

CSE 160

Lecture 25

Programming
Graphical Processing Units

Announcements

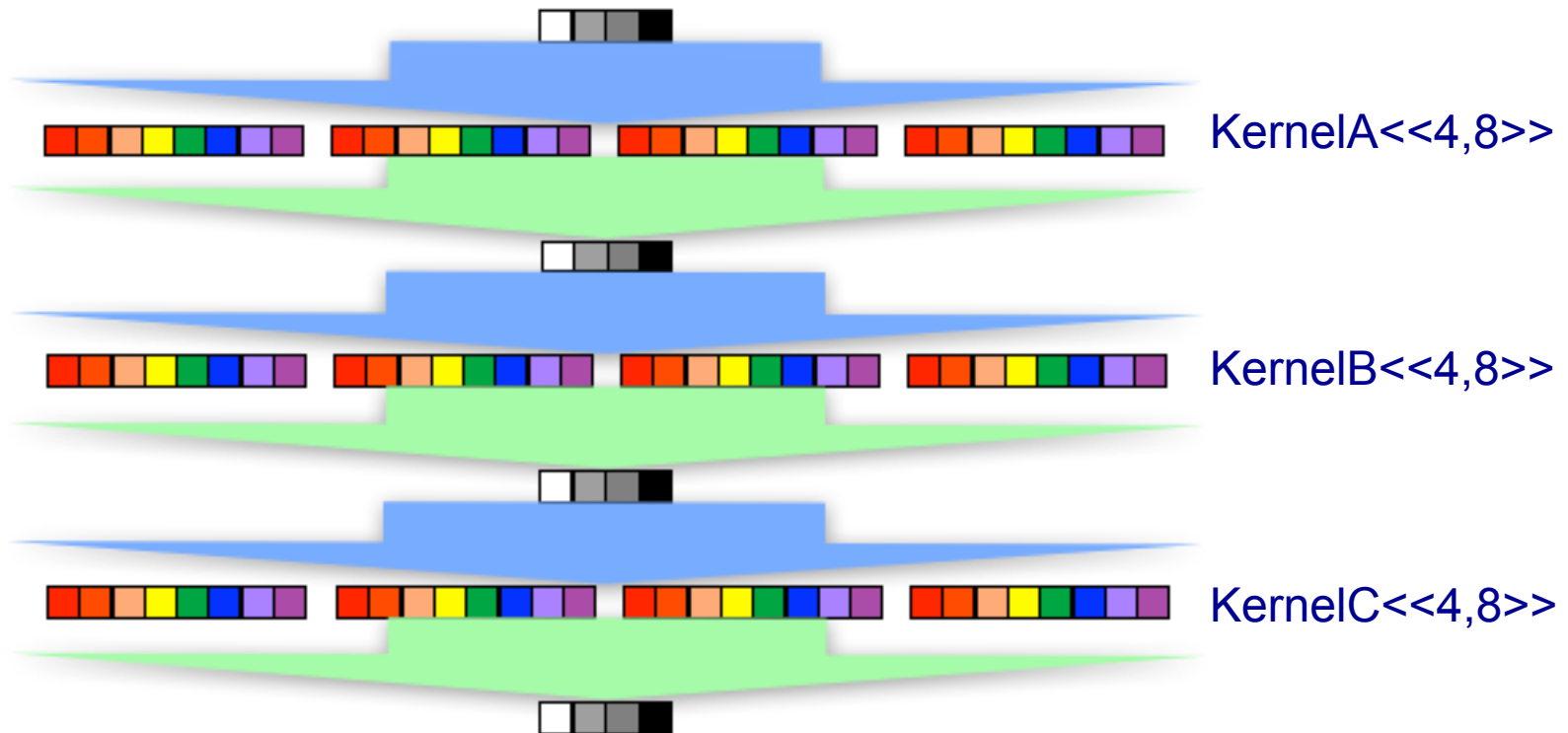
- Midterm review on Weds @Rady school
- Section: Thursday @ 5pm csb 004
 - Bring question for me to answer, problems to work out

Today's lecture

- Programming with CUDA
 - Avoiding thread divergence
- Aliev Panfilov method in CUDA

CUDA

- Programming environment with extensions to C
- Under control of the *host*, invoke sequences of multithreaded *kernels* on the *device* (GPU)
- Many lightweight *virtualized* threads
- CUDA: programming environment + C extensions

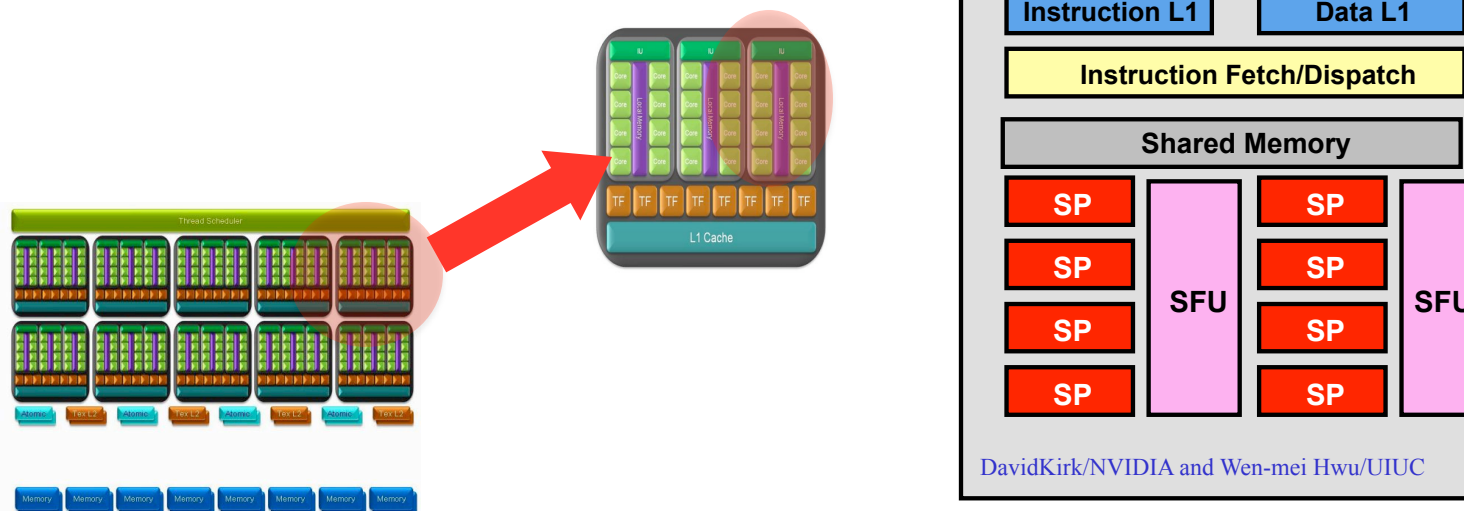


The CUDA Programmer's Goals

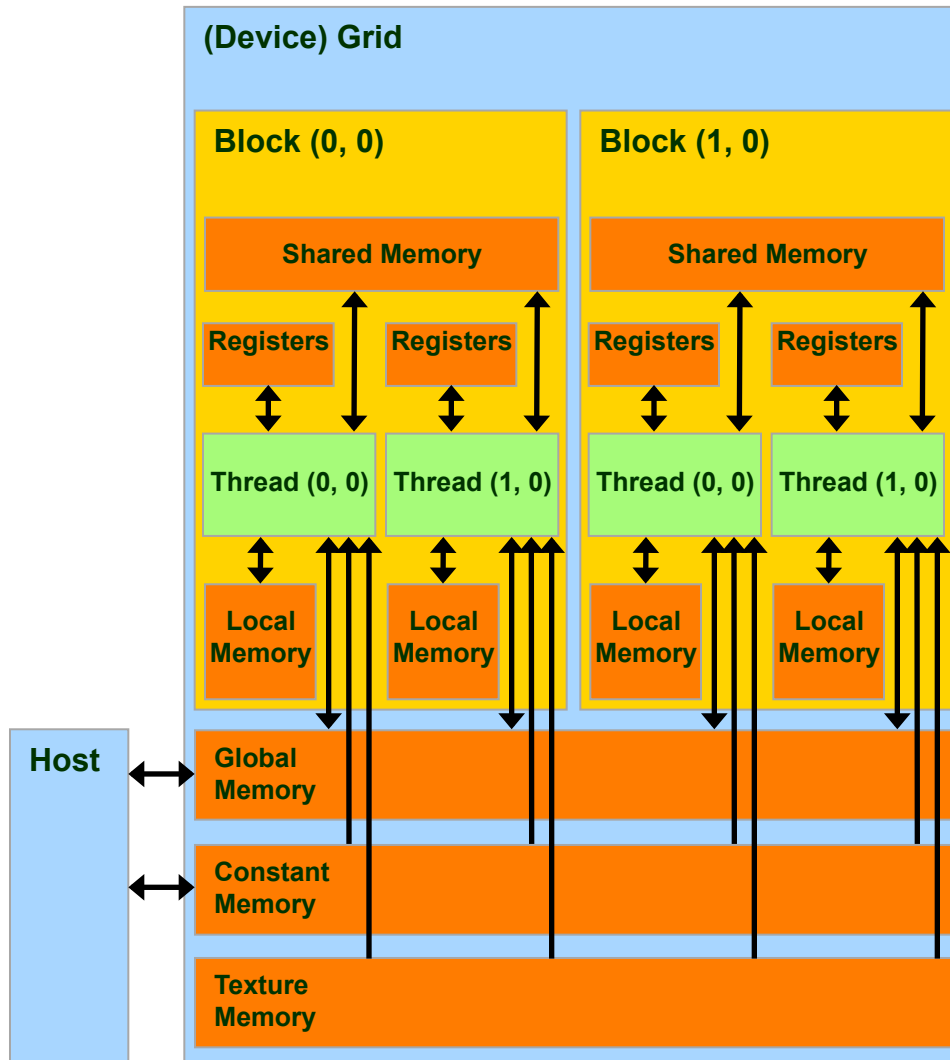
- Use the parallelism, else we lose it
- Avoid costly branches, or render them harmless
- **Cut data motion costs**
 - Hide latency of host $\leftrightarrow$ device memory transfers
 - Global memory accesses $\rightarrow$ fast on-chip accesses
 - Coalesced memory transfers
- Tradeoff: more blocks with fewer threads or more threads with fewer blocks
 - Locality: want small blocks of data (and hence more plentiful warps) that fit into fast memory
 - Register consumption
 - Scheduling: hide latency

Streaming processor cluster

- GTX-280 GPU
 - 10 clusters @ 3 streaming multiprocessors or *vector cores*
- Each vector core
 - 8 scalar cores: fused multiply adder + multiplier (32 bits), truncate intermediate result
 - Shared memory (16KB) and registers (16K × 32 bits = 64KB)
 - 1 64-bit fused multiply-adder + 2 super function units (2 fused multiply-adders)
 - 1 FMA + 1 multiply per cycle = 3 flops / cycle / core * 240 cores = 720 flops/cycle @ 1.296 Ghz: 933 GFLOPS

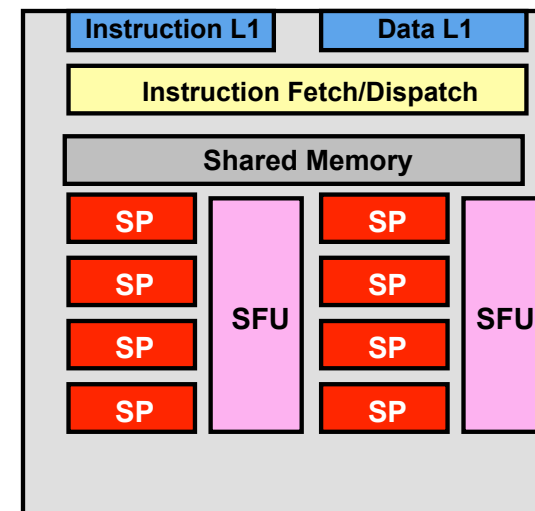


Memory Hierarchy



Courtesy DavidKirk/NVIDIA and Wen-mei Hwu/UIUC

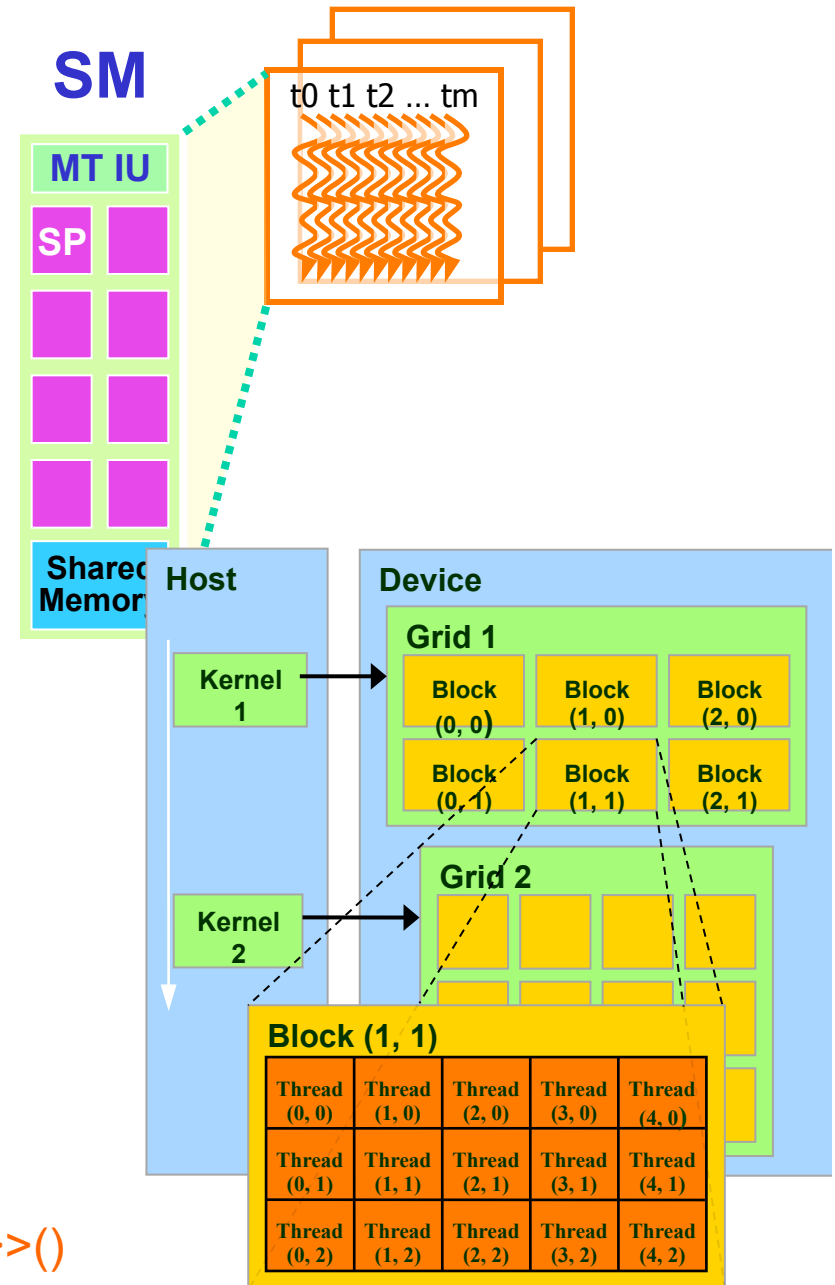
Name	Latency (cycles)	Cached
Global	DRAM – 100s	No
Local	DRAM – 100s	No
Constant	1s – 10s – 100s	Yes
Texture	1s – 10s – 100s	Yes
Shared	1	--
Register	1	--



Thread execution

- Thread Blocks
 - ▶ Unit of workload assignment
 - ▶ Each thread has its own set of registers
 - ▶ All have access to a fast on-chip *shared memory*
 - ▶ Synchronization only among all threads in a block
 - ▶ Threads in different blocks communicate via slow global memory
 - ▶ Processor groups threads into *warps* of 32 threads
- SIMT parallelism: all threads in a warp execute the same instruction
 - ▶ All branches followed
 - ▶ Instructions disabled
 - ▶ Divergence, serialization

KernelA<<<2,3>,<3,5>>>()
 Grid Block



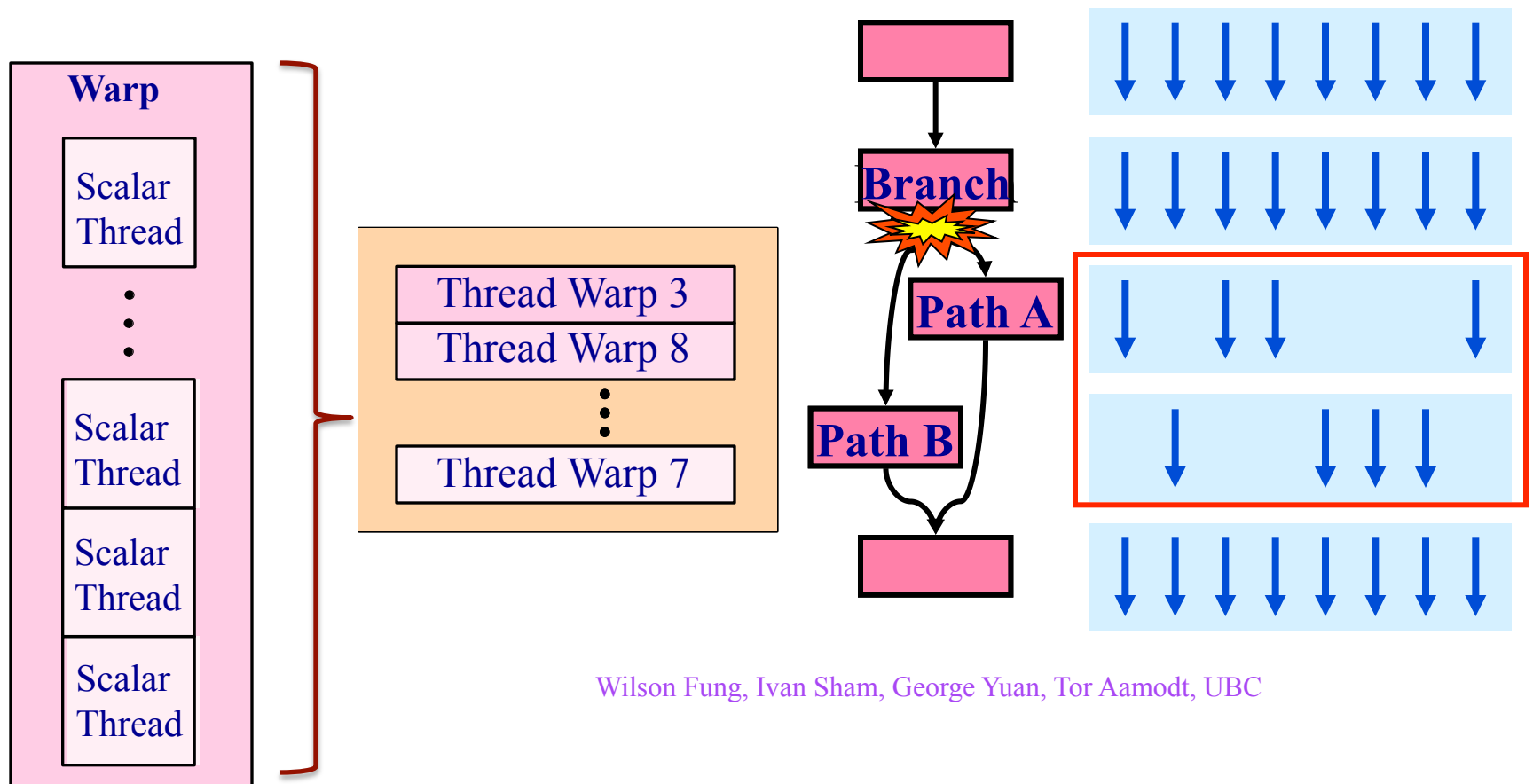
DavidKirk/NVIDIA & Wen-mei Hwu/UIUC

Today's lecture

- Programming with CUDA
 - Avoiding thread divergence
- Aliev Panfilov method in CUDA

Thread divergence

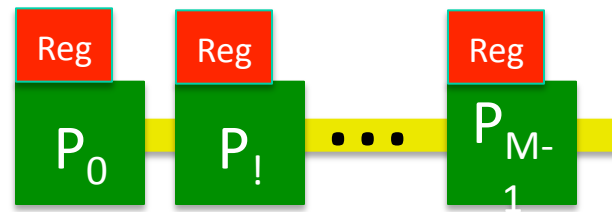
- All the threads in a warp execute the same instruction
- Different control paths are serialized



Wilson Fung, Ivan Sham, George Yuan, Tor Aamodt, UBC

Divergence example

```
if (threadIdx >= 2)
    a=100;
else
    a=-100;
```

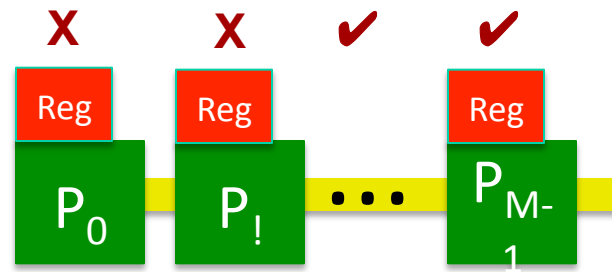


compare
threadIdx,2

Mary Hall

Divergence example

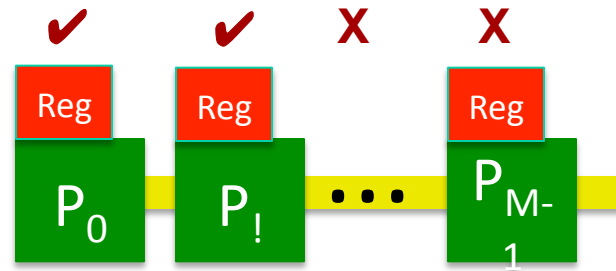
```
if (threadIdx >= 2)
    a=100;
else
    a=-100;
```



Mary Hall

Divergence example

```
if (threadIdx >= 2)
    a=100;
else
    a=-100;
```



Mary Hall

Example of thread Divergence

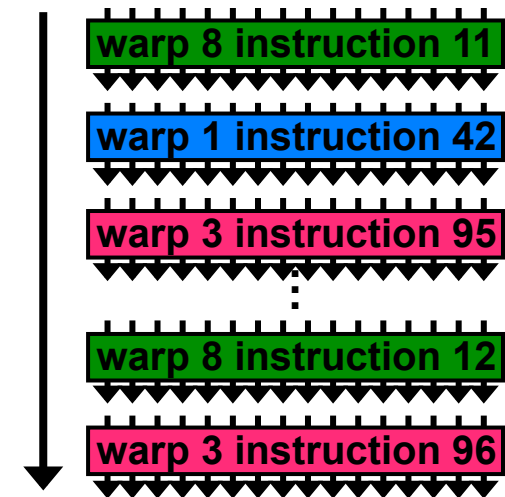
- Divergence when predicate is a function of the threadId, branch granularity > warp size

`if (threadId < 2) { }`

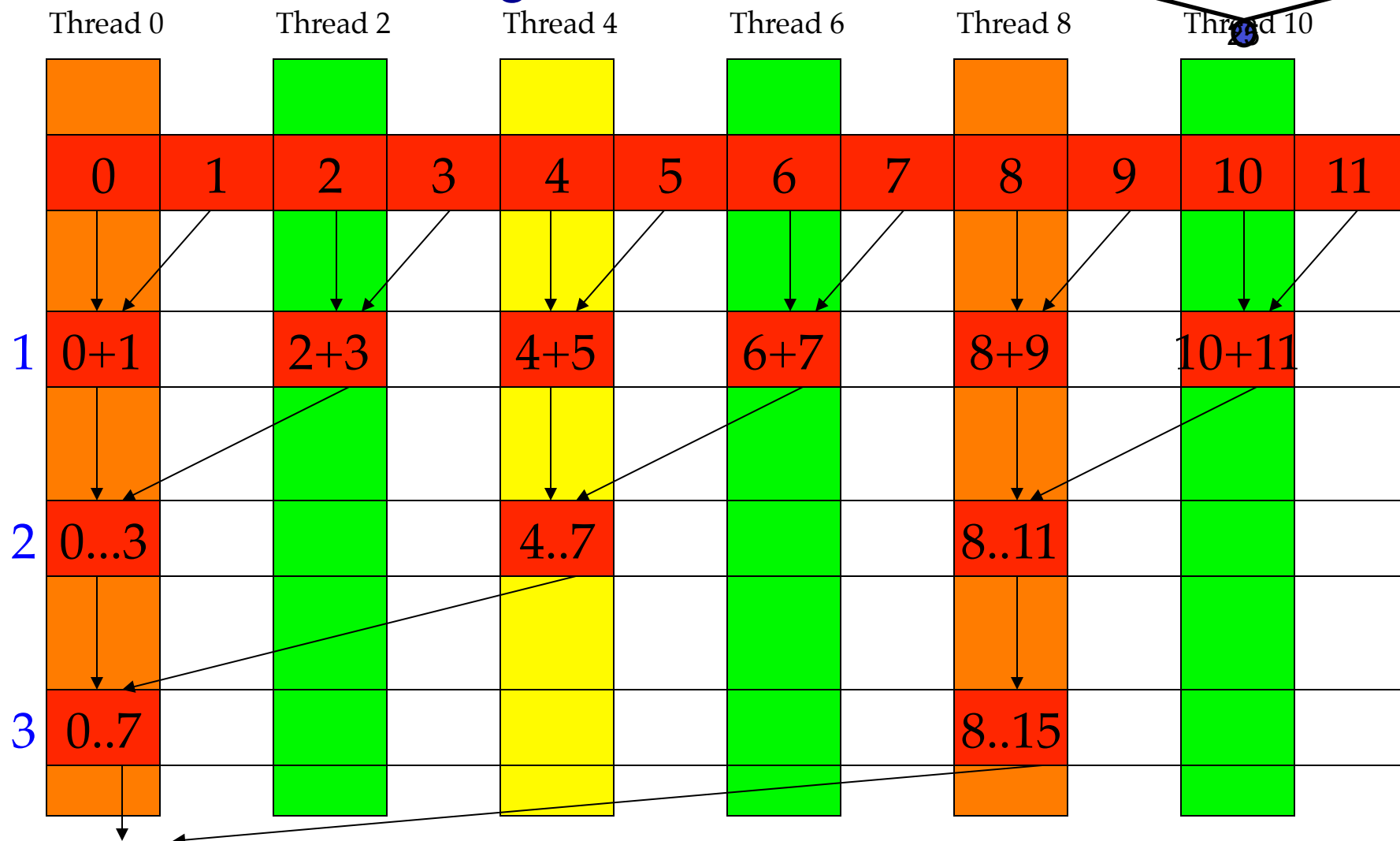
- No divergence if all follow the same path within a warp
branch granularity > warp size

`if (threadId / WARP_SIZE < 2) { }`

- But there are different control paths within the block
- Consider reduction, e.g. summation $\sum_i x_i$



Example – reduction – thread divergence

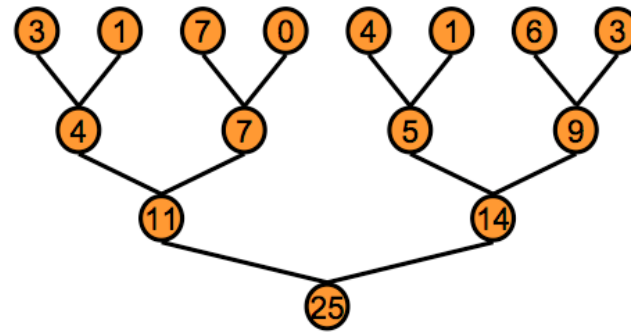


DavidKirk/NVIDIA & Wen-mei Hwu/UIUC

The naïve code

```
__global__ void reduce(int *input, unsigned int N, int *total){  
    unsigned int tid = threadIdx.x;  
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
```

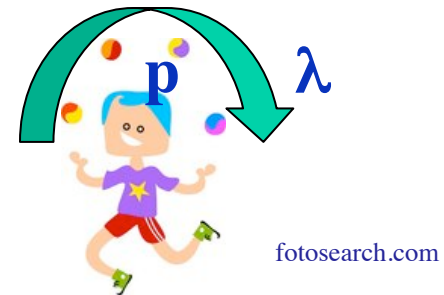
```
    __shared__ int x[BSIZE];  
    x[tid] = (i < N) ? input[i] : 0;  
    __syncthreads();
```



```
    for (unsigned int stride = 1; stride < blockDim.x; stride *= 2) {  
        __syncthreads();  
        if (tid % (2*stride) == 0)  
            x[tid] += x[tid + stride];  
    }  
    if (tid == 0) atomicAdd(total, x[tid]);  
}
```

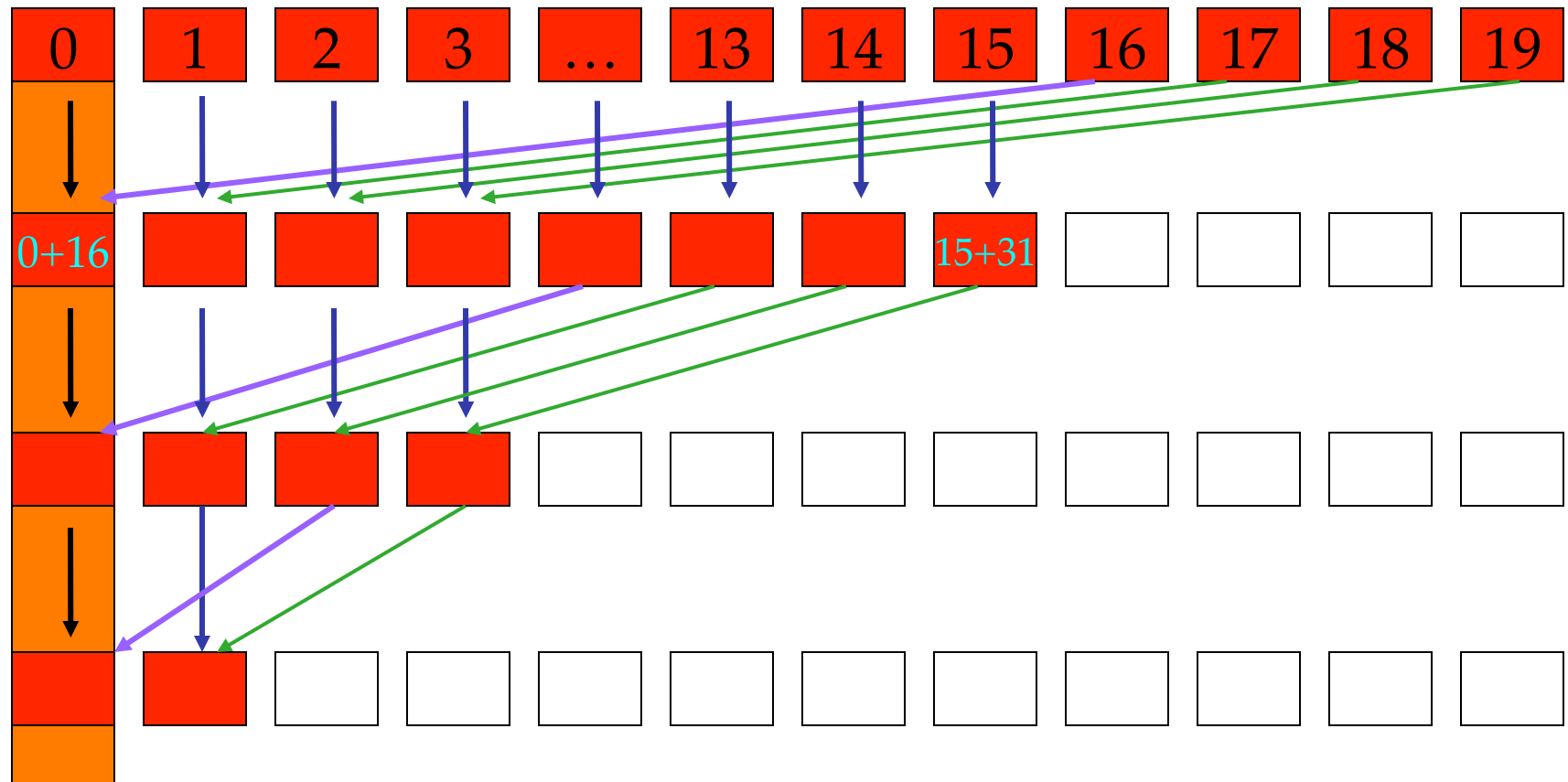
Occupancy

- A minimum number of warps needed to hide memory latency
- **Occupancy: $\# \text{ active warps} \div \text{max } \# \text{ warps supported by vector unit}$**
- Limited by vector unit resources
 - Amount of shared memory
 - Number of registers
 - Maximum number of threads
- Consider a kernel (16x16 block size)
 - Shared memory/block = 2648 bytes
 - Reg/thread=38 [38*256 =9728 < 16k]
 - # available registers is the limiting factor
- Tradeoff: more blocks with fewer threads or more threads with fewer blocks
 - Locality: want small blocks of data (and hence more plentiful warps) that fit into fast memory
 - Register consumption
- Maximizing the occupancy may not maximize performance



Reducing divergence and (also avoids memory bank conflicts)

Thread 0



The improved code

- All threads in a warp execute the same instruction
`reduceSum <<<N/512,512>>> (x,N)`
- No divergence until stride < 32
- All warps active when stride ≥ 32

```
__shared__ int x[ ];  
    unsigned int tid = threadIdx.x;  
    unsigned int s;  
  
for (stride = blockDim.x/2;  
    stride > 1;  
    stride /= 2) {  
    __syncthreads();  
  
    if (tid < stride)  
        x[tid] += x[tid + stride];  
}
```

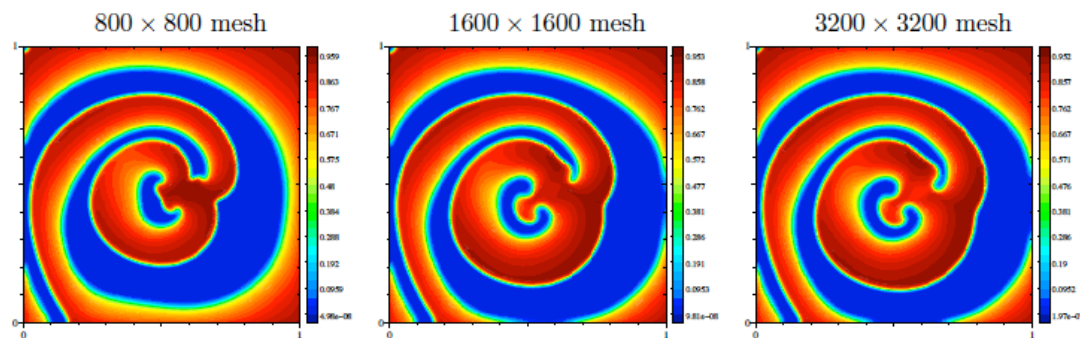
```
for (stride = 1;  
    stride < blockDim.x;  
    stride *= 2) {  
    __syncthreads();  
  
    if (tid % (2*stride) == 0)  
        x[tid] += x[tid + stride];  
}
```

Today's lecture

- Programming with CUDA
 - Avoiding thread divergence
- Aliev Panfilov method in CUDA

The Aliev-Panfilov Method

- Models signal propagation in cardiac tissue
 - Demonstrates complex behavior of spiral waves that are known to cause life-threatening situations
- Reaction-diffusion system
 - Reactions are the cellular exchanges of certain ions across the cell membrane during the cellular electrical impulse
- Our simulation has two state variables
 - Transmembrane potential: e
 - Recovery of the tissue: r



The Aliev-Panfilov Model

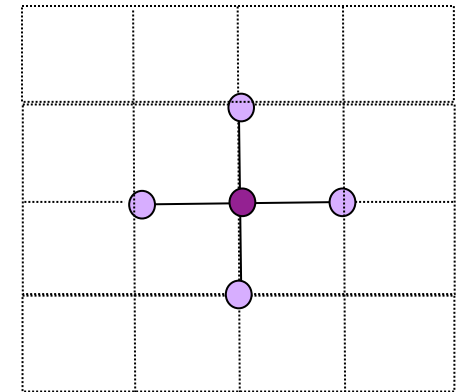
- Two parts
 - 2 Ordinary Differential Equations
 - Kinetics of reactions occurring at every point in space
 - Partial Differential Equation
 - Spatial diffusion of reactants
- First-order explicit numerical scheme

$$\begin{aligned}\frac{\partial e}{\partial t} &= \delta \nabla^2 e - ke(e - a)(e - 1) - er, & \text{on } \Omega_T, \\ \frac{\partial r}{\partial t} &= - \left[\varepsilon + \frac{\mu_1 r}{\mu_2 + e} \right] [r + ke(e - b - 1)], & \text{on } \Omega_T,\end{aligned}$$

$$\vec{n} \cdot \delta \nabla e = 0 \text{ on } \partial\Omega, \quad \text{and} \quad (e, r)_{t=0} = (e(\cdot, 0), r(\cdot, 0)),$$

Data Dependencies

- ODE solver:
 - No data dependency, trivially parallelizable
 - Requires a lot of registers to hold temporary variables
- PDE solver:
 - Jacobi update for the 5-point Laplacian operator.
 - Sweeps over a uniformly spaced mesh
 - Updates voltage to weighted contributions from the 4 nearest neighbors



```

for (j=1; j<=m+1; j++){
  double *RR = &R[j][1], *EE = &E[j][1];
  for (i=1; i<=n+1; i++, EE++, RR++) {
    // PDE Solver
    E[0]= E_p[j][i]+α*(E_p[j][i+1]+E_p[j][i-1]-4*E_p[j][i]+E_p[j+1][i]+E_p[j-1][i]);
    // ODE Solver
    EE[0] += -dt*(kk*EE[0]*(EE[0]-a)*(EE[0]-1)+EE[0]*RR[0]);
    RR[0] += dt*(ε+M1* RR[0]/( EE[0]+M2))*(-RR[0]-kk*EE[0]*(EE[0]-b-1));
  }
}

```

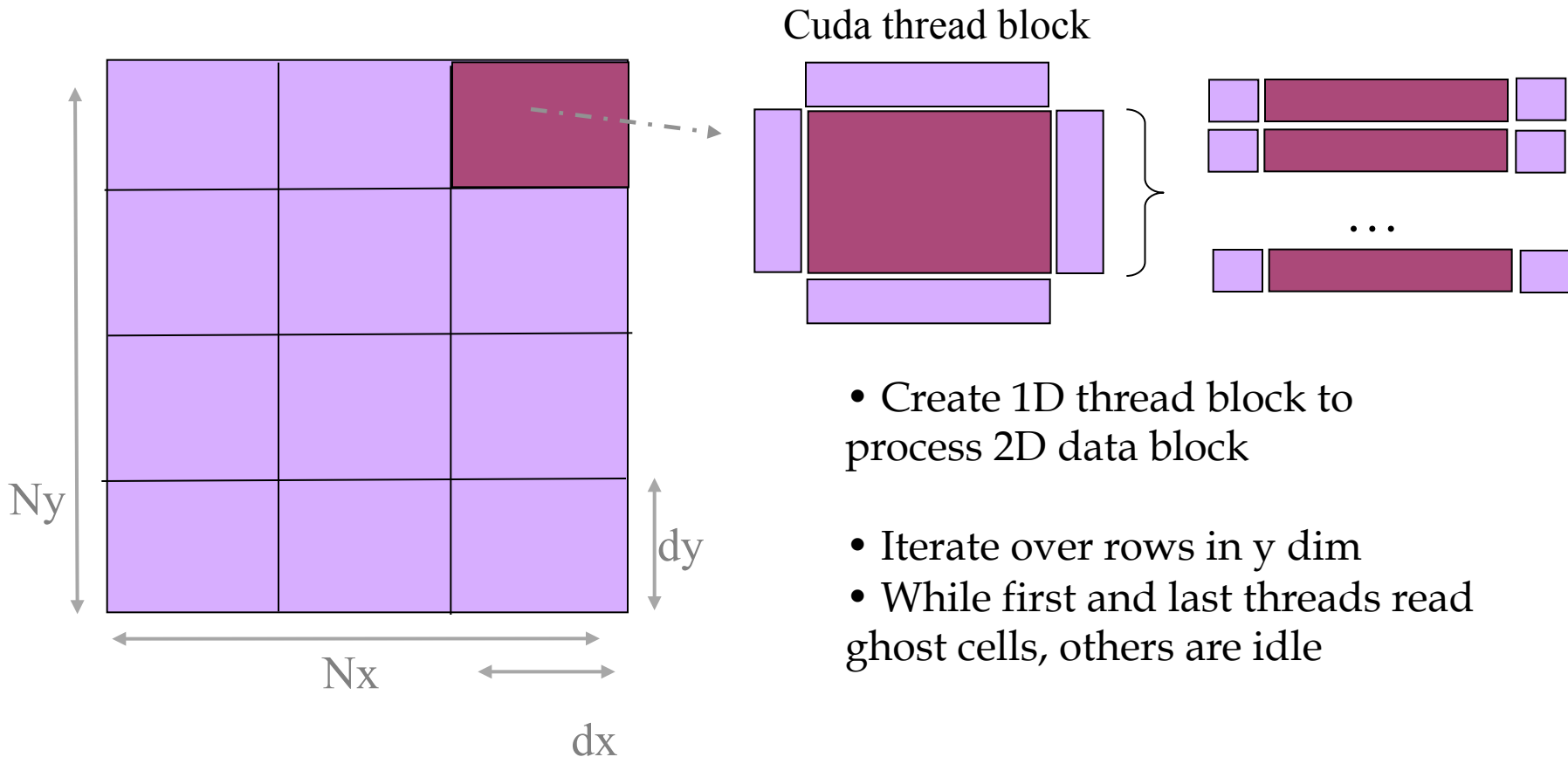
Naïve CUDA Implementation

- All array references go through device memory
- ./apf -n 6144 -t 0.04, 16x16 thread blocks
 - Tesla C1060, capability: 1.3 [[E5504](#)]
 - SP: 22, 73, 34 GFlops [[E5504](#), 32 cores, MPI: 85GF]
 - DP: 13, 45, 20 GFlops (19GF n=8192) [[E5504](#): 48 GF]

```
#define E' [i,j] E_prev[(j+1)*(m+3) + (i+1)]
I = blockIdx.y*blockDim.y + threadIdx.y;
J = blockIdx.x*blockDim.x + threadIdx.x;
if ((I <= n) && (J <= m) )
    E[I]=E' [I,J] +  $\alpha$ *(E'[I-1,J] + E'[I+1,J] + E'[I,J-1]+E' [I,J+1] - 4*E' [I,J]);

for (j=1; j<= m+1; j++)
    for (i=1; i<= n+1; i++)
        E[j][i] = E' [j][i]+ $\alpha$ *(E' [j][i-1]+E' [j][i+1] +
                                E' [j-1][i]+E' [j+1][i] - 4*E' [j][i]);
```

Using Shared Memory

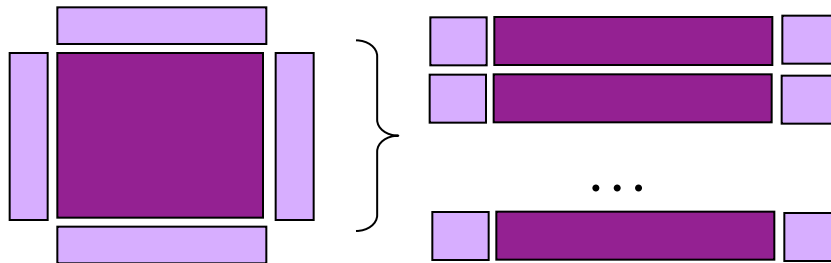


Compared to a 2D thread blocking, 1D thread blocks provide a 12% improvement in double precision and 64% improvement in single precision

Didem Unat

Sliding rows

Sliding rows with 1D thread blocks reduces global memory accesses.



Top row <-- Curr row,
 Curr row <-- Bottom row
 Bottom row <-- read new row from
 global memory

Sliding row algorithm

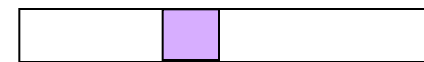
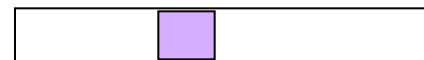
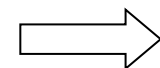
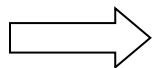
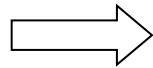
Top Row in Registers



Curr Row in Shared memory



Bottom Row in Registers



Read new row from
 global memory

CUDA Code

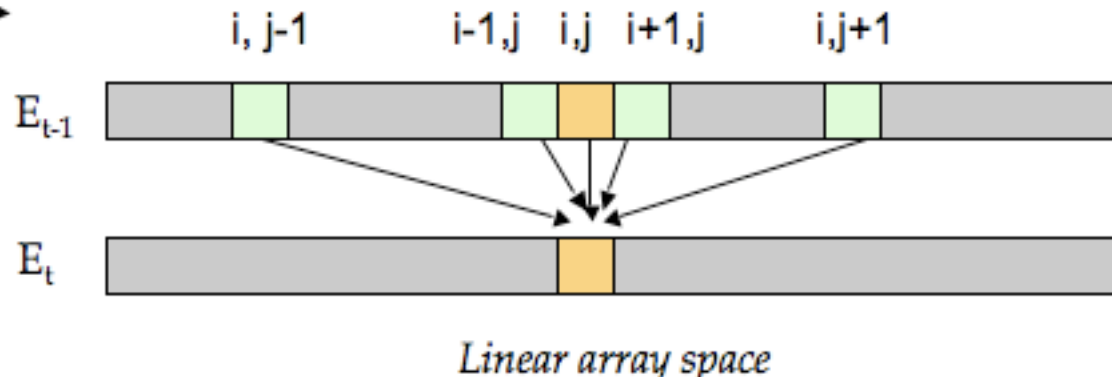
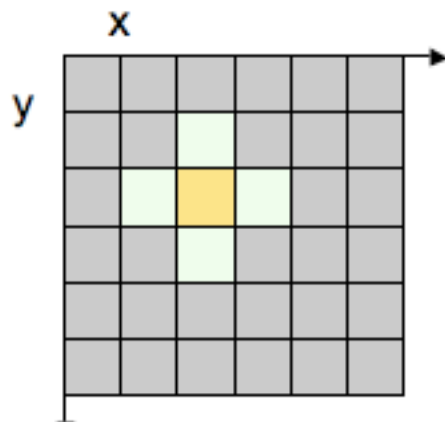
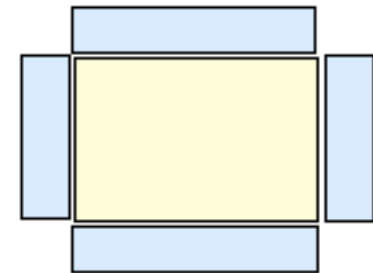
```
__shared__ float block[DIM_Y + 2 ][DIM_X + 2 ];  
int idx = threadIdx.x, idy = threadIdx.y ; //local indices  
//global indices  
int x = blockIdx.x * (DIM_X) + idx;  
int y = blockIdx.y * (DIM_Y) + idy;  
idy++; idx++;  
unsigned int index = y * N + x ;  
  
//interior points  
float center = E_prev[index] ;  
block[idy][idx] = center;  
  
__syncthreads();
```

Copying the ghost cells

When loading ghost cells, only some of the threads are active, some are idle

```

if (idy == 1 && y > 0 )
    block[0][idx] = E_prev[index - N];
else if(idy == DIM_Y && y < N-1)
    block[DIM_Y+1][idx] = E_prev[index + N];
if ( idx==1 && x > 0 )
    block[idy][0] = E_prev[index - 1];
else if( idx== DIM_X && x < N-1 )
    block[idy][DIM_X + 1] = E_prev[index + 1];
__syncthreads();
    
```



The stencil computation and ODE

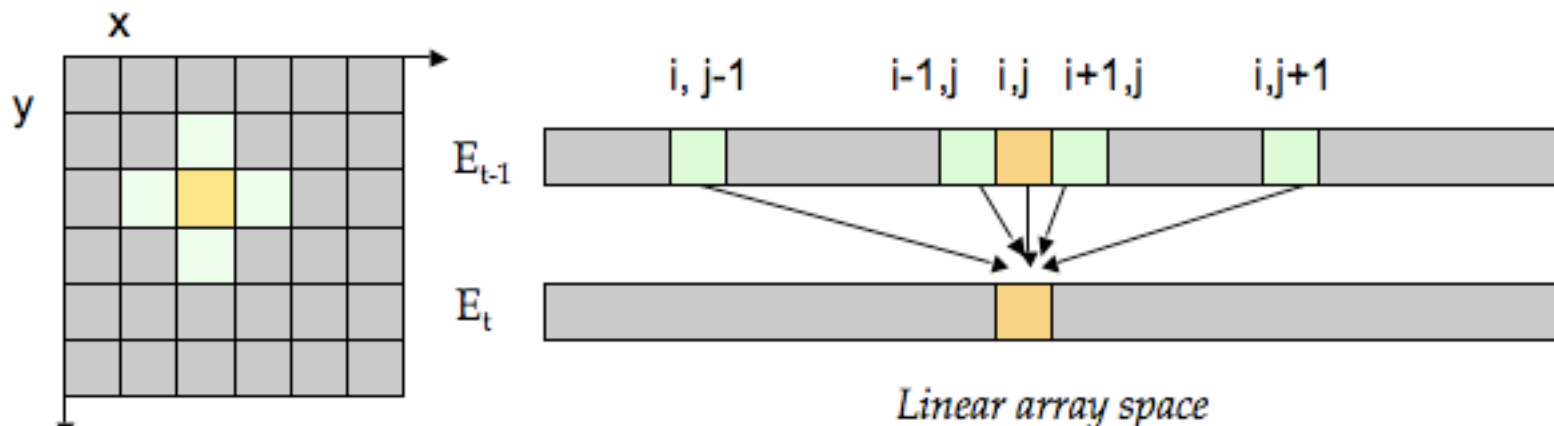
```
float r = R[index];
```

```
float e = center +  $\alpha$  * (block[idy][idx-1] + block[idy][idx+1] +  
                        block[idy-1][idx] + block[idy+1][idx] - 4*center);
```

```
e = e - dt*(kk * e * ( e- a) * ( e - 1 ) + e * r);
```

```
E[index] = e;
```

```
R[index] = r + dt * ( $\epsilon$  + M1 * r / ( e + M2 ) ) * ( -r - kk * e * (e - b - 1));
```



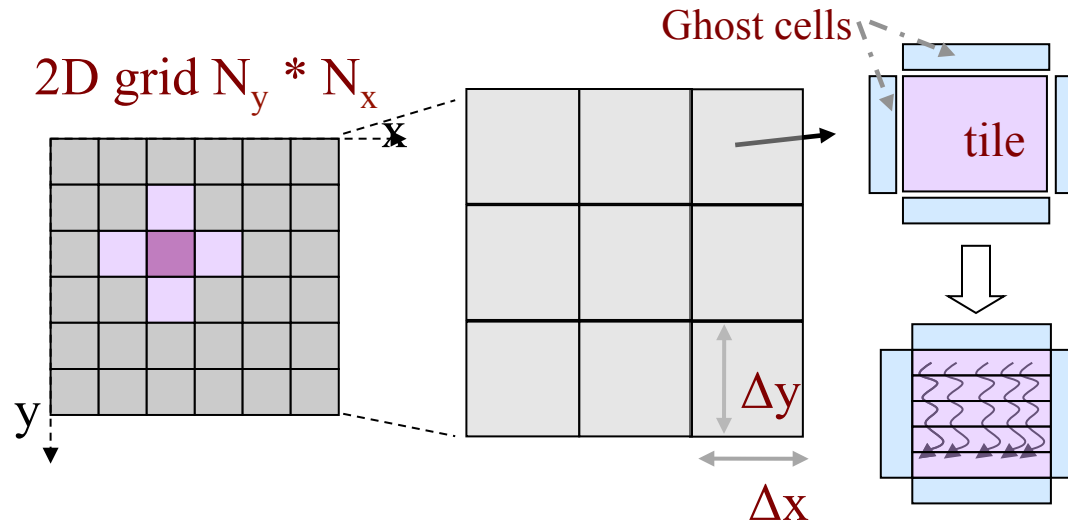
Results on C1060 (Tesla)

GFlop/s rates for Nehalem and C1060 implementations

N=4K	1 CPU-only	1 GPU	Speedup over CPU
Single	2.36	124.48	52.84
Double	2.01	25.69	12.81
N=16K	4 CPUs-only	4 GPUs	Speedup over CPU
Single	7.79	454.47	58.32
Double	3.88	102.48	26.41

- Single Precision
 - Nearly saturates the off-chip memory bandwidth
 - Utilizing **98%** of the sustainable bandwidth for the Tesla C1060.
 - Achieves **13.3%** of the single precision peak performance
 - Single precision performance is bandwidth limited.
- Double Precision
 - 41.5% of the sustainable bandwidth
 - 1/3 of the peak double precision performance
 - Performance hurt by the division operation that appears in ODE

Memory Accesses



Total Memory Accesses = $4N^2 + \text{ghost cells}$

Number of Tiles = $N / \Delta x * N / \Delta y$

Total Memory Accesses = $4N^2 + N^2 / (\Delta x \Delta y) \times 2(\Delta y + \Delta y) \times \text{sizeof(element)}$

Estimated Kernel Time = Total Mem. Acc. (bytes) / Empirical Device Bandwidth