GPU Computing with CUDA
Lecture 3 - Efficient Shared Memory Use

Christopher Cooper
Boston University

August, 2011
UTFSM, Valparaíso, Chile

Outline of lecture

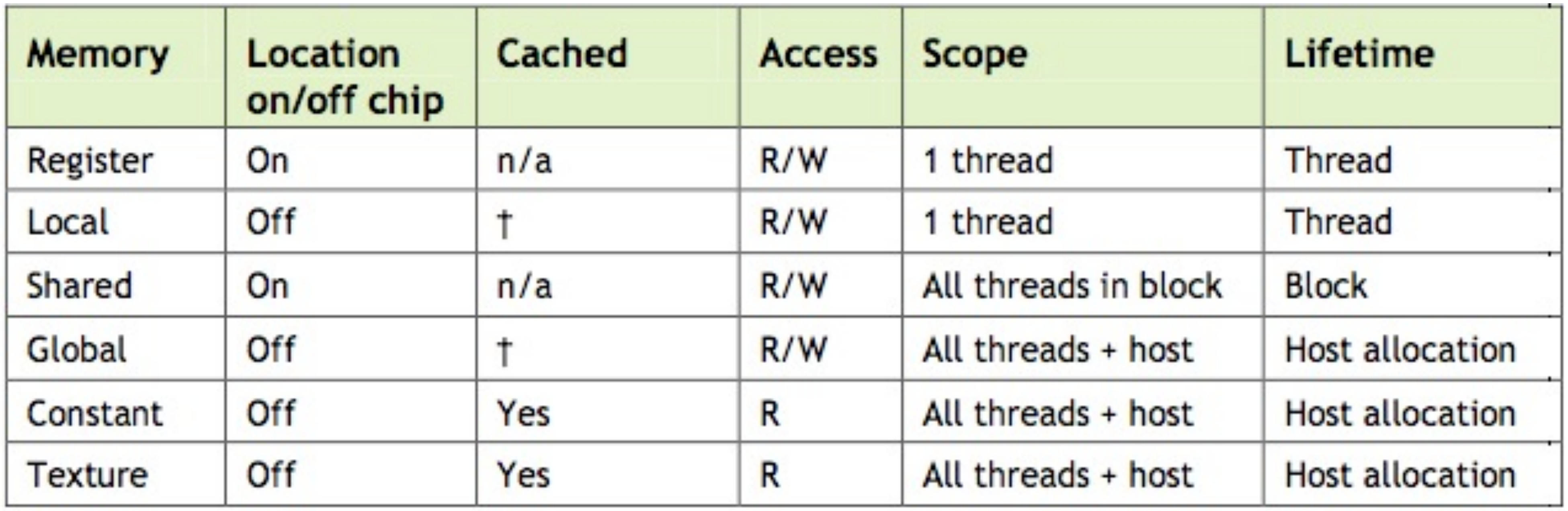
- Recap of Lecture 2
- Shared memory in detail
- Tiling
- Bank conflicts
- Thread synchronization and atomic operations

Recap

- Thread hierarchy
  - Thread are grouped in **thread blocks**
  - Threads of the same block are executed on the same SM at the same time
    - Threads can **communicate** with shared memory
    - An SM can have up to 8 blocks at the same time
  - Thread blocks are divided sequentially into **warps** of 32 threads each
  - Threads of the same warp are scheduled together
  - SM implements a zero-overhead warp scheduling

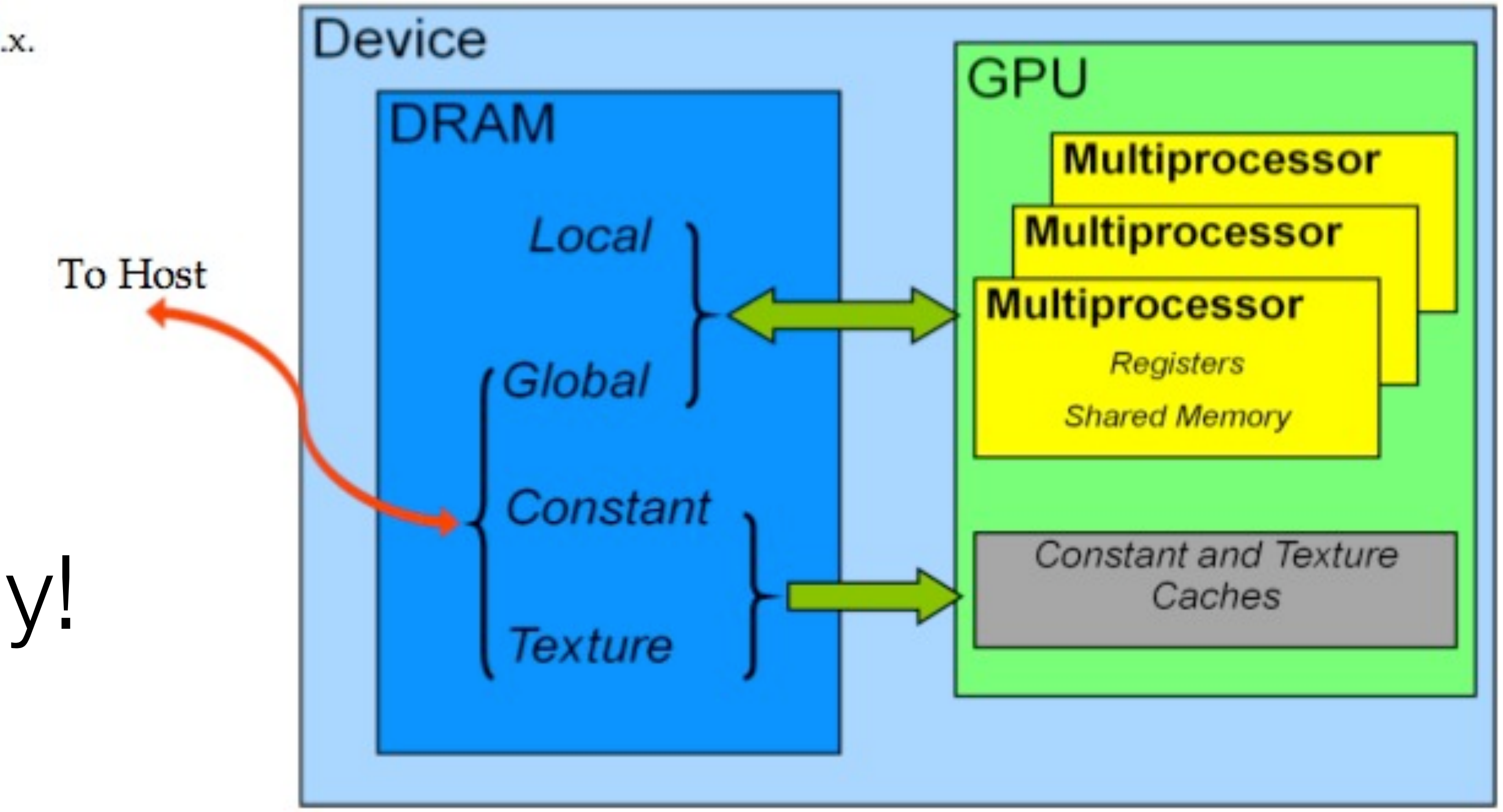
Recap

- Memory hierarchy



Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	†	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

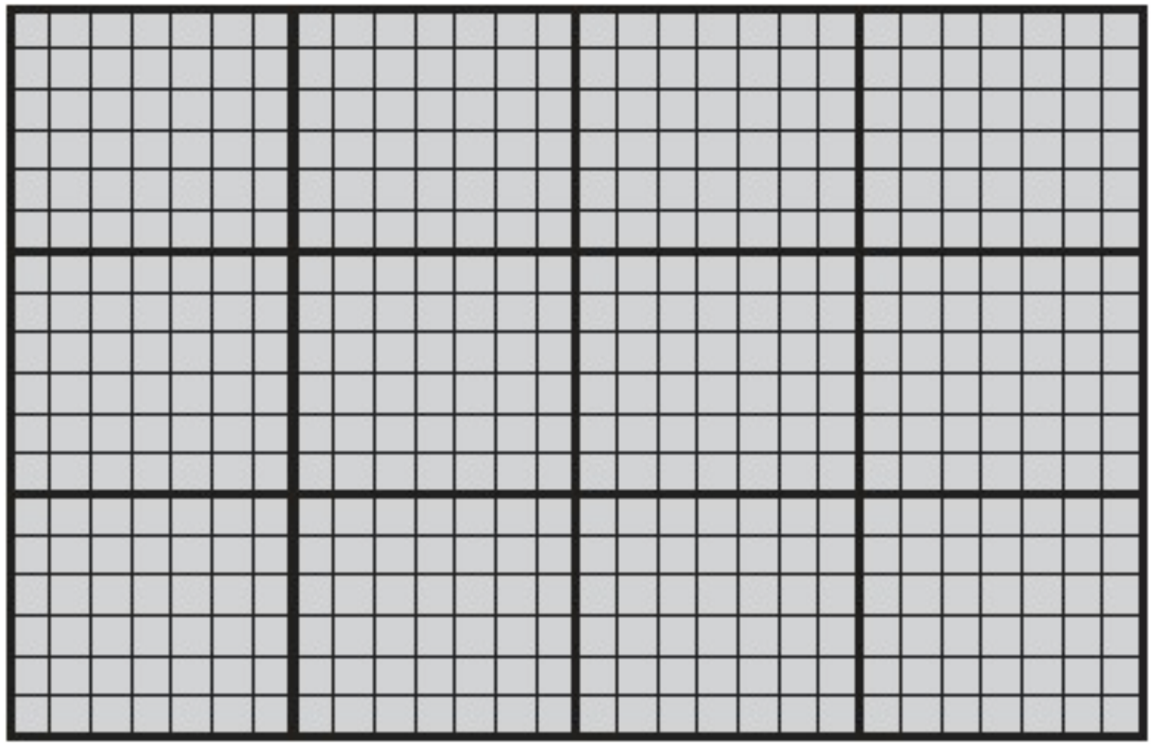
†Cached only on devices of compute capability 2.x.

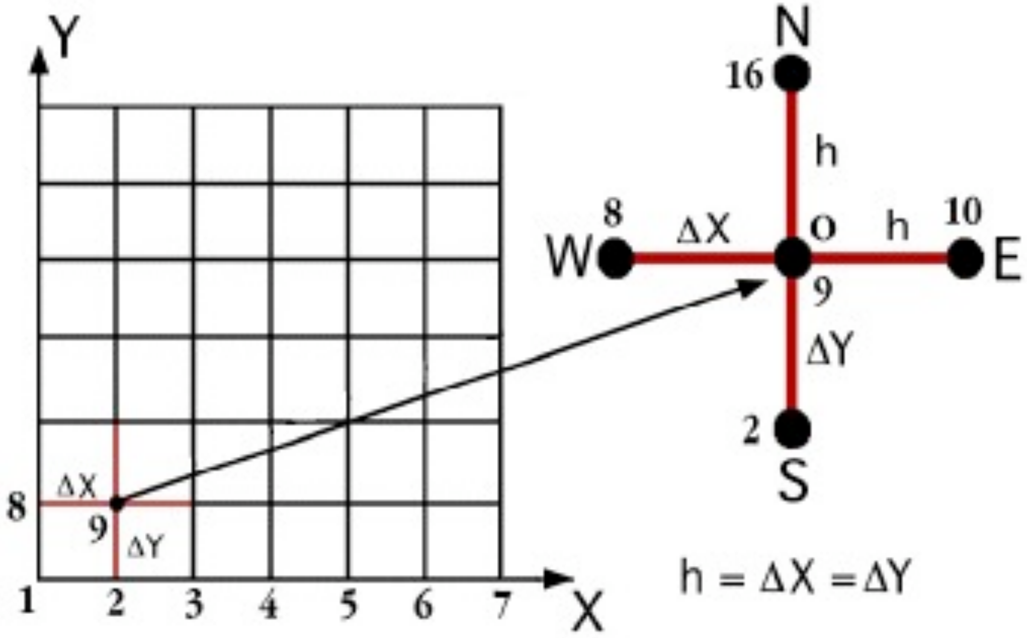


Smart use of memory hierarchy!

Recap

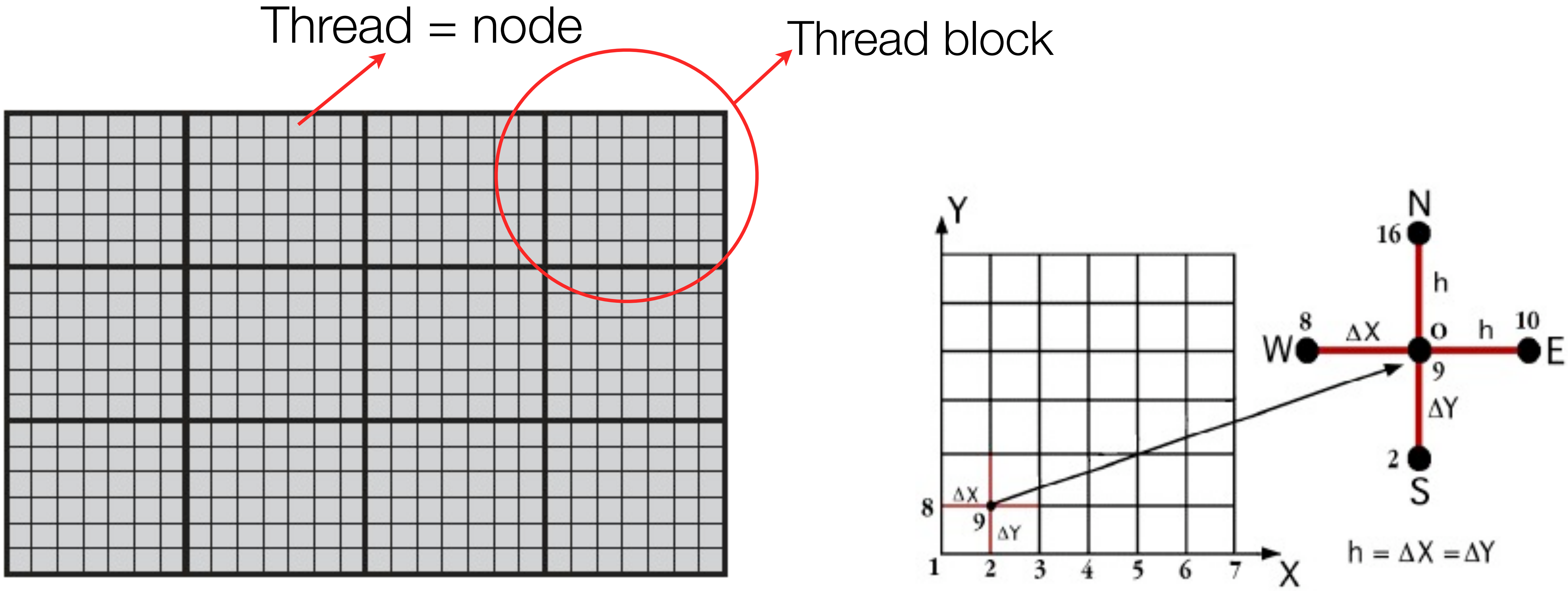
- Programming model: Finite Difference case
  - One node per thread
  - Node indexing automatically groups into thread blocks!





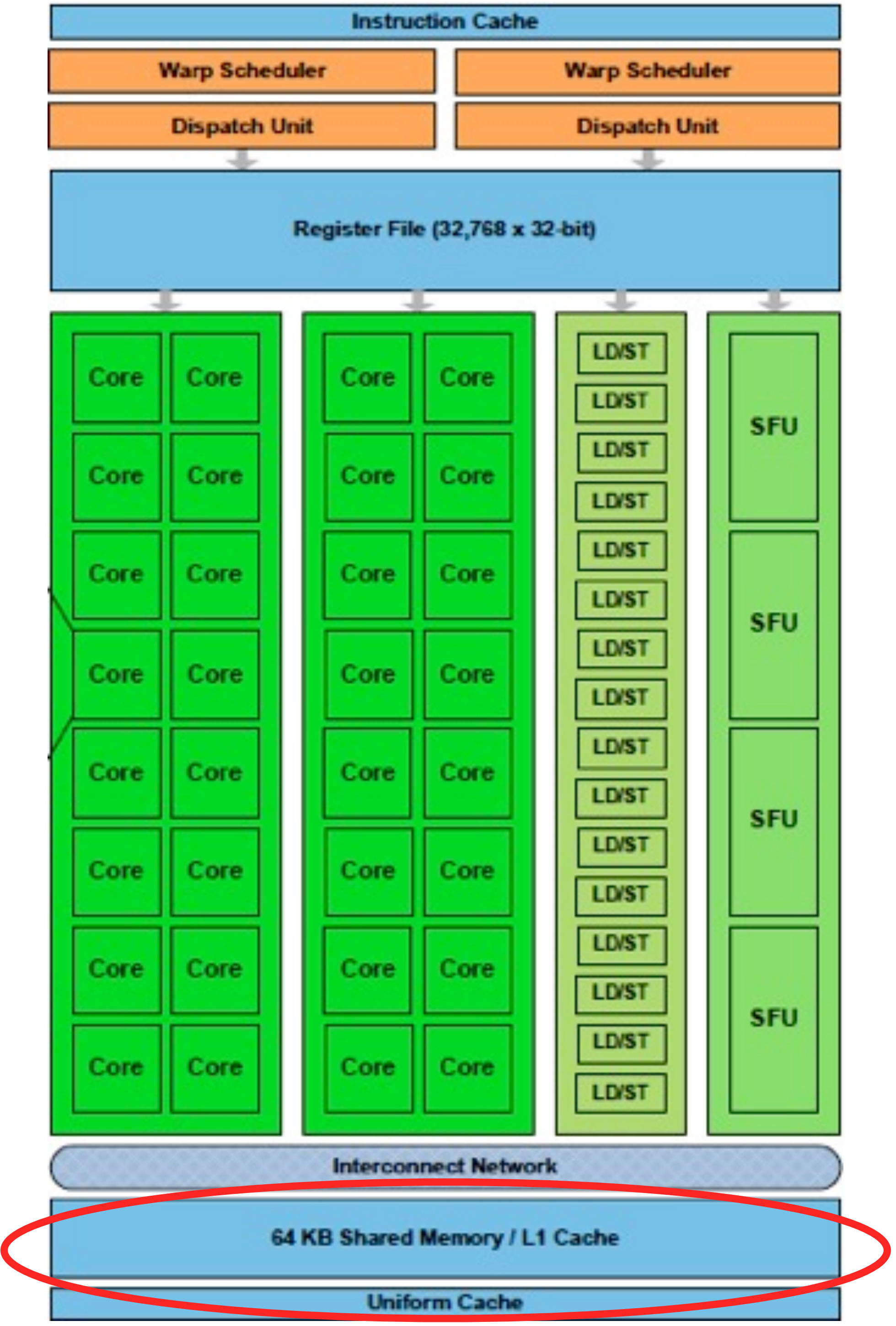
Recap

- Programming model: Finite Difference case
  - One node per thread
  - Node indexing automatically groups into thread blocks!



Shared Memory

- Small (48kB per SM)
- Fast (~4 cycles): On-chip
- Private to each block
  - Allows thread communication
- How can we use it?



Shared Memory - Making use of it

- Looking at a 1D FDM example (similar to lab)

$$\frac{\partial u}{\partial t} = c\frac{\partial u}{\partial x} \longrightarrow u_i^{n+1} = u_i^n - \frac{c\Delta t}{\Delta x}(u_i^n - u_{i-1}^n)$$

```
__global__ void update (float *u, float *u_prev, int N, float dx, float dt, float c, int 
BLOCKSIZE)
{
        // Each thread will load one element
        int i = threadIdx.x + BLOCKSIZE * blockIdx.x;

        if (i>=N){return;}

        // u_prev[i] = u[i] is done in separate kernel

        if (i>0)
        {       u[i] = u_prev[i] - c*dt/dx*(u_prev[i] - u_prev[i-1]);
        }
}
```

Shared Memory - Making use of it

- Looking at a 1D FDM example (similar to lab)

$$\frac{\partial u}{\partial t} = c\frac{\partial u}{\partial x} \longrightarrow u_i^{n+1} = u_i^n - \frac{c\Delta t}{\Delta x}(u_i^n - u_{i-1}^n)$$

```
__global__ void update (float *u, float *u_prev, int N, float dx, float dt, float c, int
BLOCKSIZE)
{
        // Each thread will load one element
        int i = threadIdx.x + BLOCKSIZE * blockIdx.x;

        if (i>=N){return;}

        // u_prev[i] = u[i] is done in separate kernel

        if (i>0)
        {       u[i] = u_prev[i] - c*dt/dx*(u_prev[i] - u_prev[i-1]);
        }
}
```

Thread i

Shared Memory - Making use of it

- Looking at a 1D FDM example (similar to lab)

$$\frac{\partial u}{\partial t} = c\frac{\partial u}{\partial x} \longrightarrow u_i^{n+1} = u_i^n - \frac{c\Delta t}{\Delta x}(u_i^n - u_{i-1}^n)$$

```
__global__ void update (float *u, float *u_prev, int N, float dx, float dt, float c, int
BLOCKSIZE)
{
        // Each thread will load one element
        int i = threadIdx.x + BLOCKSIZE * blockIdx.x;

        if (i>=N){return;}

        // u_prev[i] = u[i] is done in separate kernel

        if (i>0)
        {       u[i] = u_prev[i] - c*dt/dx*(u_prev[i] - u_prev[i-1]);
        }
}
```

Thread i

Loads element i

Shared Memory - Making use of it

- Looking at a 1D FDM example (similar to lab)

$$\frac{\partial u}{\partial t} = c\frac{\partial u}{\partial x} \longrightarrow u_i^{n+1} = u_i^n - \frac{c\Delta t}{\Delta x}(u_i^n - u_{i-1}^n)$$

```
__global__ void update (float *u, float *u_prev, int N, float dx, float dt, float c, int
BLOCKSIZE)
{
        // Each thread will load one element
        int i = threadIdx.x + BLOCKSIZE * blockIdx.x;

        if (i>=N){return;}

        // u_prev[i] = u[i] is done in separate kernel

        if (i>0)
        {       u[i] = u_prev[i] - c*dt/dx*(u_prev[i] - u_prev[i-1]);
        }
}
```

Thread i

Loads element i

Loads element i-1

Shared Memory - Making use of it

- Looking at a 1D FDM example (similar to lab)

$$\frac{\partial u}{\partial t} = c\frac{\partial u}{\partial x} \longrightarrow u_i^{n+1} = u_i^n - \frac{c\Delta t}{\Delta x}(u_i^n - u_{i-1}^n)$$

```
__global__ void update (float *u, float *u_prev, int N, float dx, float dt, float c, int
BLOCKSIZE)
{
        // Each thread will load one element
        int i = threadIdx.x + BLOCKSIZE * blockIdx.x;

        if (i>=N){return;}

        // u_prev[i] = u[i] is done in separate kernel

        if (i>0)
        {       u[i] = u_prev[i] - c*dt/dx*(u_prev[i] - u_prev[i-1]);
        }
}
```

Shared Memory - Making use of it

- Looking at a 1D FDM example (similar to lab)

$$\frac{\partial u}{\partial t} = c\frac{\partial u}{\partial x} \longrightarrow u_i^{n+1} = u_i^n - \frac{c\Delta t}{\Delta x}(u_i^n - u_{i-1}^n)$$

```
__global__ void update (float *u, float *u_prev, int N, float dx, float dt, float c, int
BLOCKSIZE)
{
        // Each thread will load one element
        int i = threadIdx.x + BLOCKSIZE * blockIdx.x;

        if (i>=N){return;}

        // u_prev[i] = u[i] is done in separate kernel

        if (i>0)
        {       u[i] = u_prev[i] - c*dt/dx*(u_prev[i] - u_prev[i-1]);
        }
}
```

Thread i +1

Shared Memory - Making use of it

- Looking at a 1D FDM example (similar to lab)

$$\frac{\partial u}{\partial t} = c\frac{\partial u}{\partial x} \longrightarrow u_i^{n+1} = u_i^n - \frac{c\Delta t}{\Delta x}(u_i^n - u_{i-1}^n)$$

```
__global__ void update (float *u, float *u_prev, int N, float dx, float dt, float c, int
BLOCKSIZE)
{
        // Each thread will load one element
        int i = threadIdx.x + BLOCKSIZE * blockIdx.x;

        if (i>=N){return;}

        // u_prev[i] = u[i] is done in separate kernel

        if (i>0)
        {       u[i] = u_prev[i] - c*dt/dx*(u_prev[i] - u_prev[i-1]);
        }
}
```

Thread i +1

Loads element i+1

Shared Memory - Making use of it

- Looking at a 1D FDM example (similar to lab)

$$\frac{\partial u}{\partial t} = c\frac{\partial u}{\partial x} \longrightarrow u_i^{n+1} = u_i^n - \frac{c\Delta t}{\Delta x}(u_i^n - u_{i-1}^n)$$

```
__global__ void update (float *u, float *u_prev, int N, float dx, float dt, float c, int
BLOCKSIZE)
{
        // Each thread will load one element
        int i = threadIdx.x + BLOCKSIZE * blockIdx.x;

        if (i>=N){return;}

        // u_prev[i] = u[i] is done in separate kernel

        if (i>0)
        {       u[i] = u_prev[i] - c*dt/dx*(u_prev[i] - u_prev[i-1]);
        }
}
```

Thread i +1

Loads element i+1

Loads element i

Shared Memory - Making use of it

- Looking at a 1D FDM example (similar to lab)

$$\frac{\partial u}{\partial t} = c\frac{\partial u}{\partial x} \longrightarrow u_i^{n+1} = u_i^n - \frac{c\Delta t}{\Delta x}(u_i^n - u_{i-1}^n)$$

```
__global__ void update (float *u, float *u_prev, int N, float dx, float dt, float c, int
BLOCKSIZE)
{
        // Each thread will load one element
        int i = threadIdx.x + BLOCKSIZE * blockIdx.x;

        if (i>=N){return;}

        // u_prev[i] = u[i] is done in separate kernel

        if (i>0)
        {       u[i] = u_prev[i] - c*dt/dx*(u_prev[i] - u_prev[i-1]);
        }
}
```

Thread i +1

Loads element i+1

Loads element i

Order N redundant loads!

Shared Memory - Making use of it

- Idea: We could load only once to shared memory, and operate there

```
__global__ void update (float *u, float *u_prev, int N, float dx, float dt, float c)
{
        // Each thread will load one element
        int i = threadIdx.x;
        int I = threadIdx.x + BLOCKSIZE * blockIdx.x;
        __shared__ float u_shared[BLOCKSIZE];

        if (I>=N){return;}

        u_shared[i] = u[I];
         __syncthreads();
        if (I>0)
        {       u[I] = u_shared[i] - c*dt/dx*(u_shared[i] - u_shared[i-1]);        }
}
```

Shared Memory - Making use of it

- Idea: We could load only once to shared memory, and operate there

```
__global__ void update (float *u, float *u_prev, int N, float dx, float dt, float c)
{
        // Each thread will load one element
        int i = threadIdx.x;
        int I = threadIdx.x + BLOCKSIZE * blockIdx.x;
        __shared__ float u_shared[BLOCKSIZE];

        if (I>=N){return;}

        u_shared[i] = u[I];
         __syncthreads();
        if (I>0)
        {       u[I] = u_shared[i] - c*dt/dx*(u_shared[i] - u_shared[i-1]);        }
}
```

Allocate shared array

Shared Memory - Making use of it

- Idea: We could load only once to shared memory, and operate there

```
__global__ void update (float *u, float *u_prev, int N, float dx, float dt, float c)
{
        // Each thread will load one element
        int i = threadIdx.x;
        int I = threadIdx.x + BLOCKSIZE * blockIdx.x;
        __shared__ float u_shared[BLOCKSIZE];

        if (I>=N){return;}

        u_shared[i] = u[I];
         __syncthreads();
        if (I>0)
        {       u[I] = u_shared[i] - c*dt/dx*(u_shared[i] - u_shared[i-1]);       }
}
```

Allocate shared array

Load to shared mem

Shared Memory - Making use of it

- Idea: We could load only once to shared memory, and operate there

```
__global__ void update (float *u, float *u_prev, int N, float dx, float dt, float c)
{
        // Each thread will load one element
        int i = threadIdx.x;
        int I = threadIdx.x + BLOCKSIZE * blockIdx.x;
        __shared__ float u_shared[BLOCKSIZE];

        if (I>=N){return;}

        u_shared[i] = u[I];
        __syncthreads();
        if (I>0)
        {       u[I] = u_shared[i] - c*dt/dx*(u_shared[i] - u_shared[i-1]);        }
}
```

Allocate shared array

Load to shared mem

Fetch shared mem

Shared Memory - Making use of it

- Idea: We could load only once to shared memory, and operate there

```
__global__ void update (float *u, float *u_prev, int N, float dx, float dt, float c)
{
        // Each thread will load one element
        int i = threadIdx.x;
        int I = threadIdx.x + BLOCKSIZE * blockIdx.x;
        __shared__ float u_shared[BLOCKSIZE];

        if (I>=N){return;}

        u_shared[i] = u[I];
        __syncthreads();
        if (I>0)
        {       u[I] = u_shared[i] - c*dt/dx*(u_shared[i] - u_shared[i-1]);        }
}
```

Allocate shared array

Load to shared mem

Fetch shared mem

Works if N <= Block size... What if not?

Shared Memory - Making use of it

```
__global__ void update (float *u, float *u_prev, int N, float dx, float dt, float c)
{
        // Each thread will load one element
        int i = threadIdx.x;
        int I = threadIdx.x + BLOCKSIZE * blockIdx.x;
        __shared__ float u_shared[BLOCKSIZE];

        if (I>=N){return;}

        u_shared[i] = u[I];
         __syncthreads();

        if (i>0 && i<BLOCKSIZE-1)
        {       u[I] = u_shared[i] - c*dt/dx*(u_shared[i] - u_shared[i-1]);
        }
        else
        {       u[I] = u_prev[I] - c*dt/dx*(u_prev[I] - u_prev[I-1]);
        }

}
```

Shared Memory - Making use of it

```
__global__ void update (float *u, float *u_prev, int N, float dx, float dt, float c)
{
        // Each thread will load one element
        int i = threadIdx.x;
        int I = threadIdx.x + BLOCKSIZE * blockIdx.x;
        __shared__ float u_shared[BLOCKSIZE];

        if (I>=N){return;}

        u_shared[i] = u[I];
         __syncthreads();

        if (i>0 && i<BLOCKSIZE-1)
        {       u[I] = u_shared[i] - c*dt/dx*(u_shared[i] - u_shared[i-1]);
        }
        else
        {       u[I] = u_prev[I] - c*dt/dx*(u_prev[I] - u_prev[I-1]);
        }

}
```

Shared Memory - Making use of it

```
__global__ void update (float *u, float *u_prev, int N, float dx, float dt, float c)
{
        // Each thread will load one element
        int i = threadIdx.x;
        int I = threadIdx.x + BLOCKSIZE * blockIdx.x;
        __shared__ float u_shared[BLOCKSIZE];

        if (I>=N){return;}

        u_shared[i] = u[I];
         __syncthreads();

        if (i>0 && i<BLOCKSIZE-1)
        {       u[I] = u_shared[i] - c*dt/dx*(u_shared[i] - u_shared[i-1]);
        }
        else
        {       u[I] = u_prev[I] - c*dt/dx*(u_prev[I] - u_prev[I-1]);
        }

}
```

Reduced loads from 2*N to N+2*N/BLOCKSIZE

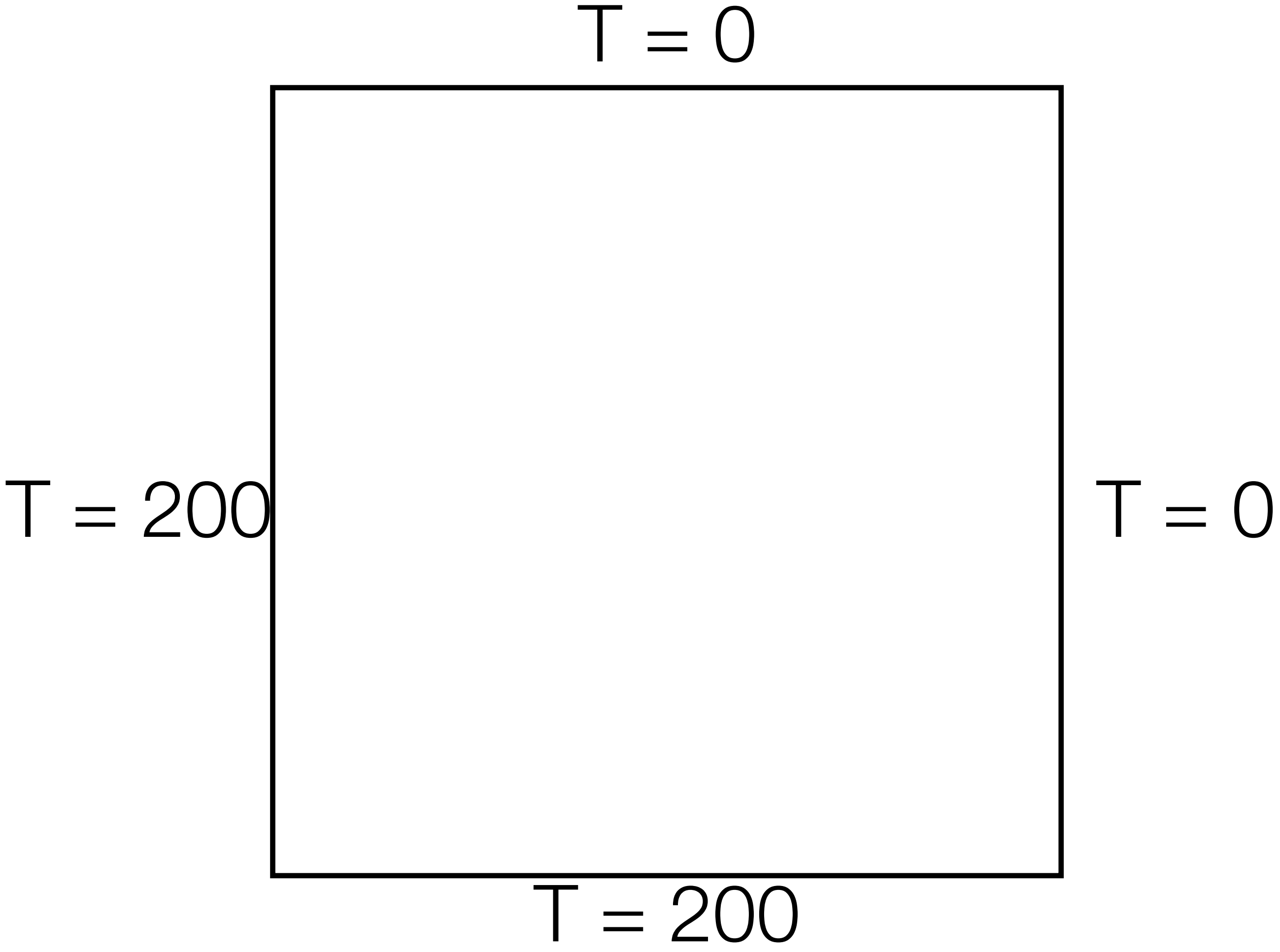
Using shared memory as cache

- Looking at the 2D heat diffusion problem from lab 2

$$\frac{\partial u}{\partial t} = \alpha \nabla^2 u$$

- Explicit scheme

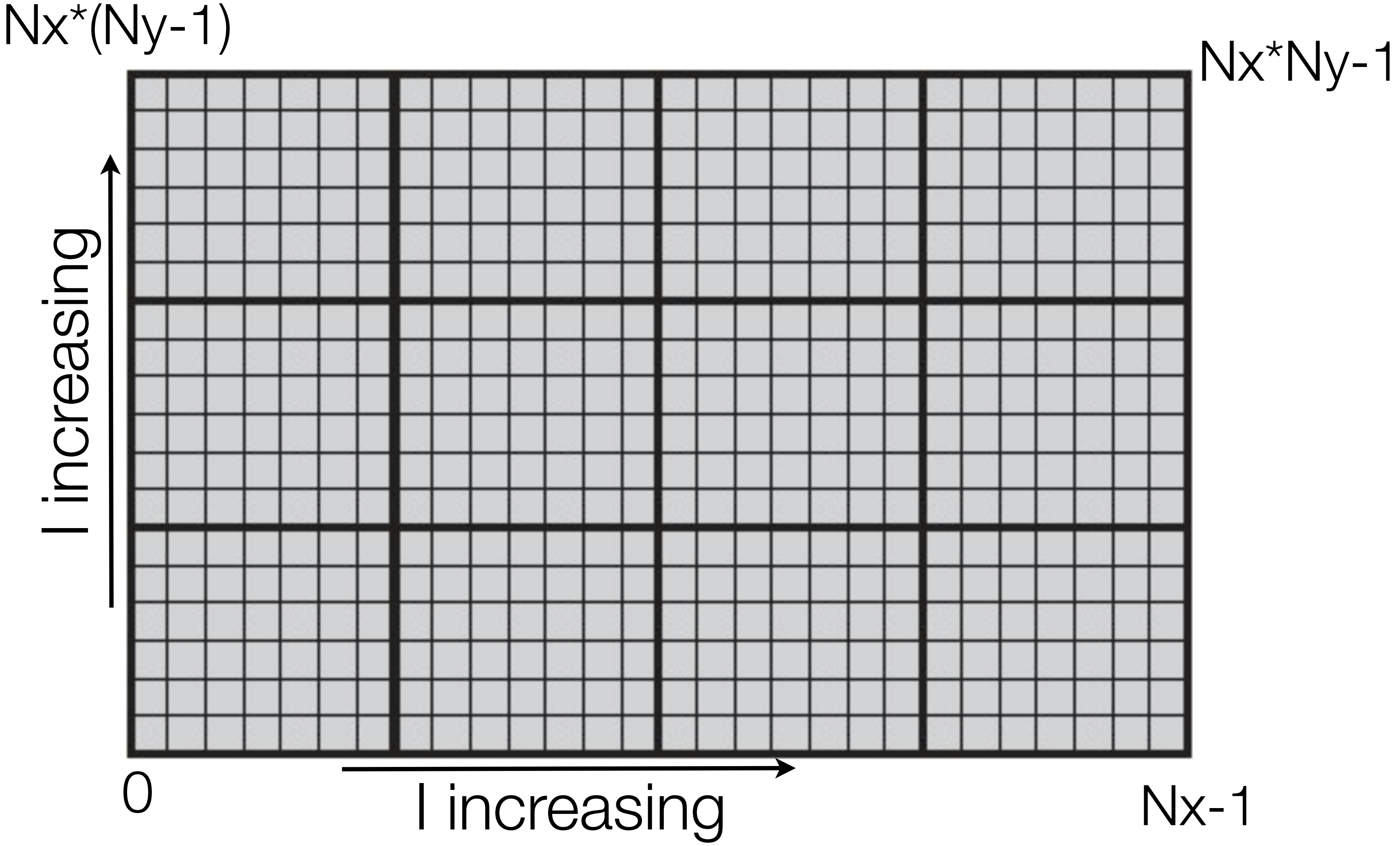
$$u_{i,j}^{n+1} = u_{i,j}^{n} + \frac{\alpha k}{h^2}(u_{i,j+1}^{n} + u_{i,j-1}^{n} + u_{i+1,j}^{n} + u_{i-1,j}^{n} - 4u_{i,j}^{n})$$



Shared Memory Implementation - Mapping Problem

- Using row major flattened array

```
int i = threadIdx.x;
int j = threadIdx.y;
int I = blockIdx.y*BSZ*N + blockIdx.x*BSZ + j*N + i;
```



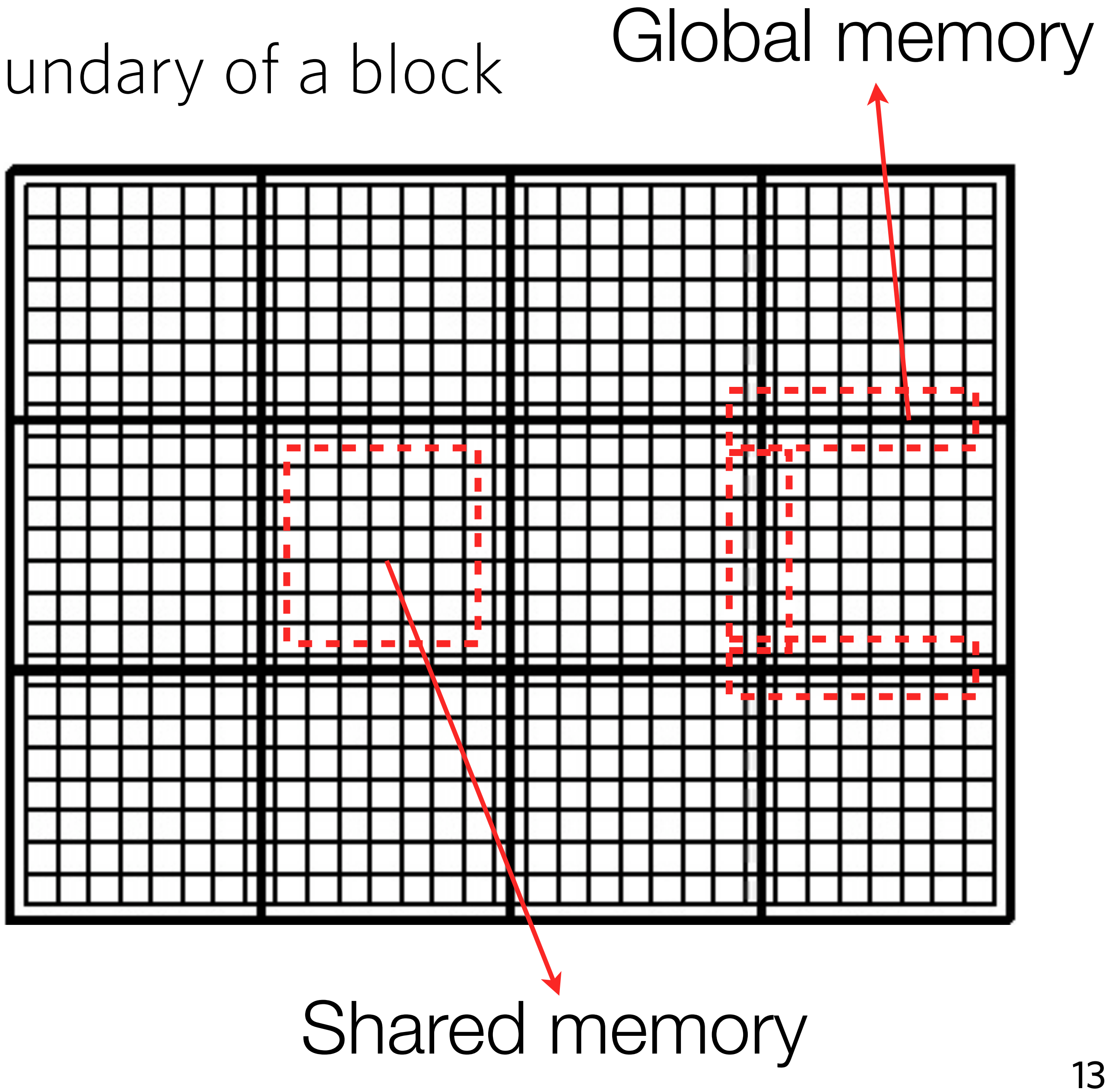
Shared Memory Implementation - Global Memory

- This implementation has redundant loads to global memory → slow

```
__global__ void update (float *u, float *u_prev, int N, float h, float dt, float
alpha, int BSZ)
{
        // Setting up indices
        int i = threadIdx.x;
        int j = threadIdx.y;
        int I = blockIdx.y*BSZ*N + blockIdx.x*BSZ + j*N + i;

        if (I>=N*N){return;}

        // if not boundary do
        if ( (I>N) && (I< N*N-1-N) && (I%N!=0) && (I%N!=N-1))
        {       u[I] = u_prev[I] + alpha*dt/(h*h) * (u_prev[I+1] + u_prev[I-1] +
u_prev[I+N] + u_prev[I-N] - 4*u_prev[I]);
        }
}
```

Shared Memory Implementation - Solution 1

- Recast solution given earlier
  - Load to shared memory
  - Use shared memory if not on boundary of a block
  - Use global memory otherwise
- Advantage
  - Easy to implement
- Disadvantage
  - Branching statement
  - Still have some redundant loads

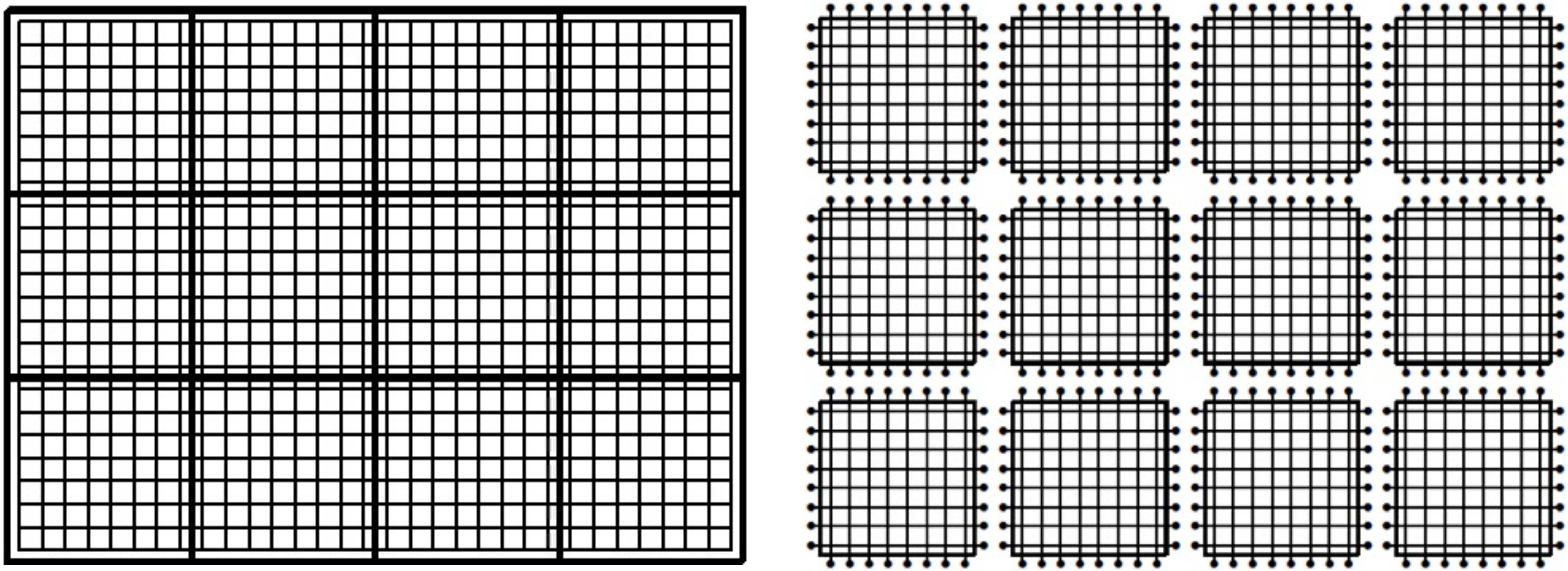


Shared Memory Implementation - Solution 1

```
__global__ void update (float *u, float *u_prev, int N, float h, float dt, float alpha)
{
        // Setting up indices
        int i = threadIdx.x;
        int j = threadIdx.y;
        int I = blockIdx.y*BSZ*N + blockIdx.x*BSZ + j*N + i;
        if (I>=N*N){return;}

        __shared__ float u_prev_sh[BSZ][BSZ];

        u_prev_sh[i][j] = u_prev[I];

        __syncthreads();
        bool bound_check = ((I>N) && (I< N*N-1-N) && (I%N!=0) && (I%N!=N-1));
        bool block_check = ((i>0) && (i<BSZ-1) && (j>0) && (j<BSZ-1));

        // if not on block boundary do
        if (block_check)
        {       u[I] = u_prev_sh[i][j] + alpha*dt/h/h * (u_prev_sh[i+1][j] + u_prev_sh[i-1]
[j] + u_prev_sh[i][j+1] + u_prev_sh[i][j-1] - 4*u_prev_sh[i][j]);
        }
        // if not on boundary
        else if (bound_check)
        {       u[I] = u_prev[I] + alpha*dt/(h*h) * (u_prev[I+1] + u_prev[I-1] + u_prev[I+N]
+ u_prev[I-N] - 4*u_prev[I]);
        }
}
```

Shared Memory Implementation - Solution 2

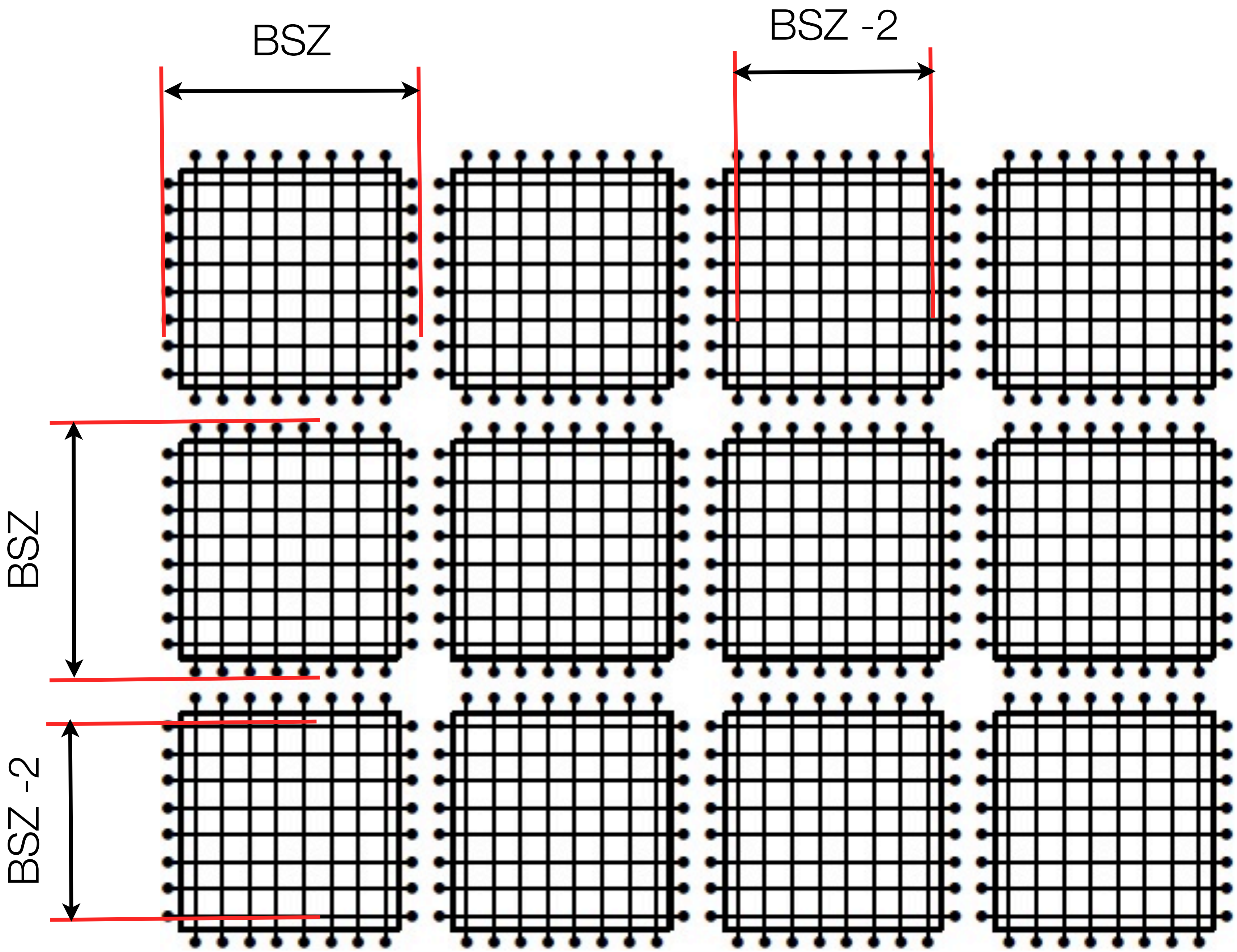
- We want to avoid the reads from global memory
  - Let's use halo nodes to compute block edges



Images: Mark Giles, Oxford, UK

Shared Memory Implementation - Solution 2

- Change indexing so to jump in steps of BSZ-2 instead of BSZ
- Load data to shared memory
- Operate on internal nodes
- We'll need Nx/(BSZ-2) blocks per dimension, instead of Nx/BSZ



Shared Memory Implementation - Solution 2

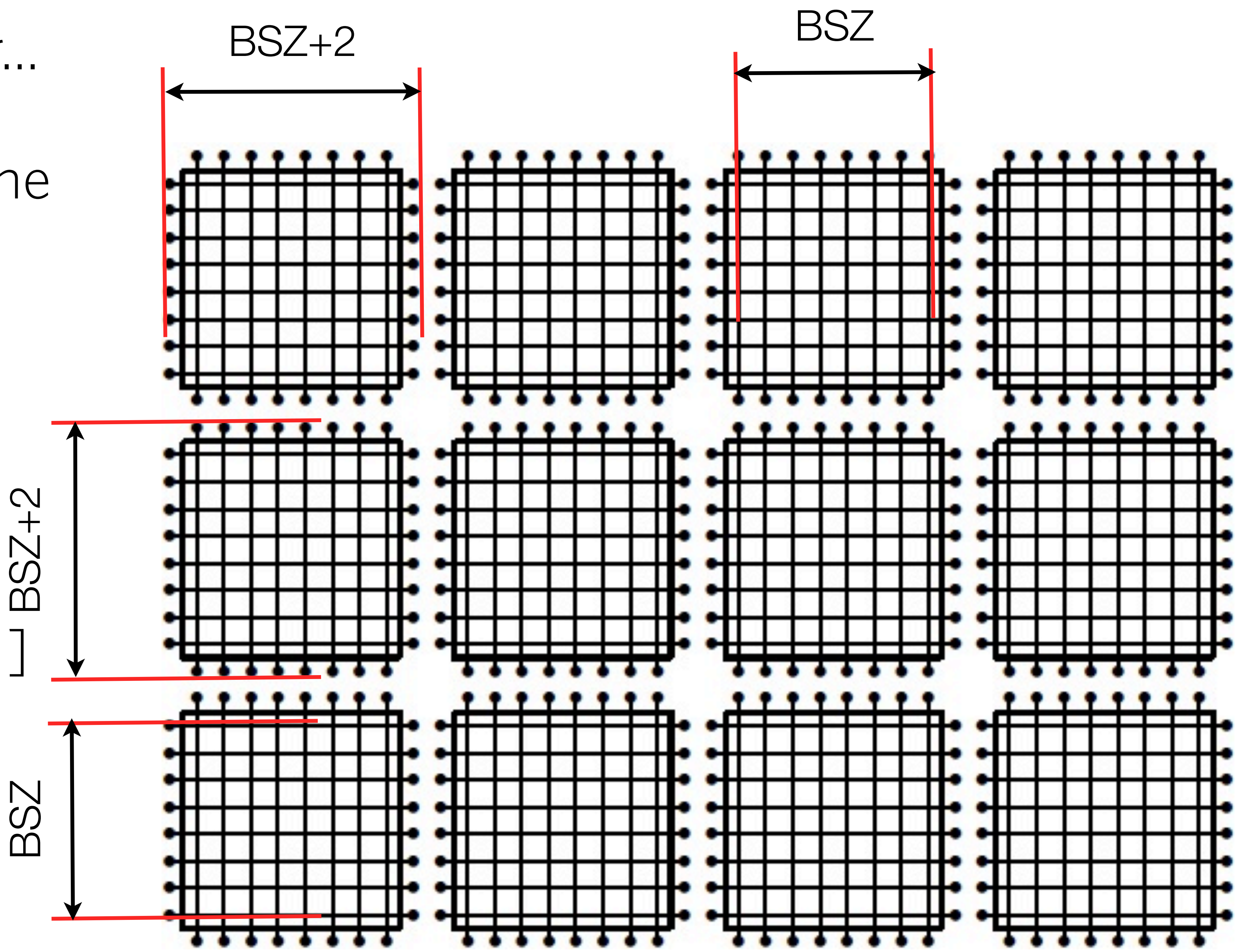
```
__global__ void update (float *u, float *u_prev, int N, float h, float dt, float alpha)
{
        // Setting up indices
        int i = threadIdx.x, j = threadIdx.y, bx = blockIdx.x, by = blockIdx.y;

        int I = (BSZ-2)*bx + i, J = (BSZ-2)*by + j;
        int Index = I + J*N;

        if (I>=N || J>=N){return;}

        __shared__ float u_prev_sh[BSZ][BSZ];

        u_prev_sh[i][j] = u_prev[Index];

        __syncthreads();

        bool bound_check = ((I!=0) && (I<N-1) && (J!=0) && (J<N-1));
        bool block_check = ((i!=0) && (i<BSZ-1) && (j!=0) && (j<BSZ-1));

        if (bound_check && block_check)
        {       u[Index] = u_prev_sh[i][j] + alpha*dt/h/h * (u_prev_sh[i+1][j] +
u_prev_sh[i-1][j] + u_prev_sh[i][j+1] + u_prev_sh[i][j-1] - 4*u_prev_sh[i][j]);
        }
}
```

Shared Memory Implementation - Solution 2

- We've reduced global memory accesses!
- But...
  - There's still a heavy amount of branching
    - GPUs are not great at branching...
  - All threads read, but only some operate
    - We're underutilizing the device!
    - If we have 16x16 = 256 threads, all read, but only 14x14 = 196 operate, and we're using only ~75% of the device. In 3D this number drops to ~40%!

Shared Memory Implementation - Solution 3

- We need to go further…
  - To not underutilize the device, we need to load more data than threads
  - Load in two stages
  - Operate on [i+1][j+1] threads



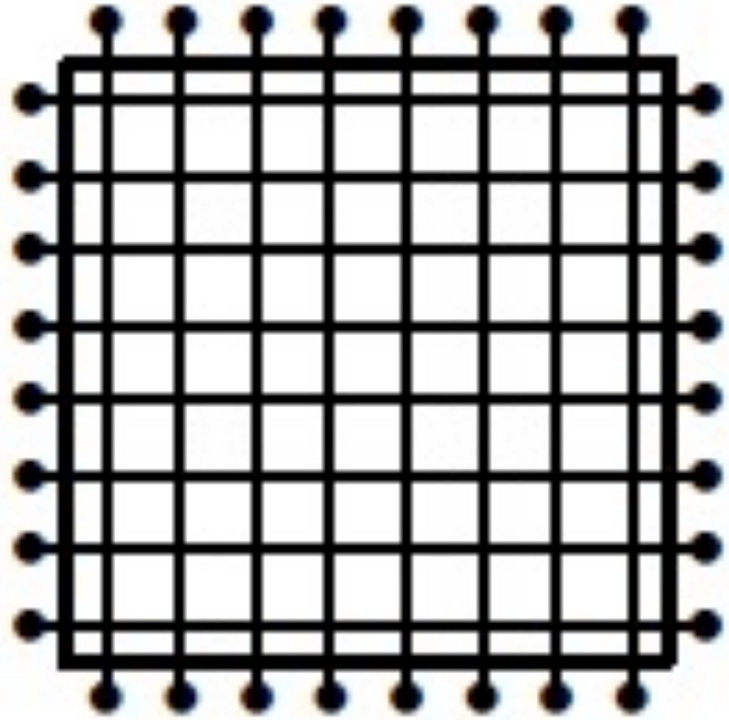
Shared Memory Implementation - Solution 3

- Loading in 2 steps
  - Use the 64 available threads to load the 64 first values to shared

```
__shared__ float u_prev_sh[BSZ+2][BSZ+2];

int ii = j*BSZ + i, // Flatten thread indexing
     I = ii%(BSZ+2), // x-direction index including halo
     J = ii/(BSZ+2); // y-direction index including halo

int I_n = I_0 + J*N + I; //General index
u_prev_sh[I][J] = u_prev[I_n];
```

  - Load the remaining values

```
int ii2 = BSZ*BSZ + j*BSZ + i;
int I2  = ii2%(BSZ+2);
int J2  = ii2/(BSZ+2);

int I_n2 = I_0 + J2*N + I2; //General index

if ( (I2<(BSZ+2)) && (J2<(BSZ+2)) && (ii2 < N*N) )
        u_prev_sh[I2][J2] = u[I_n2];
```

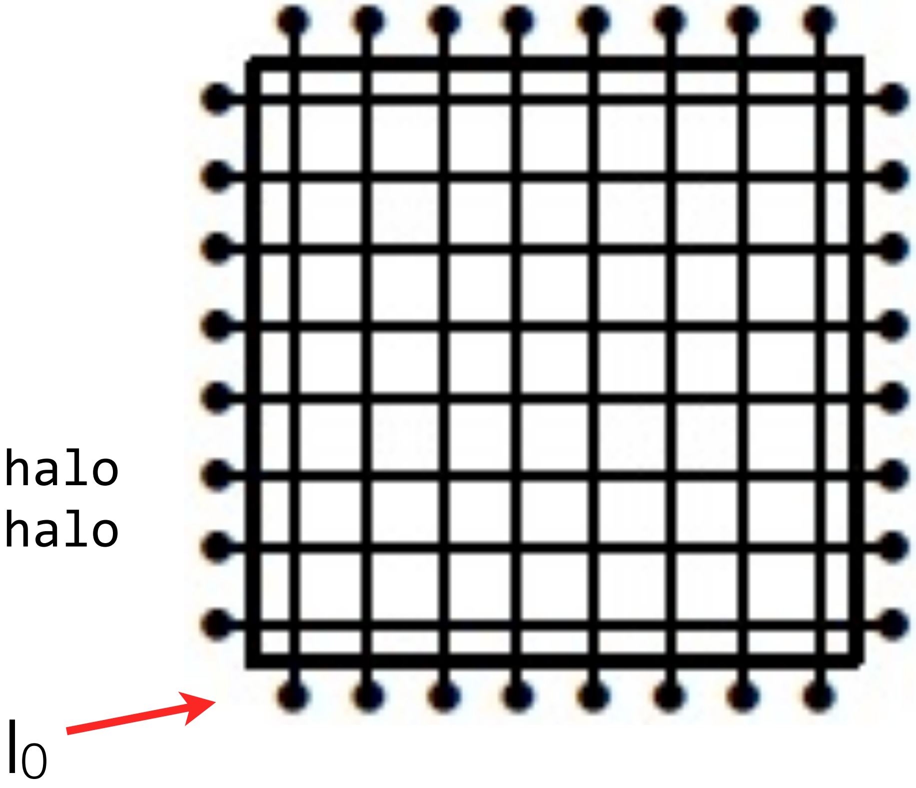


8x8 threads
10x10 loads

Shared Memory Implementation - Solution 3

- Loading in 2 steps
  - Use the 64 available threads to load the 64 first values to shared

```
__shared__ float u_prev_sh[BSZ+2][BSZ+2];

int ii = j*BSZ + i, // Flatten thread indexing
     I = ii%(BSZ+2), // x-direction index including halo
     J = ii/(BSZ+2); // y-direction index including halo

int I_n = I_0 + J*N + I; //General index
u_prev_sh[I][J] = u_prev[I_n];
```



8x8 threads

10x10 loads

  - Load the remaining values

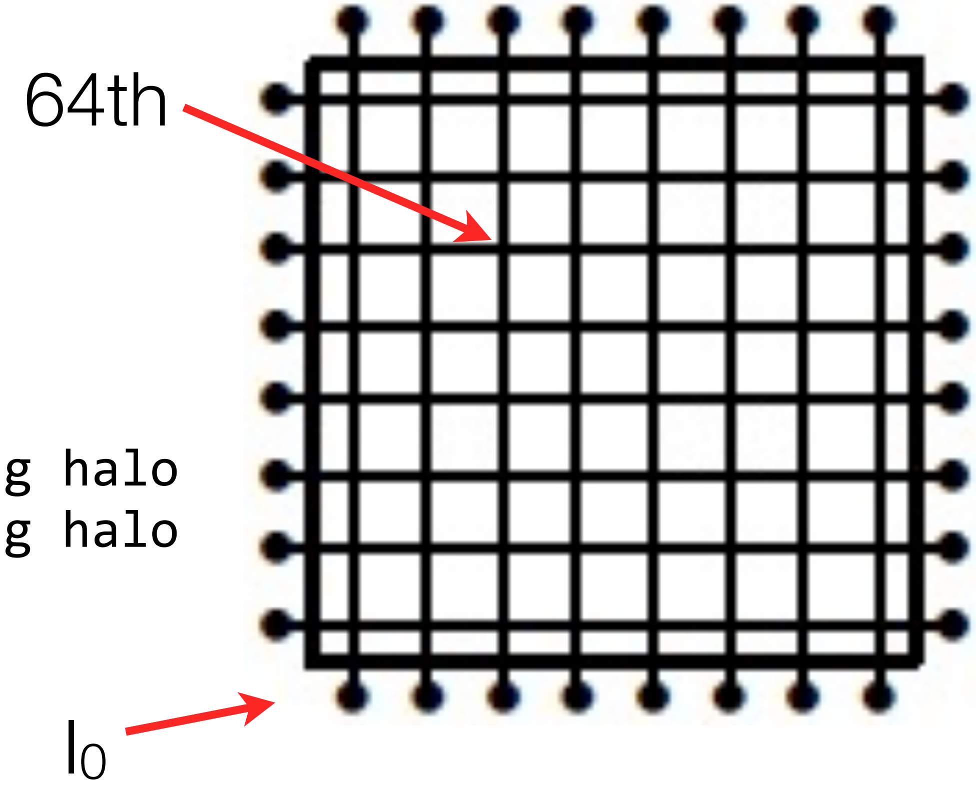
```
int ii2 = BSZ*BSZ + j*BSZ + i;
int I2  = ii2%(BSZ+2);
int J2  = ii2/(BSZ+2);

int I_n2 = I_0 + J2*N + I2; //General index

if ( (I2<(BSZ+2)) && (J2<(BSZ+2)) && (ii2 < N*N) )
        u_prev_sh[I2][J2] = u[I_n2];
```

Shared Memory Implementation - Solution 3

- Loading in 2 steps

  - Use the 64 available threads to load the 64 first values to shared

```
__shared__ float u_prev_sh[BSZ+2][BSZ+2];

int ii = j*BSZ + i, // Flatten thread indexing
     I = ii%(BSZ+2), // x-direction index including halo
     J = ii/(BSZ+2); // y-direction index including halo

int I_n = I_0 + J*N + I; //General index
u_prev_sh[I][J] = u_prev[I_n];
```



8x8 threads
10x10 loads

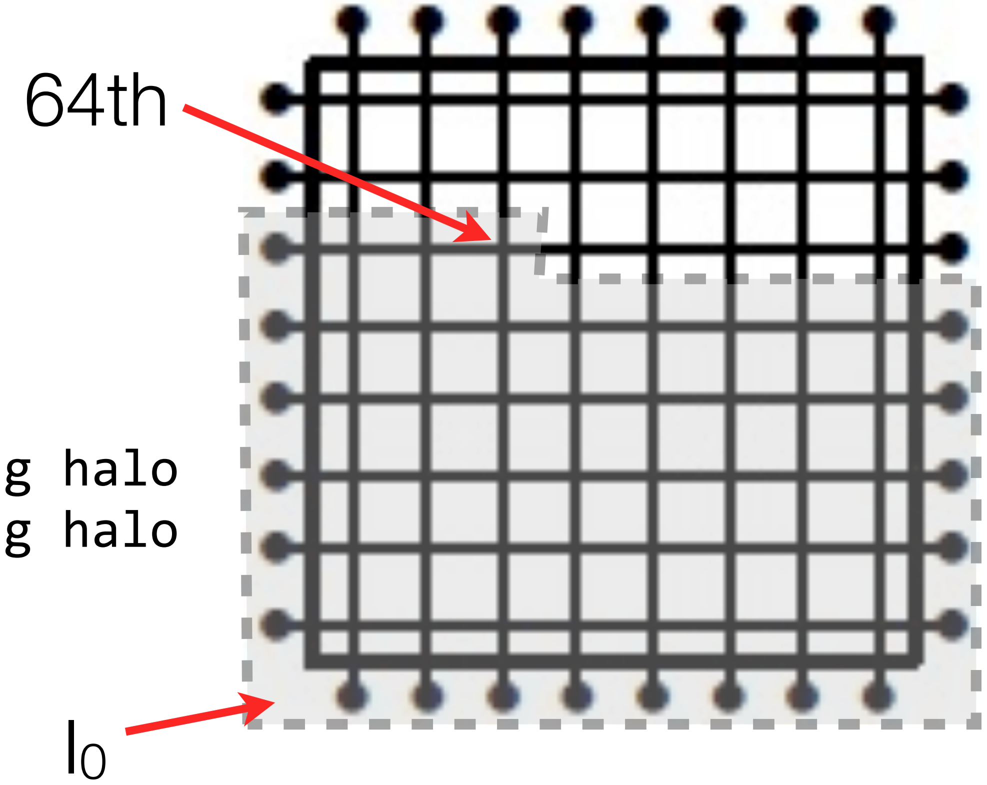
  - Load the remaining values

```
int ii2 = BSZ*BSZ + j*BSZ + i;
int I2  = ii2%(BSZ+2);
int J2  = ii2/(BSZ+2);

int I_n2 = I_0 + J2*N + I2; //General index

if ( (I2<(BSZ+2)) && (J2<(BSZ+2)) && (ii2 < N*N) )
        u_prev_sh[I2][J2] = u[I_n2];
```

Shared Memory Implementation - Solution 3

- Loading in 2 steps
  - Use the 64 available threads to load the 64 first values to shared

```
__shared__ float u_prev_sh[BSZ+2][BSZ+2];

int ii = j*BSZ + i, // Flatten thread indexing
     I = ii%(BSZ+2), // x-direction index including halo
     J = ii/(BSZ+2); // y-direction index including halo

int I_n = I_0 + J*N + I; //General index
u_prev_sh[I][J] = u_prev[I_n];
```



8x8 threads
10x10 loads

  - Load the remaining values

```
int ii2 = BSZ*BSZ + j*BSZ + i;
int I2  = ii2%(BSZ+2);
int J2  = ii2/(BSZ+2);

int I_n2 = I_0 + J2*N + I2; //General index

if ( (I2<(BSZ+2)) && (J2<(BSZ+2)) && (ii2 < N*N) )
        u_prev_sh[I2][J2] = u[I_n2];
```

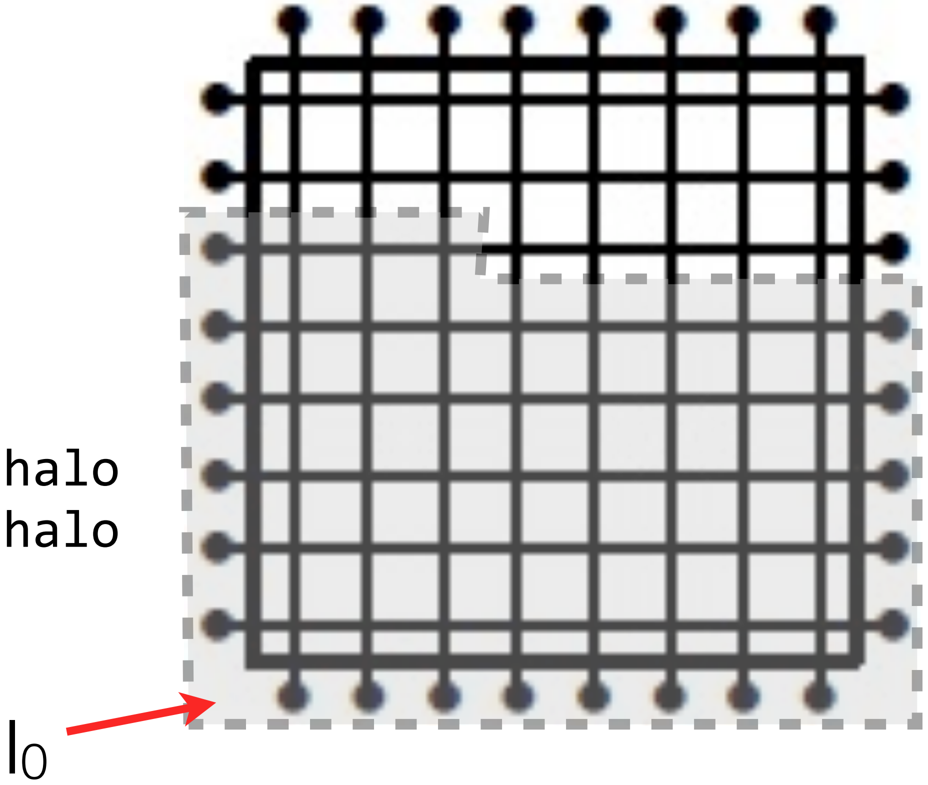
Shared Memory Implementation - Solution 3

- Loading in 2 steps
  - Use the 64 available threads to load the 64 first values to shared

```
__shared__ float u_prev_sh[BSZ+2][BSZ+2];

int ii = j*BSZ + i, // Flatten thread indexing
     I = ii%(BSZ+2), // x-direction index including halo
     J = ii/(BSZ+2); // y-direction index including halo

int I_n = I_0 + J*N + I; //General index
u_prev_sh[I][J] = u_prev[I_n];
```

  - Load the remaining values

```
int ii2 = BSZ*BSZ + j*BSZ + i;
int I2  = ii2%(BSZ+2);
int J2  = ii2/(BSZ+2);

int I_n2 = I_0 + J2*N + I2; //General index

if ( (I2<(BSZ+2)) && (J2<(BSZ+2)) && (ii2 < N*N) )
        u_prev_sh[I2][J2] = u[I_n2];
```



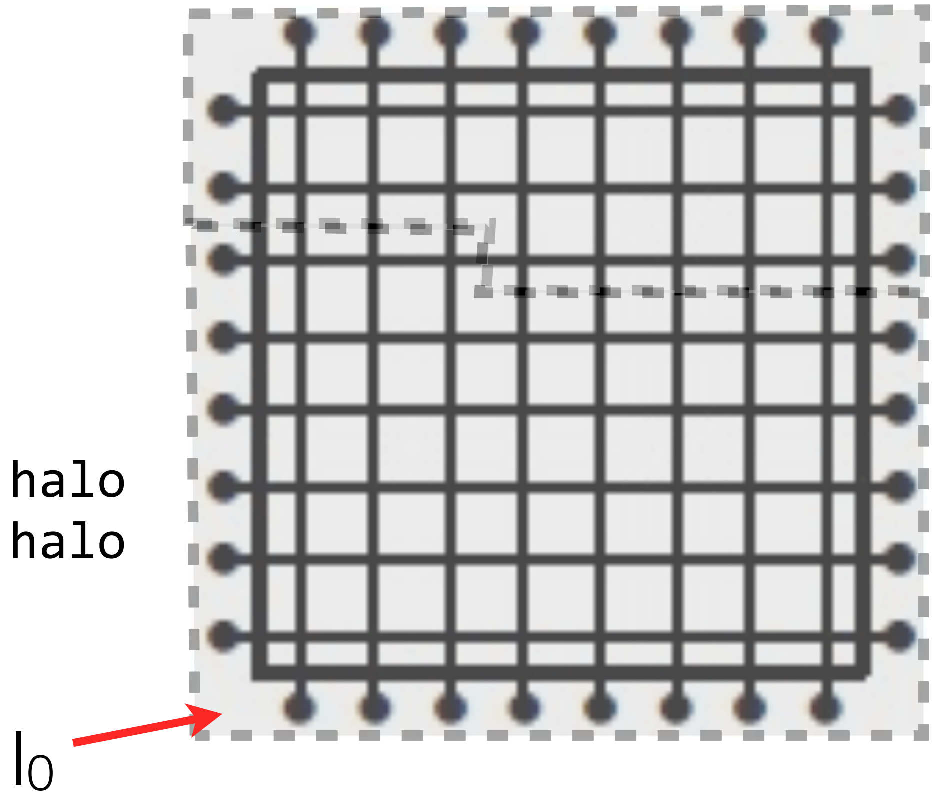
8x8 threads

10x10 loads

Shared Memory Implementation - Solution 3

- Loading in 2 steps
  - Use the 64 available threads to load the 64 first values to shared

```
__shared__ float u_prev_sh[BSZ+2][BSZ+2];

int ii = j*BSZ + i, // Flatten thread indexing
     I = ii%(BSZ+2), // x-direction index including halo
     J = ii/(BSZ+2); // y-direction index including halo

int I_n = I_0 + J*N + I; //General index
u_prev_sh[I][J] = u_prev[I_n];
```



8x8 threads
10x10 loads

  - Load the remaining values

```
int ii2 = BSZ*BSZ + j*BSZ + i;
int I2  = ii2%(BSZ+2);
int J2  = ii2/(BSZ+2);

int I_n2 = I_0 + J2*N + I2; //General index

if ( (I2<(BSZ+2)) && (J2<(BSZ+2)) && (ii2 < N*N) )
        u_prev_sh[I2][J2] = u[I_n2];
```

Shared Memory Implementation - Solution 3

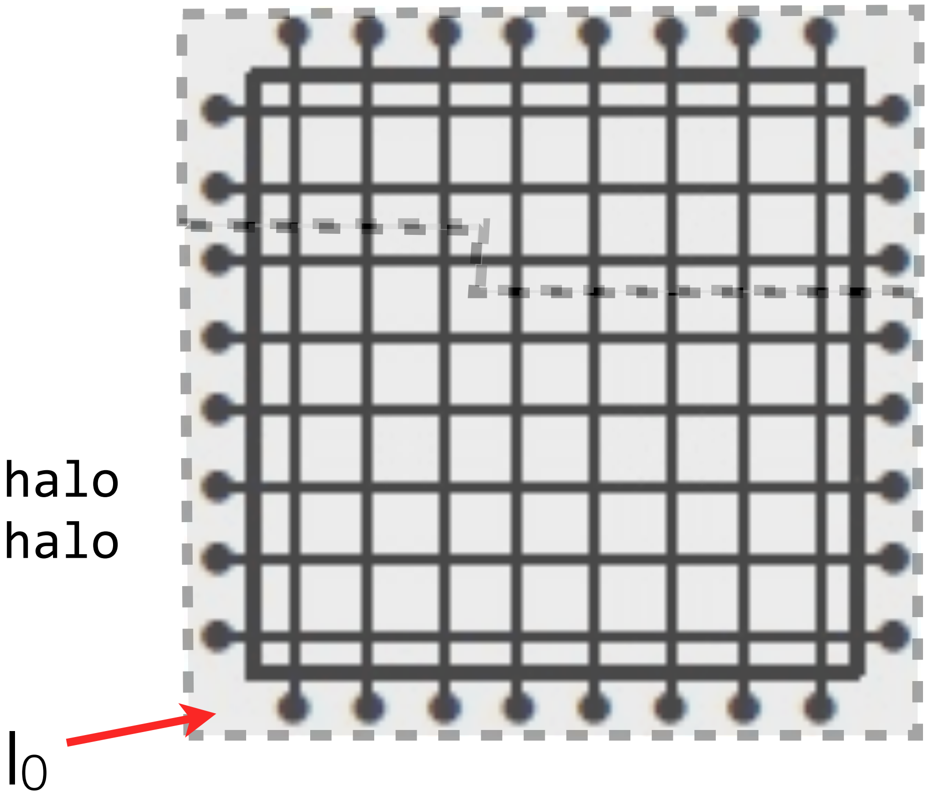
- Loading in 2 steps
  - Use the 64 available threads to load the 64 first values to shared

```
__shared__ float u_prev_sh[BSZ+2][BSZ+2];

int ii = j*BSZ + i, // Flatten thread indexing
     I = ii%(BSZ+2), // x-direction index including halo
     J = ii/(BSZ+2); // y-direction index including halo

int I_n = I_0 + J*N + I; //General index
u_prev_sh[I][J] = u_prev[I_n];
```

  - Load the remaining values

```
int ii2 = BSZ*BSZ + j*BSZ + i;
int I2  = ii2%(BSZ+2);
int J2  = ii2/(BSZ+2);

int I_n2 = I_0 + J2*N + I2; //General index

if ( (I2<(BSZ+2)) && (J2<(BSZ+2)) && (ii2 < N*N) )
        u_prev_sh[I2][J2] = u[I_n2];
```

Some threads won't load

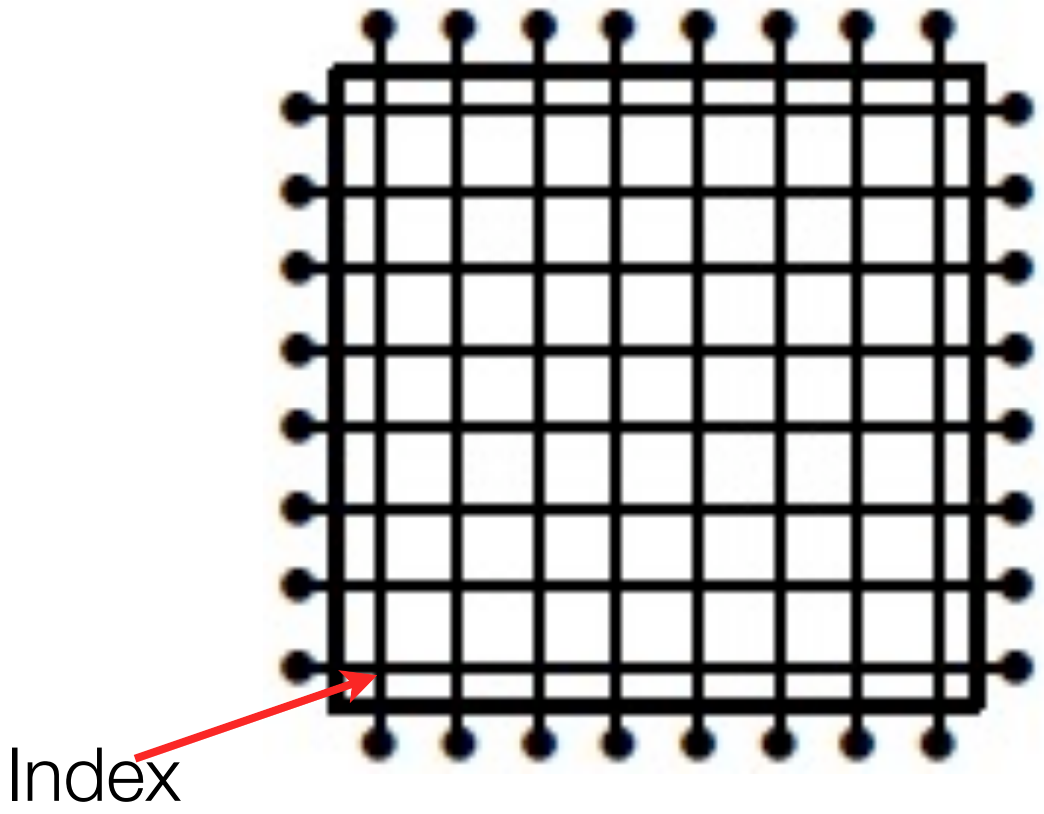


8x8 threads

10x10 loads

Shared Memory Implementation - Solution 3

- Compute on interior points: threads [i+1][j+1]



```
int Index = by*BSZ*N + bx*BSZ + (j+1)*N + i+1;

u[Index] = u_prev_sh[i+1][j+1] + alpha*dt/h/h * (u_prev_sh[i+2][j+1] + u_prev_sh[i][j+1] +
u_prev_sh[i+1][j+2] + u_prev_sh[i+1][j] - 4*u_prev_sh[i+1][j+1]);
```

SM Implementation

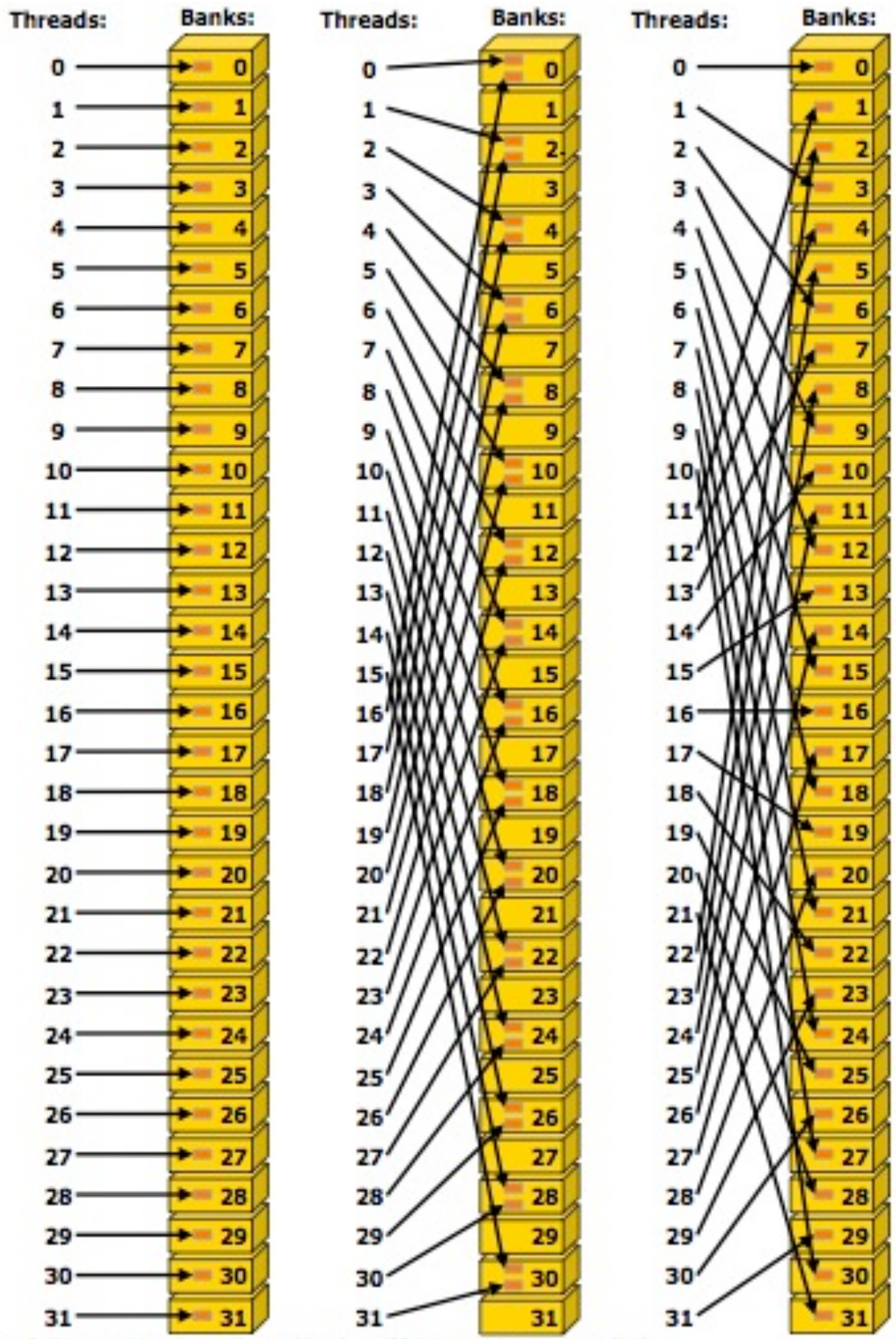
- The technique described is called **tiling**
  - Tiling means loading data to shared memory in tiles
  - Useful when shared memory is used as cache
  - Also used when all data is to large to fit in shared memory and you load it in smaller chunks

We will implement this in next lab!

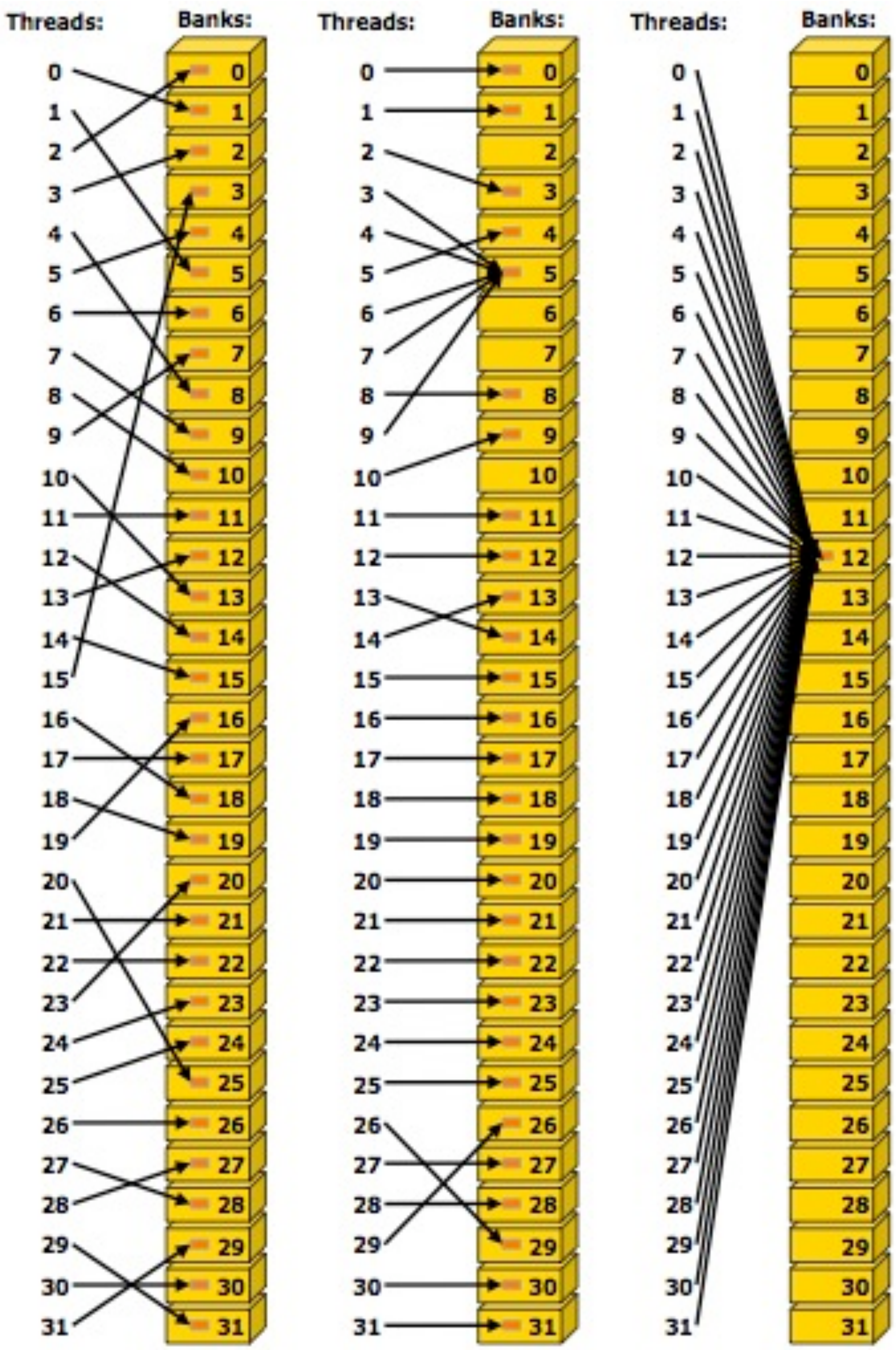
Shared Memory - Bank conflicts

- Shared memory arrays are subdivided into smaller subarrays called **banks**
- Shared memory has 32 (16) banks in 2.X (1.X). Successive 32-bit words are assigned to successive banks
- Different banks can be accessed simultaneously
- If two or more addresses of a memory request are in the same bank, the access is serialized
  - Bank conflicts exist only within a warp (half warp for 1.X)
- In 2.X there is no bank conflict if the memory request is for the same 32-bit word. This is not valid in 1.X.

Shared Memory - Bank conflicts



Left: Linear addressing with a stride of one 32-bit word (no bank conflict).
Middle: Linear addressing with a stride of two 32-bit words (2-way bank conflicts).
Right: Linear addressing with a stride of three 32-bit words (no bank conflict).



Left: Conflict-free access via random permutation.
Middle: Conflict-free access since threads 3, 4, 6, 7, and 9 access the same word within bank 5.
Right: Conflict-free broadcast access (all threads access the same word).

Shared Memory

- `__syncthreads()`
  - Barrier that waits for all threads of the block before continuing
  - Need to make sure all data is loaded to shared before access
  - Avoids **race conditions**
  - Don’t over use!

```
 u_shared[i] = u[I];
__syncthreads();

 if (i>0 && i<BLOCKSIZE-1)
        u[I] = u_shared[i] - c*dt/dx*(u_shared[i] - u_shared[i-1]);
```

Race condition

- When two or more threads want to access and operate on a memory location without syncronization

- Example: we have the value 3 stored in global memory and two threads want to add one to that value.

  - Possibility 1:
    - Thread 1 reads the value 3 adds 1 and writes 4 back to memory
    - Thread 2 reads the value 4 and writes 5 back to memory
  - Possibility 2:
    - Thread 1 reads the value 3
    - Thread 2 reads the value 3
    - Both threads operate on 3 and write back the value 4 to memory

- Solutions:

  - __syncthreads() or atomic operations

Race condition

- When two or more threads want to access and operate on a memory location without syncronization

- Example: we have the value 3 stored in global memory and two threads want to add one to that value.

  - Possibility 1:
    - Thread 1 reads the value 3 adds 1 and writes 4 back to memory
    - Thread 2 reads the value 4 and writes 5 back to memory ✓
  - Possibility 2:
    - Thread 1 reads the value 3
    - Thread 2 reads the value 3
    - Both threads operate on 3 and write back the value 4 to memory

- Solutions:

  - __syncthreads() or atomic operations

Race condition

- When two or more threads want to access and operate on a memory location without syncronization

- Example: we have the value 3 stored in global memory and two threads want to add one to that value.

  - Possibility 1:
    - Thread 1 reads the value 3 adds 1 and writes 4 back to memory
    - Thread 2 reads the value 4 and writes 5 back to memory ✓
  - Possibility 2:
    - Thread 1 reads the value 3
    - Thread 2 reads the value 3
    - Both threads operate on 3 and write back the value 4 to memory ✗

- Solutions:

  - __syncthreads() or atomic operations

Atomic operations

- Atomic operations deal with race conditions
  - It guarantees that while the operation is being executed, that location in memory os not accessed
  - Still we can't rely on any ordering of thread execution!
  - Types

```
atomicAdd
atomicSub
atomicExch
atomicMin
atomicMax
etc...
```

Atomic operations

```
__global__ update (int *values, int *who)
{
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  int I = who[i];

  atomicAdd(&values[I], 1);
}
```

Atomic operations

- Useful if you have a sparse access pattern
- Atomic operations are slower than “normal” function
- They can serialize your execution if many threads want to access the same memory location
  - Think about parallelizing your data, not only execution
  - Use hierarchy of atomic operations to avoid this
- Prefer __syncthreads() if you can use it instead
  - If you have a regular access pattern