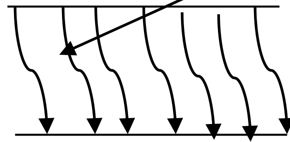
How to Compute My Rank with a Linearized formula?

Kernel<<<1,512 >>>();

me = threadIdx.x;

gridDim.x=1
gridDim.y=1
gridDim.z=1
blockDim.x=512
blockDim.y=1
blockDim.z=1

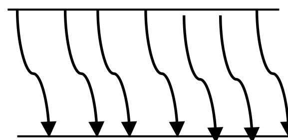
1 block of threads

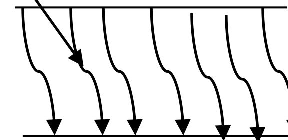


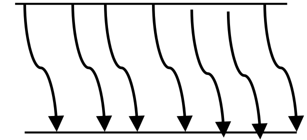
blockIdx.x=0
blockIdx.y=0
blockIdx.z=0
threadIdx.x=1
threadIdx.y=0
threadIdx.z=0

Kernel<<<9,10 >>>();
gridDim.x=9
blockDim.x=10

me = blockIdx.x * blockDim.x + threadIdx.x;







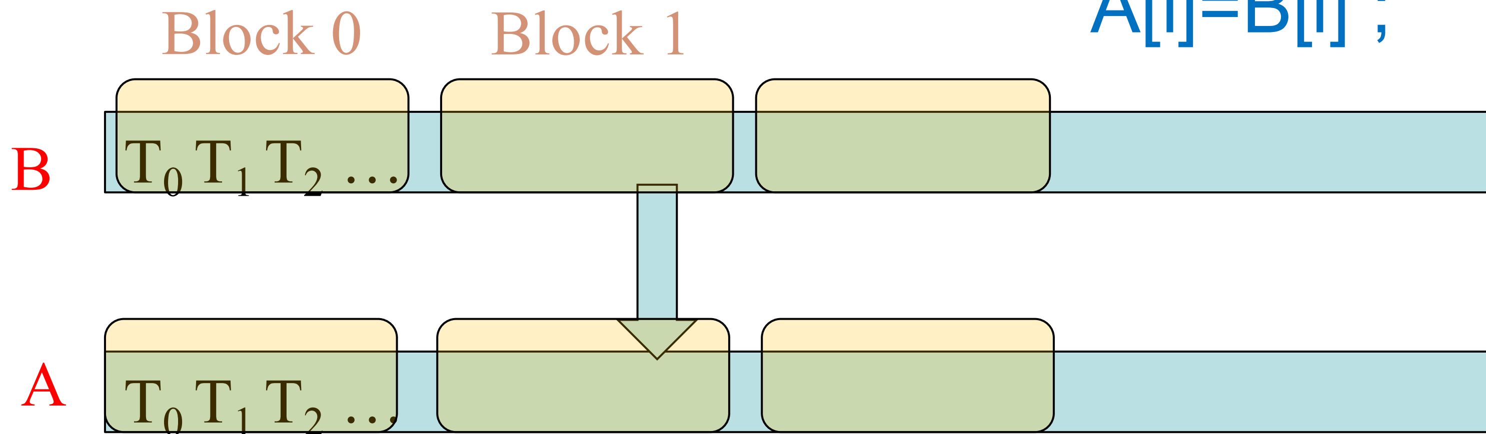
Block 0 Block 1 Block 2

blockIdx.x=1
threadIdx.x=1

UCSB

Parallelize with 5120 threads:

for(i=0;i<5120;i++)
A[i]=B[i] ;



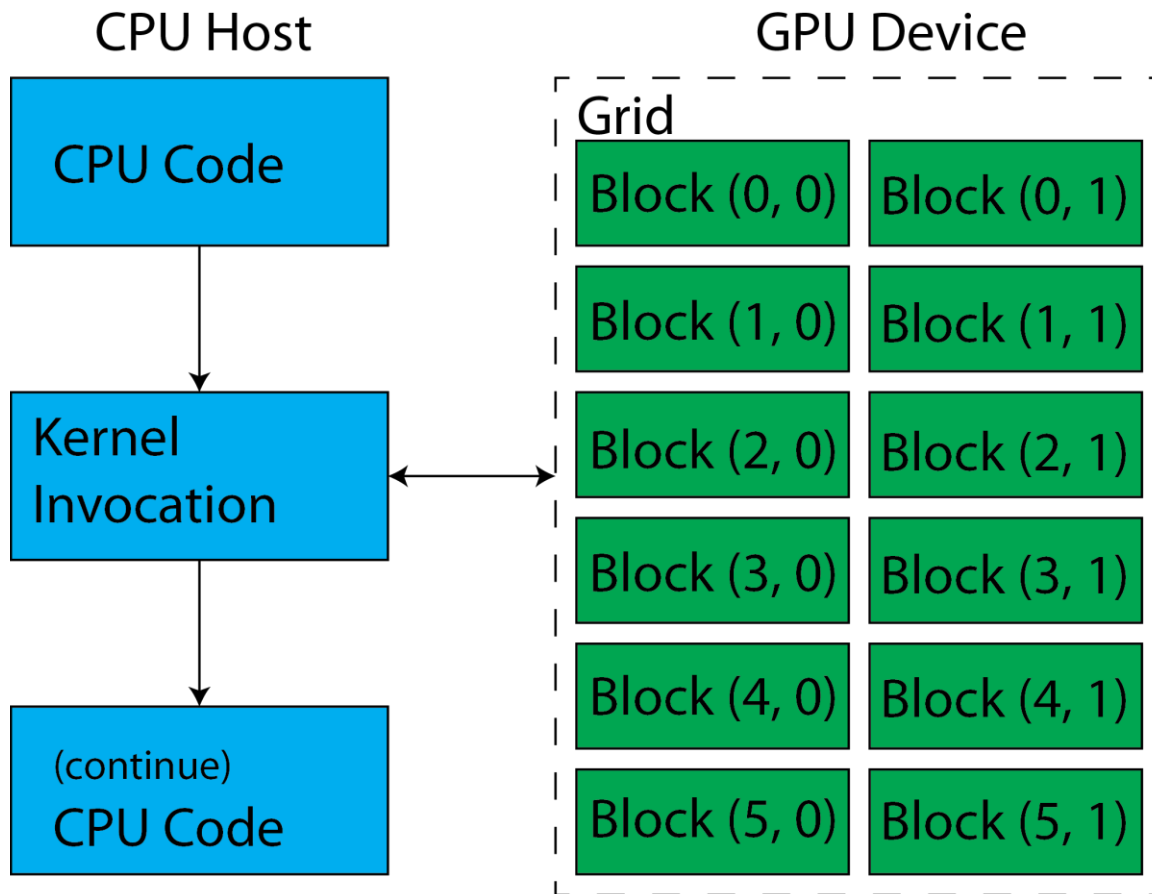

```
Host:
kernel <<<10,512>>>(A,B);
```

```
__global__ kernel(float *A, float *B){
  int me = blockIdx.x *blockDim.x +threadIdx.x;

  A[me]= B[me];
}
```

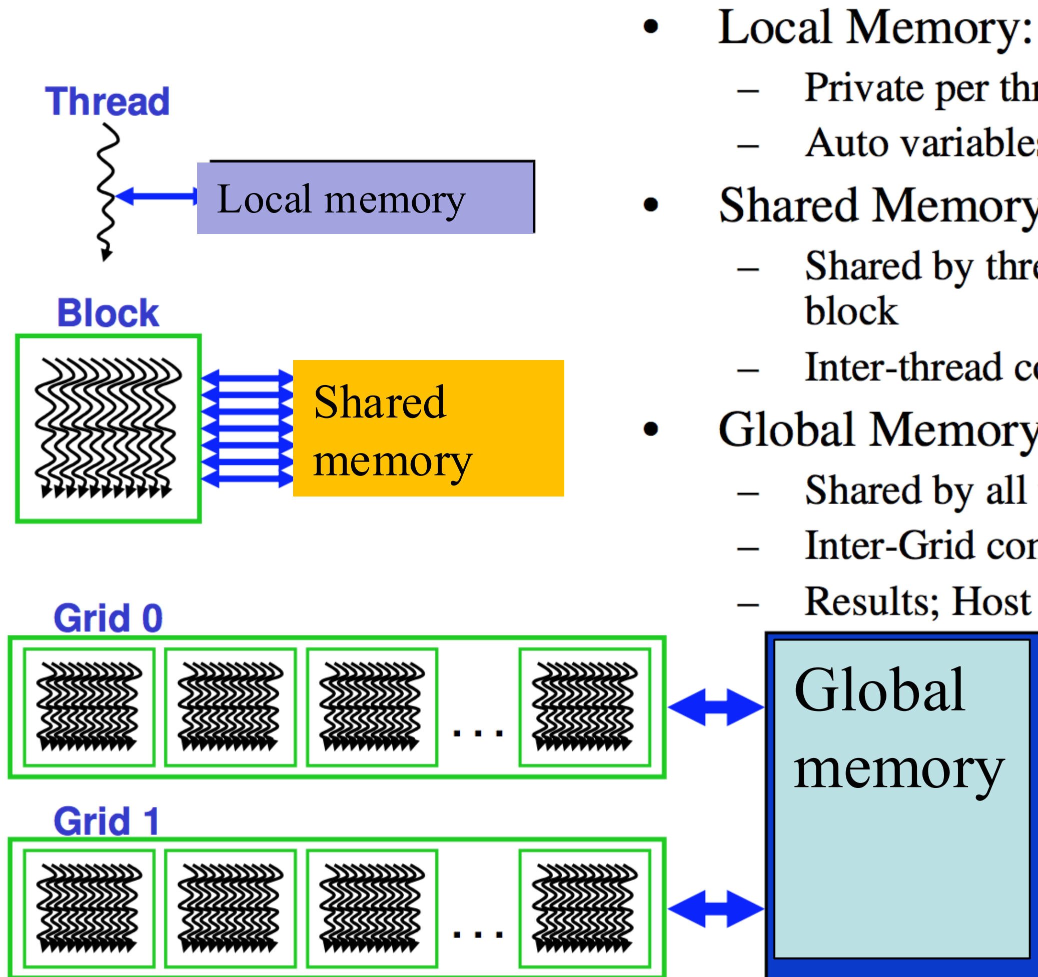
Every thread runs one assignment

Summary



- How to fork and run blocks of threads that call a kernel function.
- Where do threads access memory?

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

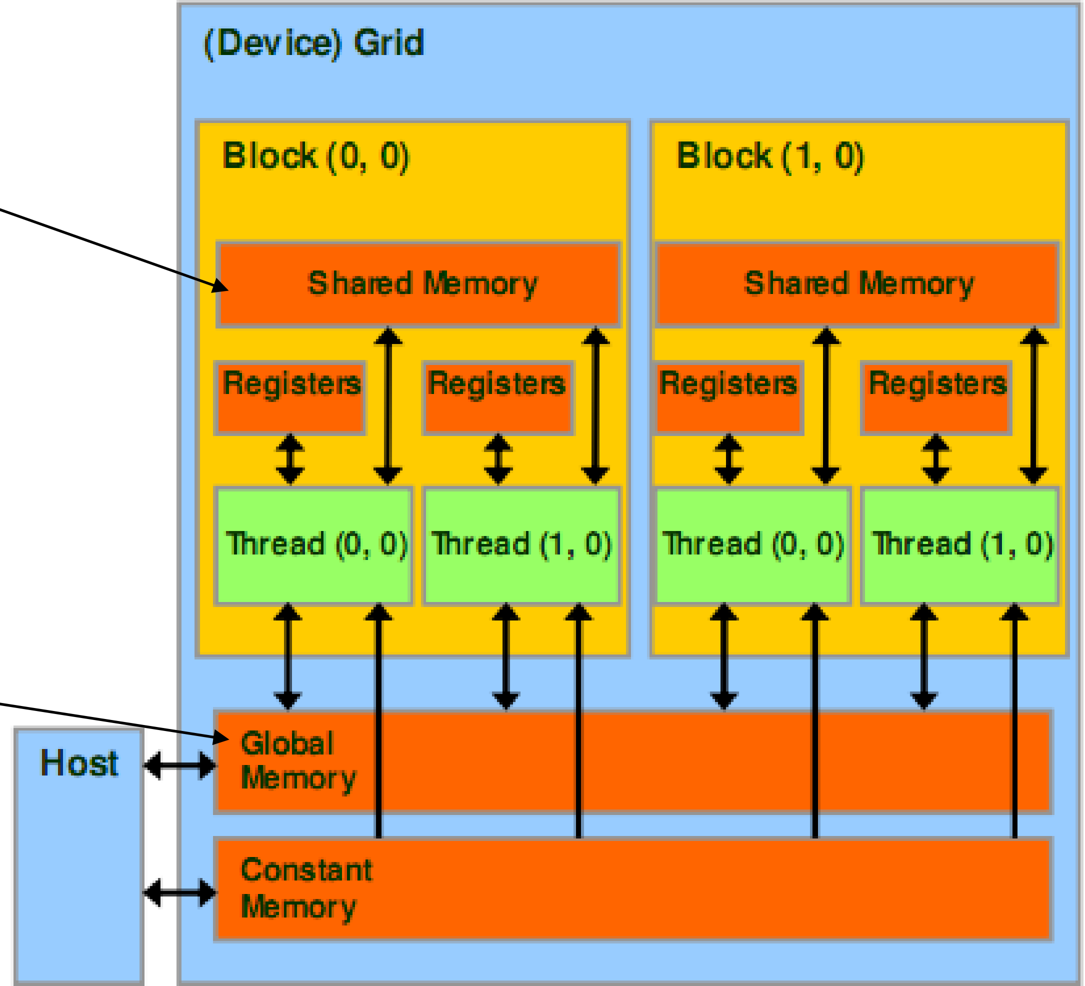
Access Speed

- 1st place: Registers
- 2nd place: Shared Memory
- 3rd place: Constant Memory
- 4th: Texture Memory
- Last place: Local Memory and Global Memory

Memory Model

- **Special Registers**
  - for built-in variables: threadIdx, blockIdx, blockDim, etc

- **Shared memory per block** in one SM
- Fast (~100x faster than global memory) limited size
- V100: 96KB/GPU



- **Global memory** shared among all thread blocks.
- Slower, bigger
  - ~100s cycles
- V100: 32GB/GPU

Size Impact of Shared Memory within SM

- Size of shared memory affects the number of thread blocks running simultaneously
  - E.g. Shared memory in a streaming multiprocessor in Nvidia G80 has 16KB
  - To run 8 blocks of threads, how much shared memory(SM) available per thread block?
    - 16 KB / 8 = 2 KB per thread block
  - If each thread block uses 5 KB, how many blocks can an SM host?

16 KB / 5 KB = 3 blocks per SM

UCSB

Memory Hierarchy in CUDA C

Variable Declaration	Memory location	Scope	Lifetime
`Automatic variables other than arrays`	Register On-chip	1 thread	Thread
`Automatic array variables`	Local Off-chip	1 thread	Thread
`__shared__ int sharedVar;`	Shared On-chip	Threads in a thread block	Block of threads
`__device__ int globalVar;`	Global Off-chip	All threads +host	application
`__constant__ int constantVar;`	Constant Off-chip	All threads +host	application

- Constant memory/ texture memory
  - Short latency, high bandwidth, read only access by threads
  - Stored in global memory but cached in L1/L2

Example of Using On-chip Shared Memory

```
int main(void) {
    const int n = 64;
    int a[n], b[n];
    for (int i = 0; i < n; i++) {
        a[i] = i;
    }
    int *d;
    cudaMalloc(&d, n * sizeof(int));
    cudaMemcpy(d, a, n*sizeof(int), cudaMemcpyHostToDevice);
    AddReverse<<<1,n>>>(d, n);
    cudaMemcpy(b, d, n*sizeof(int), cudaMemcpyDeviceToHost);
}
```

```
For i=0; i<64; i++)
    b[i]=a[i]+a[63-i];
```

UCSB

Ex: Kernel function uses fast shared memory

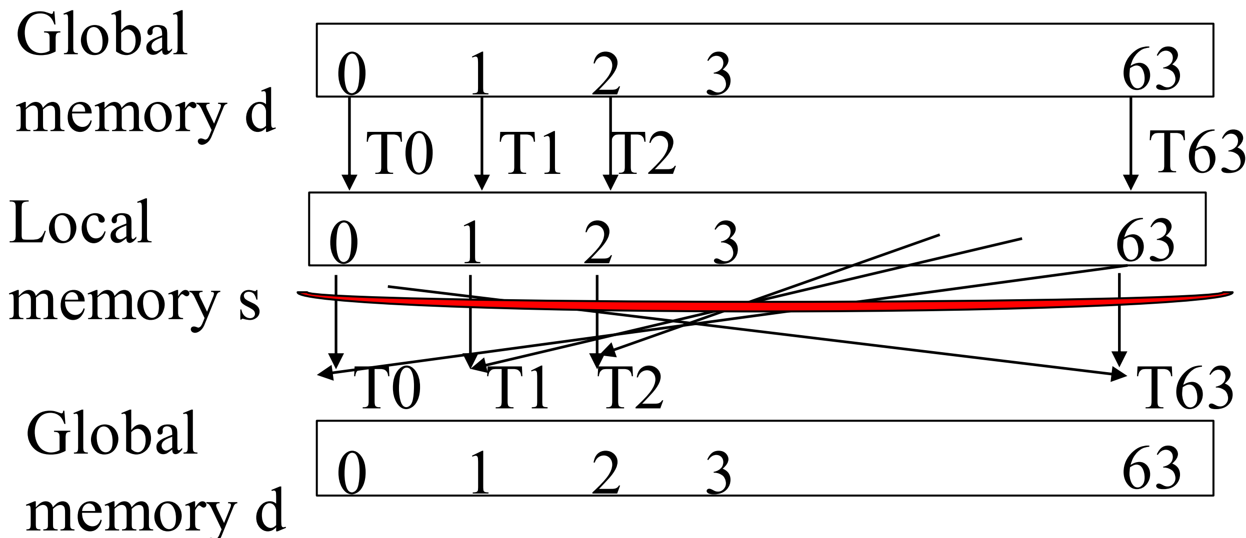
Loading from memory is done in parallel by threads in a block

```
__global__ void AddReverse(int *d, int n) {
    __shared__ int s[64];
    int t = threadIdx.x;
    int r = n-t-1;
    s[t] = d[t];
    __syncthreads(); //Block barrier
    d[t] = s[r]+s[t];
}
```

```
For i=0; i<64; i++)
    s[i]=d[i];
```

Fast shared mem

```
For i=0; i<64; i++)
    d[i]=s[i]+s[63-i];
```



UCSB

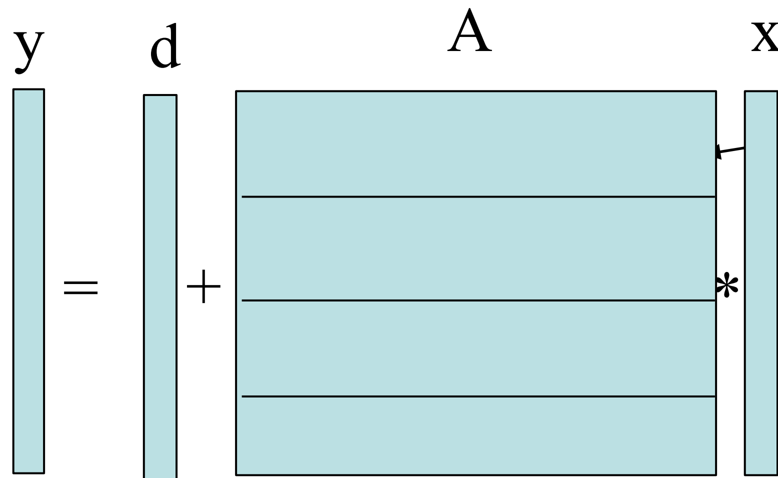
Summary of Lecture

- **Introduction to CUDA: C API supporting heterogeneous data-parallel CPU+GPU execution**
  - Computation partitioning
  - Data partitioning (parts of this implied by decomposition into threads)
  - Data organization and management
  - Concurrency management
- **Compiler nvcc takes as input a .cu program and produces**
  - C Code for host processor (CPU), compiled by native C compiler
  - Code for device processor (GPU), compiled by nvcc compiler

Takeaways and Summary

- **Thread scheduling is block by block**, and warp by warp within each block.
  - Multiple blocks can run in parallel with multiple streaming multiprocessors.
- **Threads are organized as blocks of threads**, and blocks are grouped as grid.
  - Typically with 2D naming. There is a limit on number of threads per block.
  - Power GPU can often execute 10K-30K of threads in parallel. Need to create a lot of fine-grain threads to exploit GPU hardware parallelism.
- **Memory layout for threads** include: global memory for all threads, shared memory for a block, and thread specific memory.

PA3 Q1: Kernel Jacobi code for y=d+Ax

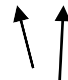


Each thread owns a number of rows: rows_per_thread

4 blocks of 128 threads
Total 512 threads

Memory operations involved for computing y_i

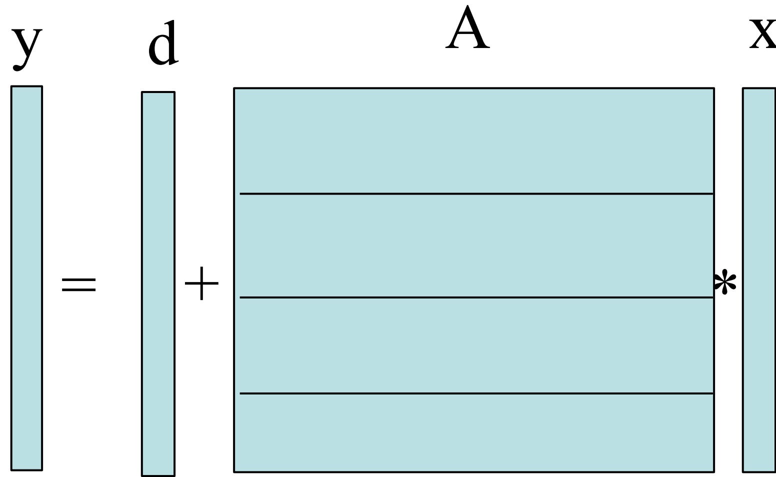
$$y_i=d_i+A_ix$$



Global memory pre-allocated

PA3: Parallel Jacobi Method: y=d+Ax on GPU

Memory operations involved for computing y_i



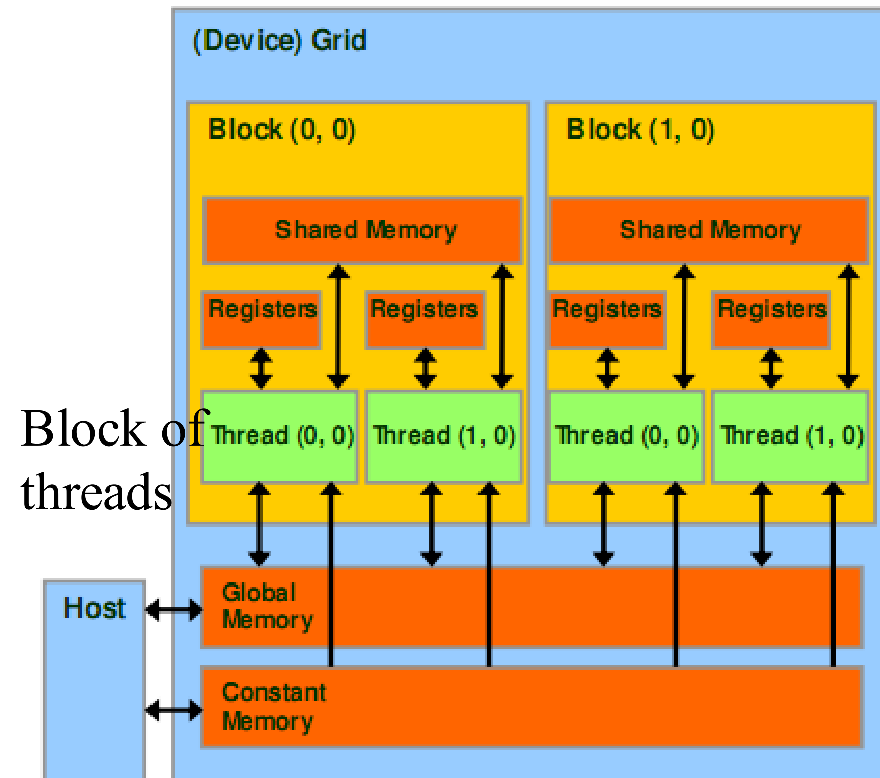
$y_i = d_i + A_i x$

Where are matrix A and vectors d, x and y located in GPU?

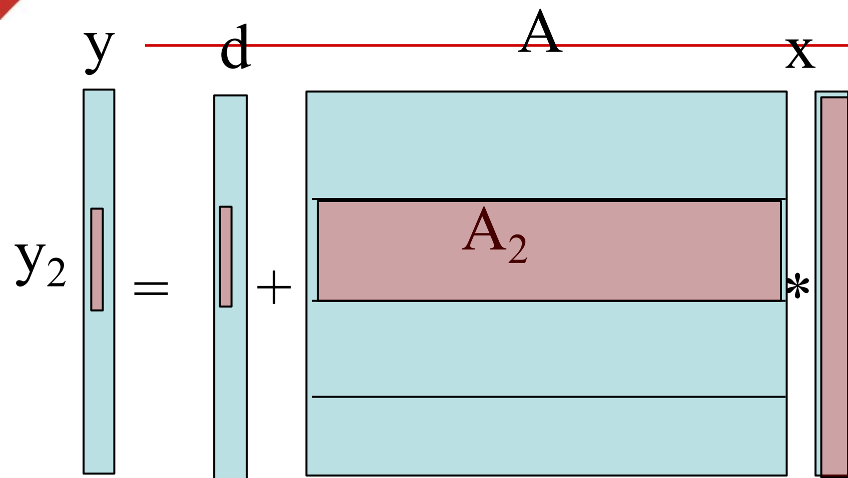
Initially all in global memory

For each iteration, all threads access global memory to fetch A, d and x. Use owner compute rule by distributing vector y to threads.

Block of threads



PA3 Q1: Parallel Jacobi

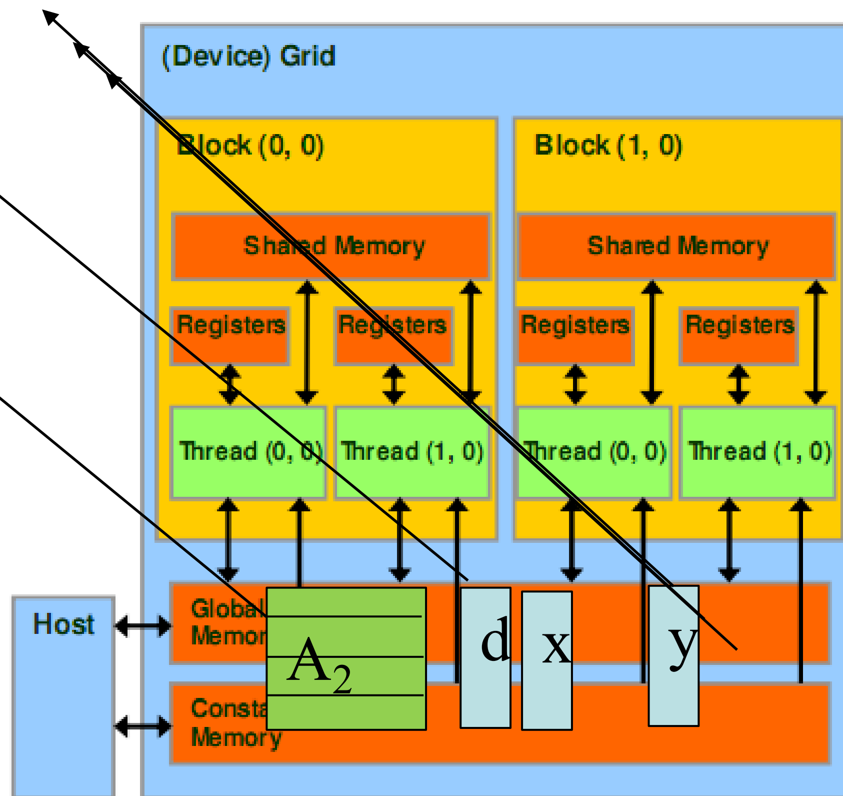


Memory operations involved for computing y_i

$$y_i=d_i+A_i x$$

For each iteration, all threads access global memory to fetch A, d, and x. Use owner compute rule by distributing vector y to threads.

Slow for every x element to accessed from global memory



PA3: Kernel Jacobi code for y=d+Ax

4 blocks of 16 threads: 64 threads

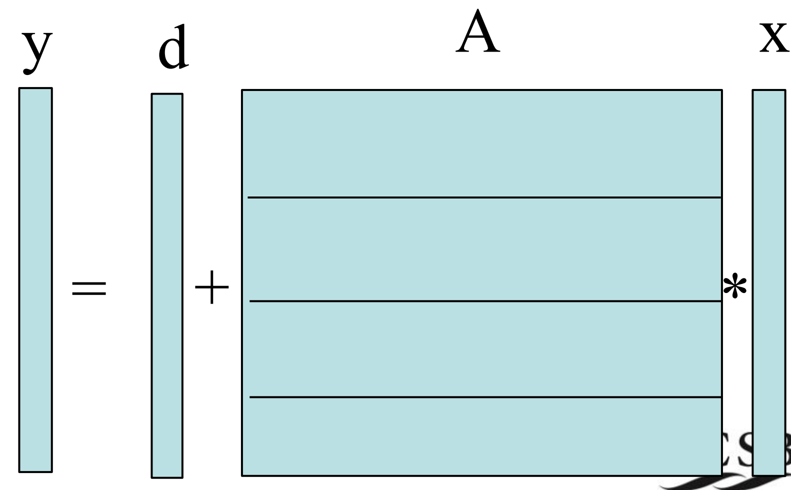
```
__global__ void mult_vec(int n, int rows_per_thread, float *y, float *d, float
*A,   float *x, float *diff) {

  int idx=??; /*Assign a linearized thread ID, to determine what I own*/

  for (int i = 0; i < rows_per_thread; i++) {
    int row_index = idx * rows_per_thread + i;
    double sum = d[row_index];
    for (int j = 0; j < n; j++) {
          sum += A[row_index*n + j]*x[j];
    }
    y[row_index] = sum;

    diff[row_index] = fabs(sum - x[row_index]);
  }
}
```

Each thread owns a number of rows: rows_per_thread

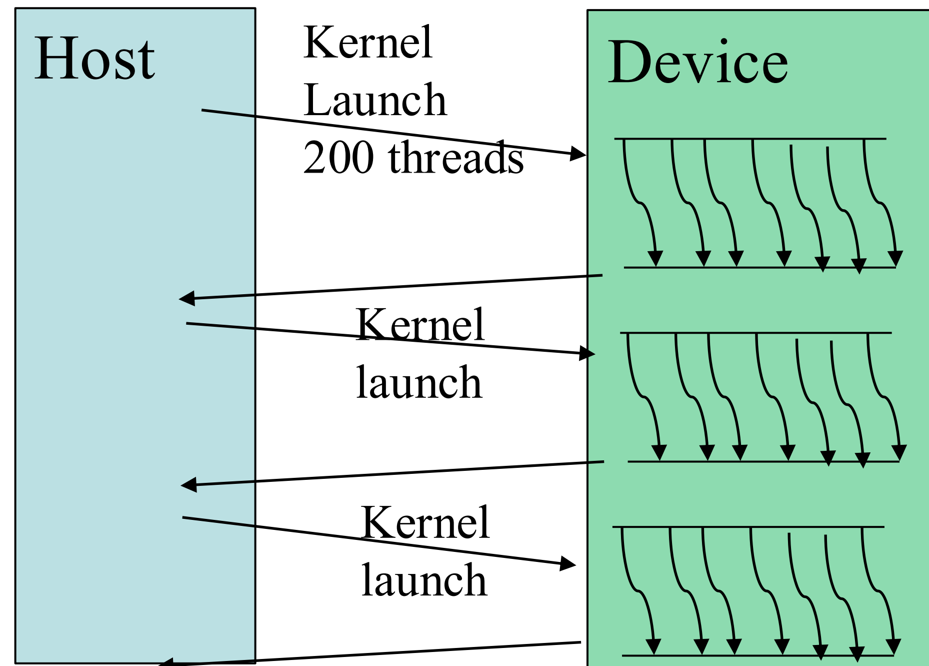


PA3 Q2: Parallel Gauss Seidel

Use **consecutive kernel calls** from CPU host to fork threads and implicitly join these threads.

Host runs a Gauss-Seidel kernel on 2 blocks of threads and each block has 100 threads:

```
for(i=0;i< 100;i++){
   kernelGauss-Seidel<<<2, 100>>>();
   Check error.
}
```



UCSB

PA3: Parallel Jacobi/Gauss Seidel method

$\Longrightarrow$ Jacobi method.

$$x_1^{k+1} = \frac{1}{6}(11 - (-2x_2^k + x_3^k))$$
$$x_2^{k+1} = \frac{1}{7}(5 - (-2x_1^k + 2x_3^k))$$
$$x_3^{k+1} = \frac{1}{-5}(-1 - (x_1^k + 2x_2^k))$$

$\Longrightarrow$ Gauss-Seidel method.

$$x_1^{k+1} = \frac{1}{6}(11 - (-2x_2^k + x_3^k))$$
$$x_2^{k+1} = \frac{1}{7}(5 - (-2x_1^{k+1} + 2x_3^k))$$
$$x_3^{k+1} = \frac{1}{-5}(-1 - (x_1^{k+1} + 2x_2^{k+1}))$$

Matrix notation:

$x^k = d + A\, x^k$

- Jacobi method has sufficient parallelism
- Gauss Seidel uses updated solutions ASAP.
  - More dependence
  - Less parallelism

UCSB

Gauss-Seidel method with vectors x and y

y=x;

While *(error >threshold)*

For *i= 1* ***to*** *n*

T_i: $y_i=d_i+A_iy$;

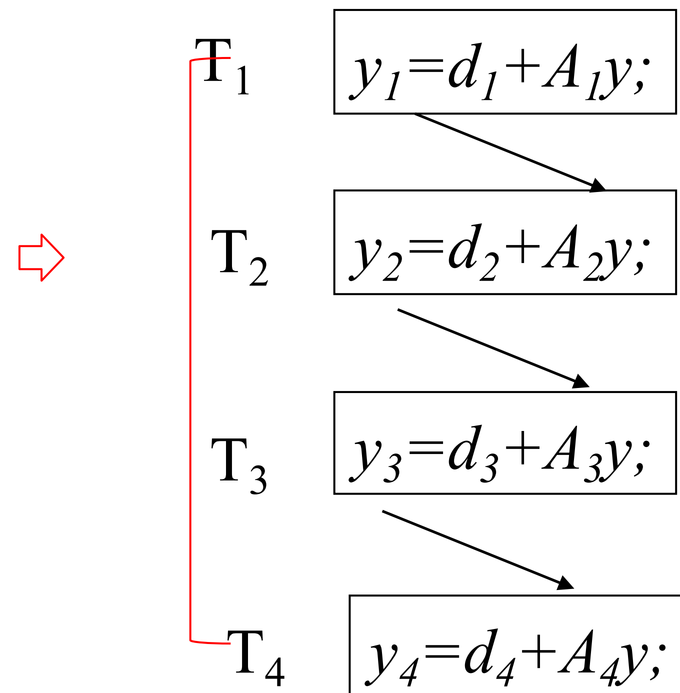
EndFor

error = ||y-x||;

x=y;

EndWhile

Task graph for i=1, 2, 3, 4



Asynchronous Parallel Gauss-Seidel

Asynchronous Execution

$y=x;$

While *(error >threshold)*

  ParallelFor *i= 1* ***to*** *n*

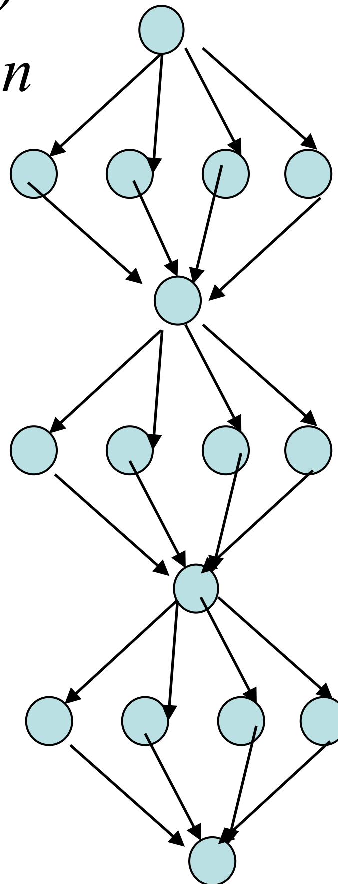
    $y_i=d_i+A_i y;$

  EndFor

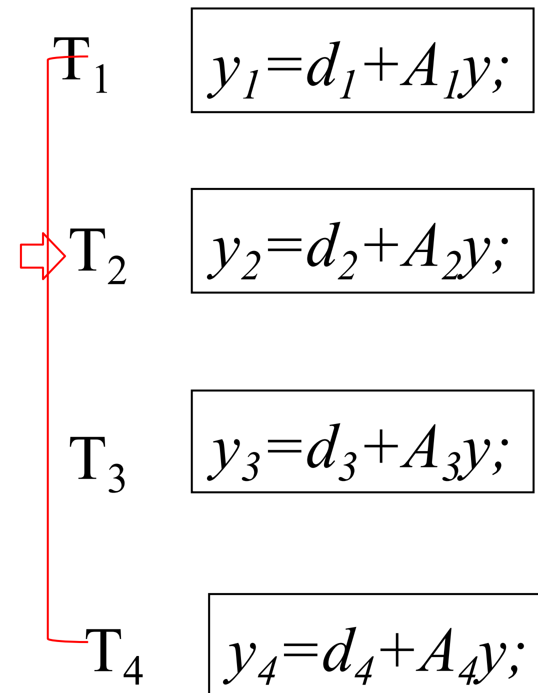
  error = ||y-x||;

  $x=y;$

EndWhile



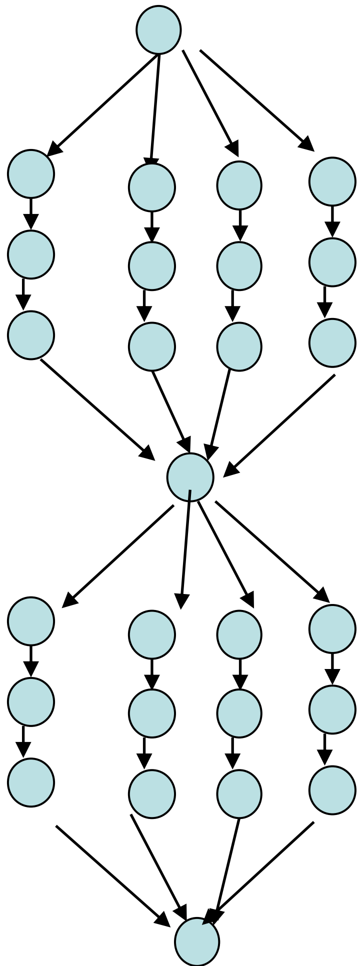
Use whatever available to Compute without synchronization



Still lots of barrier calls in error check

UCSB

Batched Asynchronous Gauss-Seidel



GPU device code
No inter-thread synchronization within a batch

Compute partial error in parallel

Asynchronous Batches

$y=x;$
While *(error >threshold)*
  ParallelFor $i= 1$ ***to*** n
    Repeat *r* *iterations*
      $y_i=d_i+A_iy;$
    EndRepeat
    $e_i=y_i-x_i$
  EndFor
  $error = ||e||;$
  $x=y;$ ← Host control
EndWhile

UCSB

PA3: Kernel code for y=d+Ax with asynchronous Gauss-Seidel

```
__global__ void mult_vec_async(int n, int rows_per_thread, int num_async_iter, float *y, float
*d, float *A, float *x, float *diff) {

  int idx=??; /*Assign a linearized thread ID*/
  for (int i = 0; i < rows_per_thread; i++) {
    int row_index = idx * rows_per_thread + i;
    y[row_index] = x[row_index]; //Start with current value of x
  }
  for (int k = 0; k < num_async_iter; k++) {
    /*Perform asynchronous Gauss-Seidel method for y=d+Ay*/

  }

  for (int i = 0; i < rows_per_thread; i++) {//Error difference
    int row_index = idx * rows_per_thread + i;
    diff[row_index] = fabs(x[row_index] - y[row_index]);
  }
}
```