

Lecture 10

Stencil Methods Limits to Performance

Announcements

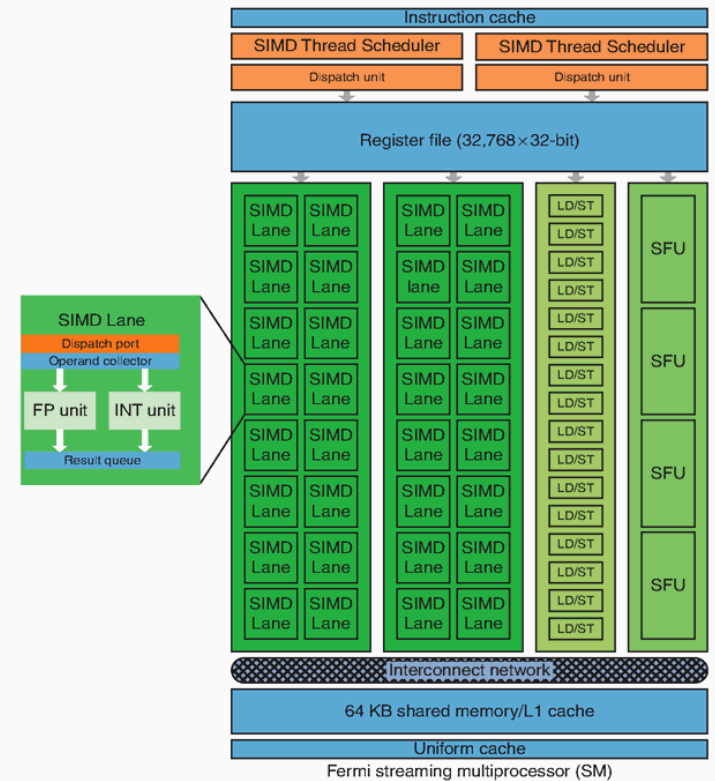
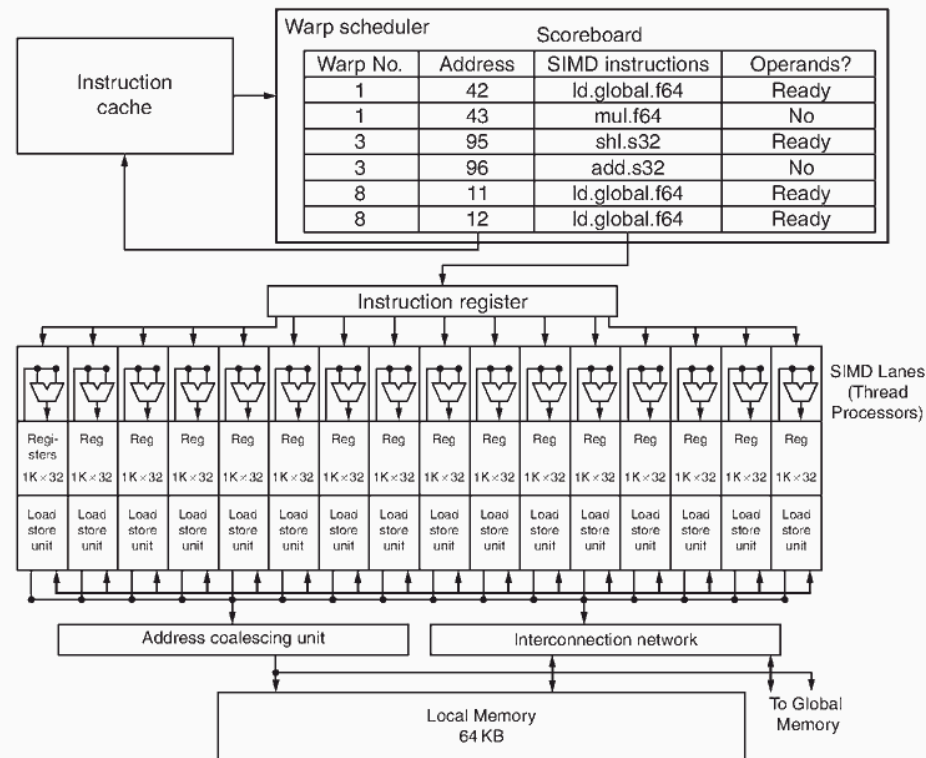
- Tuesday's lecture on 2/11 will be moved to room 4140 from 6.30 PM to 7.50 PM
- Office hours are on for today

Today's lecture

- Stencil methods on the GPU
 - ♦ 2D
 - ♦ 3D
- Limits to Performance

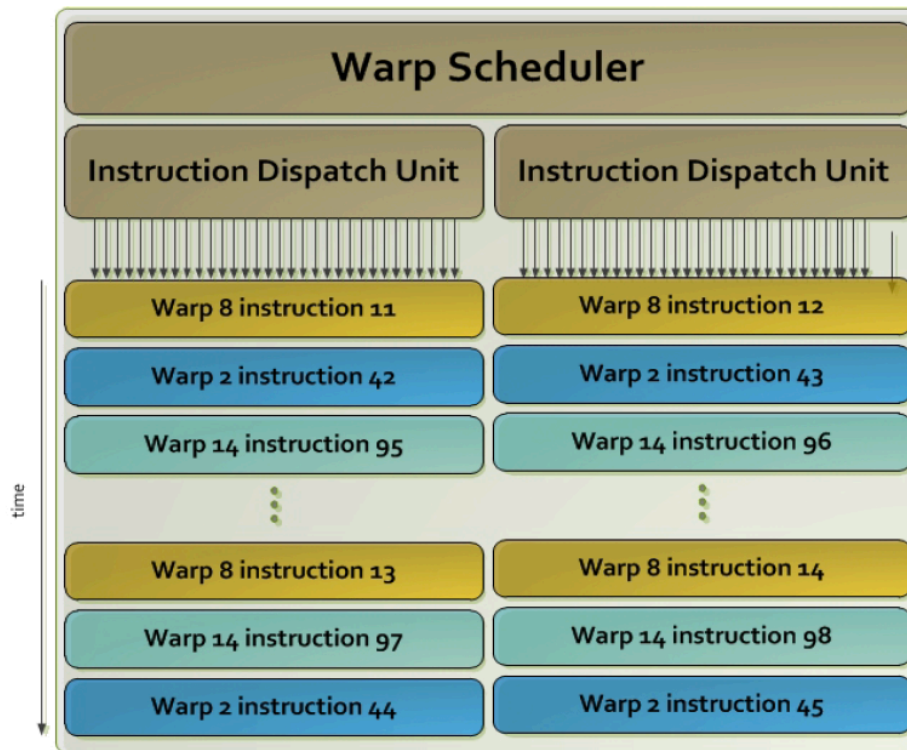
Hardware in perspective

- Each warp runs as an independent thread of SIMD instructions
- A warp takes two steps to execute, and acts as a 1024-bit SIMD instruction
- 512-bit wide vector SIMD instructions (16x32-bit SIMD lanes)



Thread scheduling

- Assigns independent instructions for processing
- From same or different warp
in same or different blocks



Mapping work onto processors

- A grid corresponds to a vectorizable loop
- From the software perspective a thread block ...

- ♦ is a single thread of vector instructions with a programmable vector length (the block size), allowing us to run on devices with different configurations
- ♦ strip mines the loop

- Consider Vector- matrix multiply

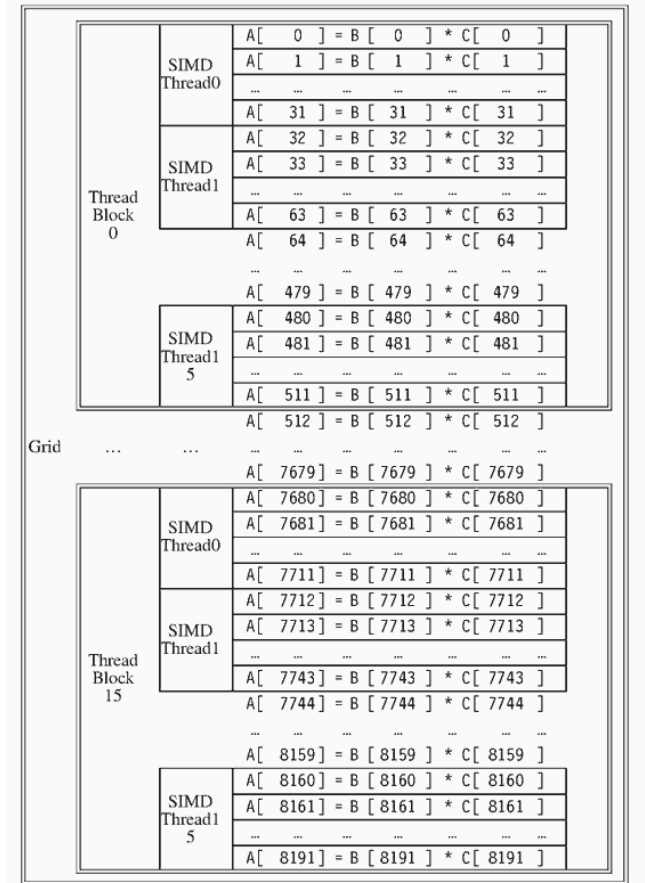
$$\underline{A} = \underline{B}\underline{C}$$

for $i = 0 : n - 1$

for $j = 0 : n - 1$

$$A[i] += B[i,j] * C[j]$$

Patterson and Hennessey, Morgan-Kaufmann pub



Strip mining

- Partitioning the iteration space into chunks

for $i = 0$ **to** $N-1$

$a[i] = b[i] + c[i];$

for $j = 0$ **to** $N-1$ **by** VL

for $i = j$ **to** $\min(N, j+VL) - 1$

$a[i] = b[i] + c[i];$

int $idx = blockIdx.x * blockDim.x + threadIdx.x;$

if ($idx < N$) $a[idx] = a[idx] + 1.f;$

Strip mining on the GPU

- Partitioning a thread block into warps corresponds to strip-mining into *independent* instruction streams
- Traditionally: independent instructions in the *same* instruction stream

```
int idx = blockIdx.x*blockDim.x + threadIdx.x;  
if (idx<N) a[idx] = a[idx]+1.f;
```

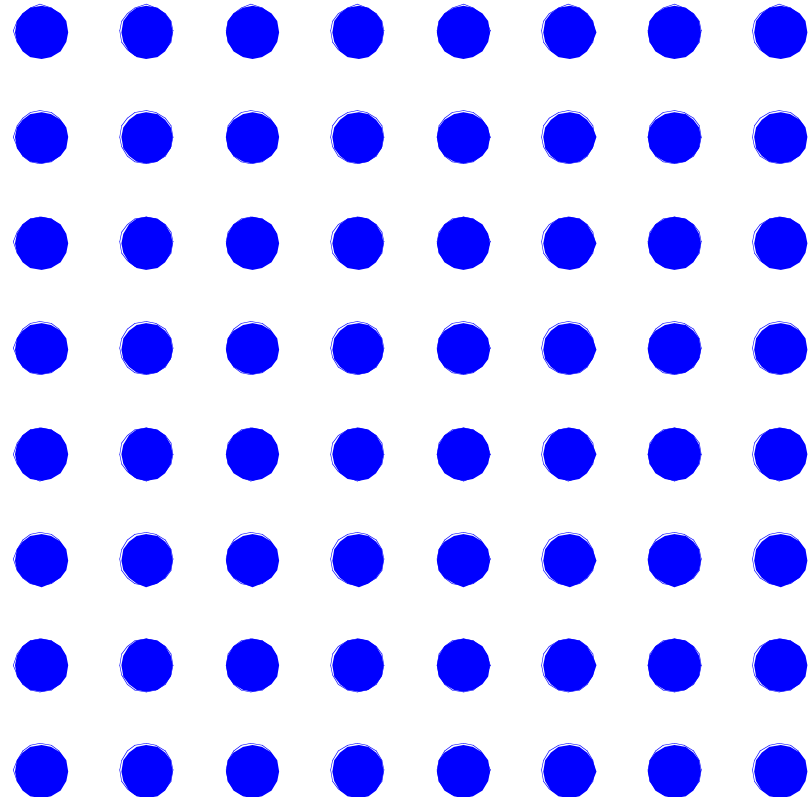
```
for j = 0 to N-1 by VL  
    for i = j to min(N, j+VL) - 1  
        a[i] = b[i] + c[i];
```

Today's lecture

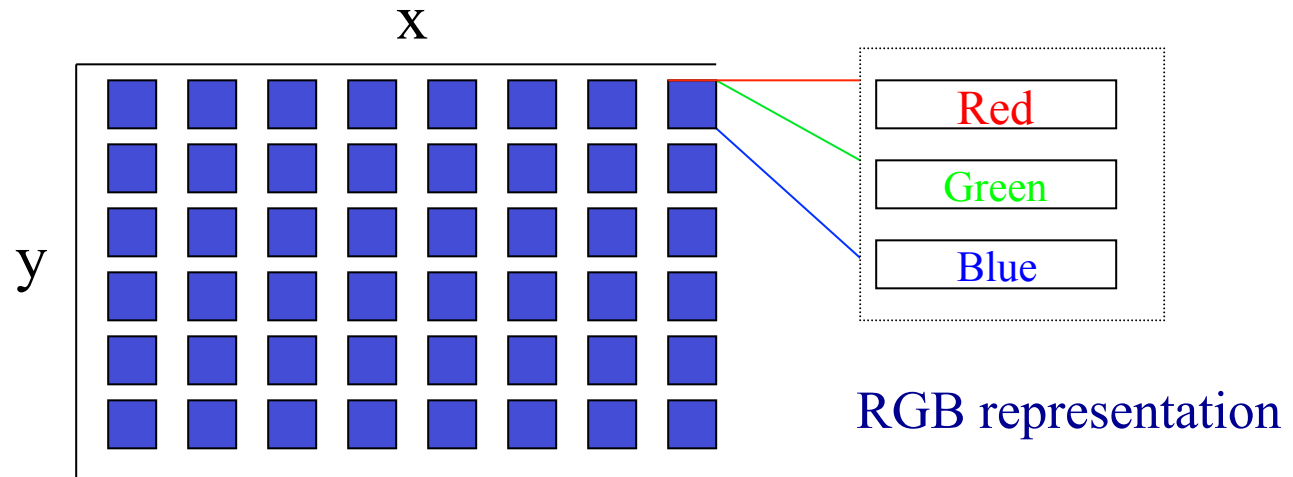
- Stencil methods on the GPU
 - ♦ 2D
 - ♦ 3D
- Limits to Performance

Stencil methods

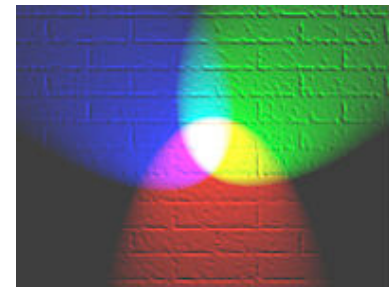
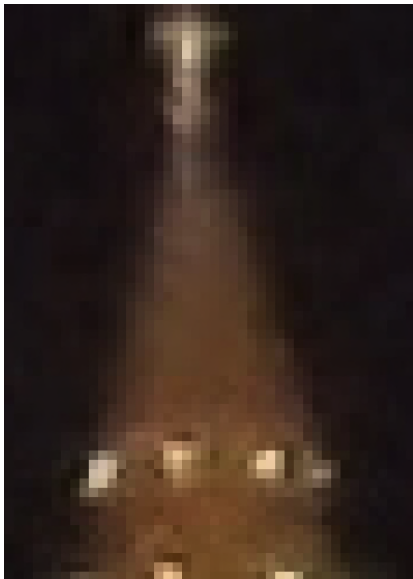
- Many physical problems are simulated on a uniform *mesh* in 1, 2 or 3 dimensions
- *Field variables* defined on a discrete set of points
- A *mapping* from ordered pairs to *physical observables* like temperature and pressure
- Important applications
 - ♦ Differential equations
 - ♦ Image processing



Digital Image Representation



Ryan Cuprak



wikipedia

Photos by Martin Juell

Image smoothing algorithm

- Repeat as many times as necessary

for (i,j) in 0:N-1 x 0:N-1

$$I^{\text{new}}[i,j] = (I[i-1,j] + I[i+1,j] + I[i,j-1] + I[i,j+1])/4$$

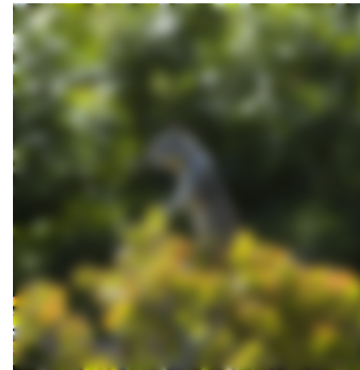
$I = I^{\text{new}}$



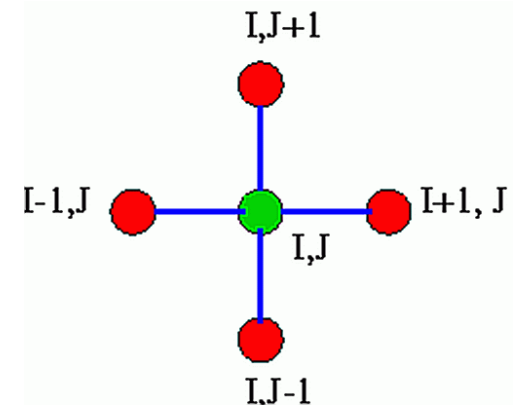
Original



100 iter

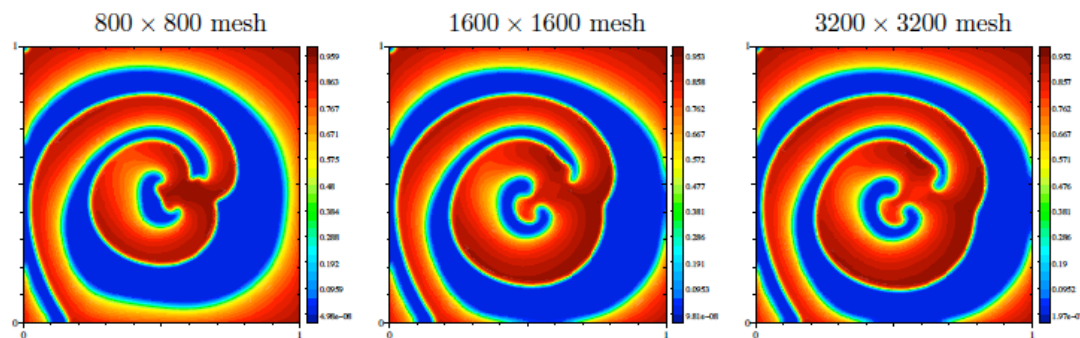


1000 iter



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

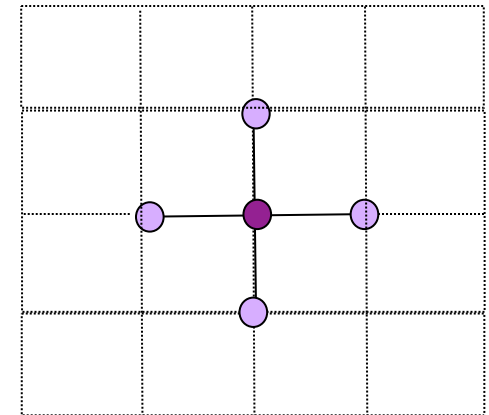
- Two parts
 - ♦ 2 Ordinary Differential Equations
 - Kinetics of reactions occurring at every point in space
 - ♦ Partial Differential Equation
 - Spatial diffusion of reactants
- First-order explicit numerical scheme

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

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

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
        EE[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));
    }
}
    
```

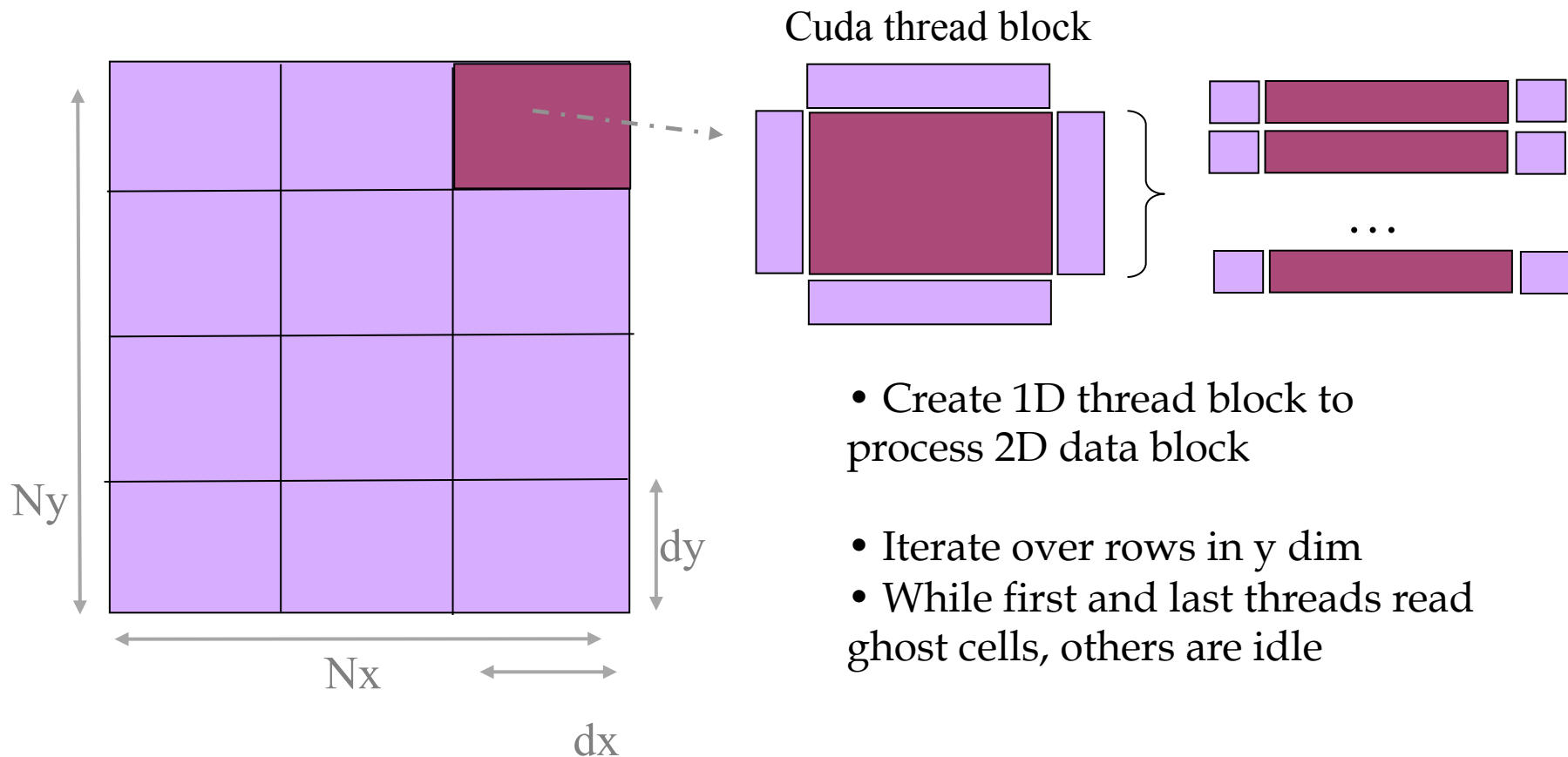
Naïve CUDA Implementation

- All array references go through device memory
- ./apf -n 6144 -t 0.04, 16x16 thread blocks
 - ♦ C1060 (1.3)
 - ♦ SP, DP: 22, 13GFlops

```
#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 (device cap. 1.3)

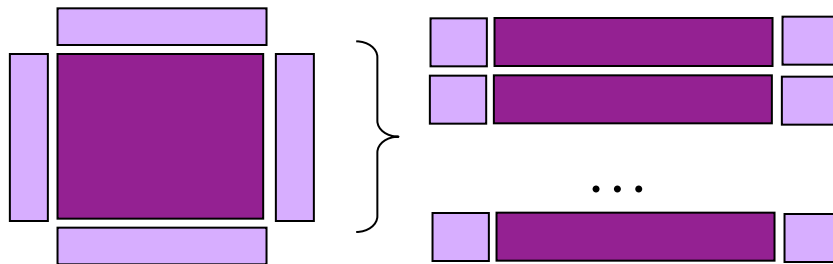


Compared to a 2D thread blocking, 1D thread blocks provide a 12% improvement in double precision and 64% improvement in single precision

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

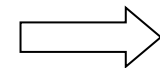
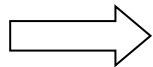
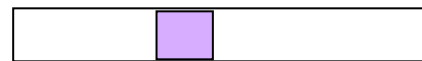
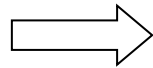
Top Row in Registers



Curr Row in Shared memory



Bottom Row in Registers



Read new row from
 global memory

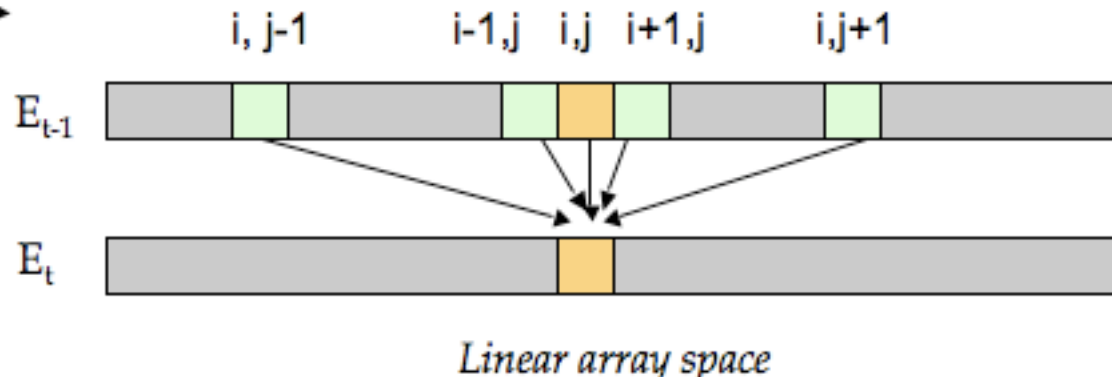
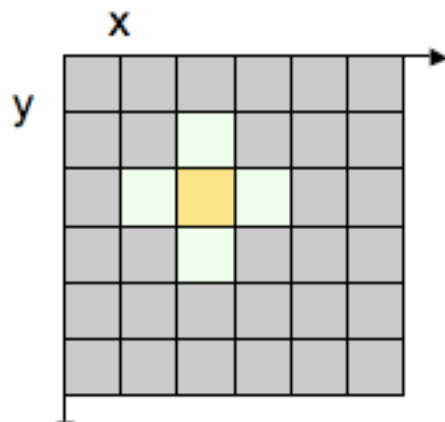
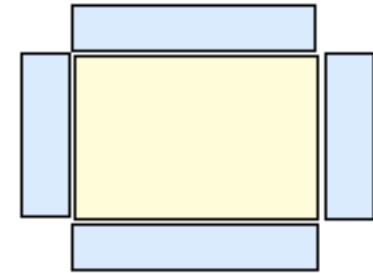
CUDA Code

```
__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

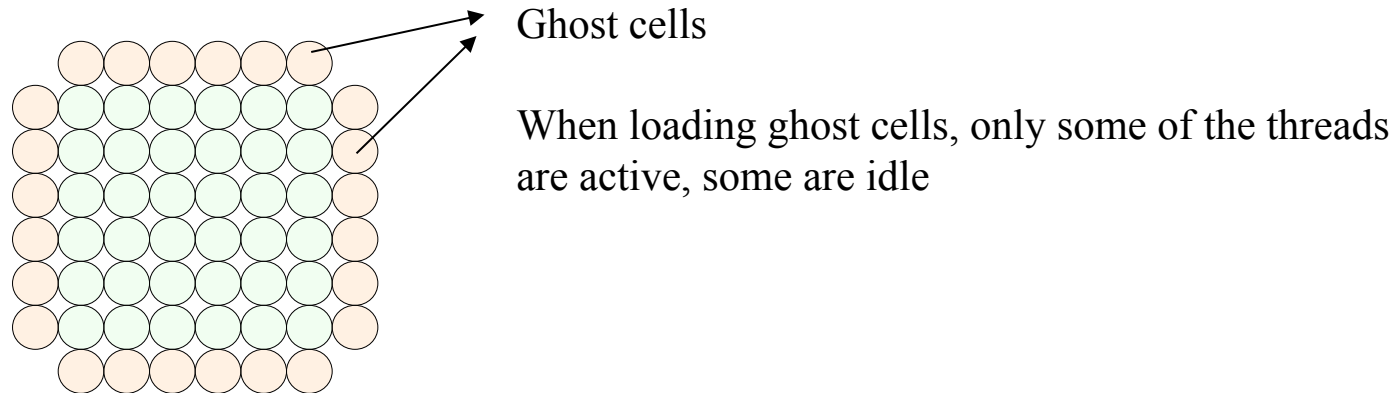
```
if (idy == 1 && y > 0 )
    block[0][idx] = E_prev[index - N];
else if(idy == DIM_Y && y < N-1)
    block[DIM_Y+1][idx] = E_prev[index + N];
if ( idx==1 && x > 0 )
    block[idy][0] = E_prev[index - 1];
else if( idx== DIM_X && x < N-1 )
    block[idy][DIM_X + 1] = E_prev[index + 1];
__syncthreads();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



Thread Mapping for Ghost Cells

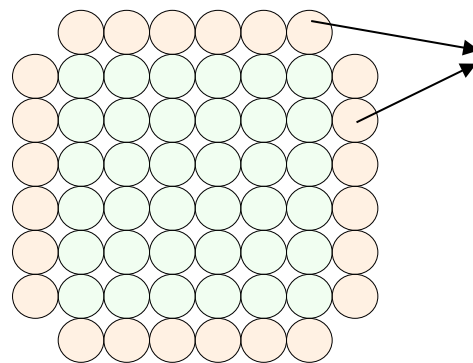
- Branches and thread divergence

```
if(threadIdx.y < 4 ){  
    //read a ghost cell into shared memory  
    block[borderIdx][borderIdx] = Uold[index];  
}
```



Ghost Cells (cont.)

- Divide the work between threads so each thread responsible for one ghost cell load
- For 16×16 tile size
 - ◆ There are $16 \times 4 = 64$ ghost cells
 - ◆ Create 64 threads
 - ◆ Each thread computes 4 elements in the Y-dim



Ghost cells

Each thread loads one ghost cell

Thread uses a hashmap to find its ghost cell assignment

The stencil computation and ODE

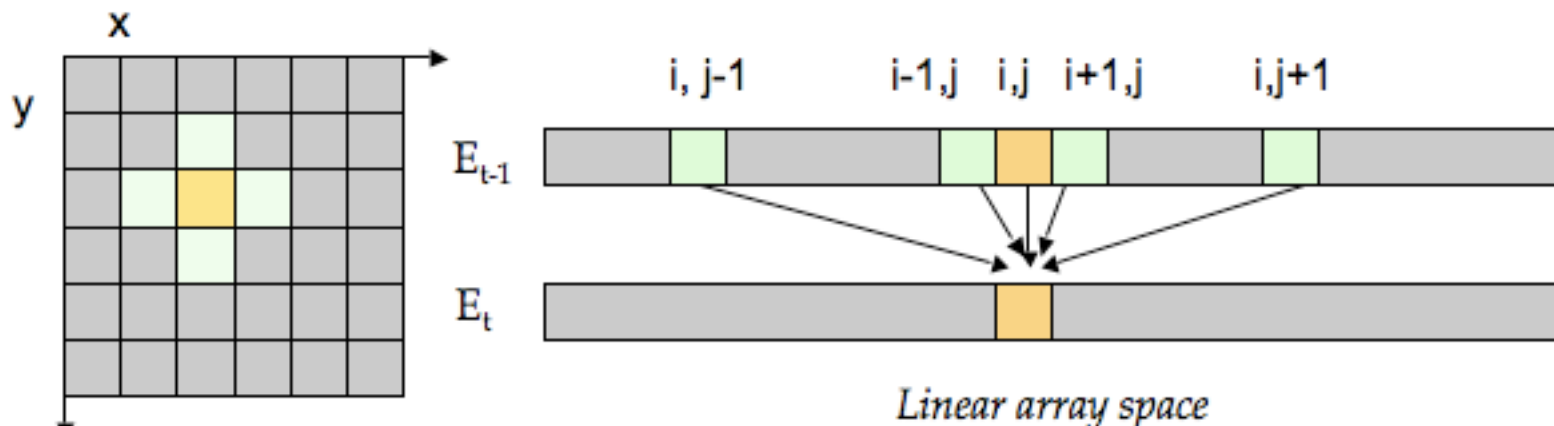
```
float r = R[index];
```

```
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

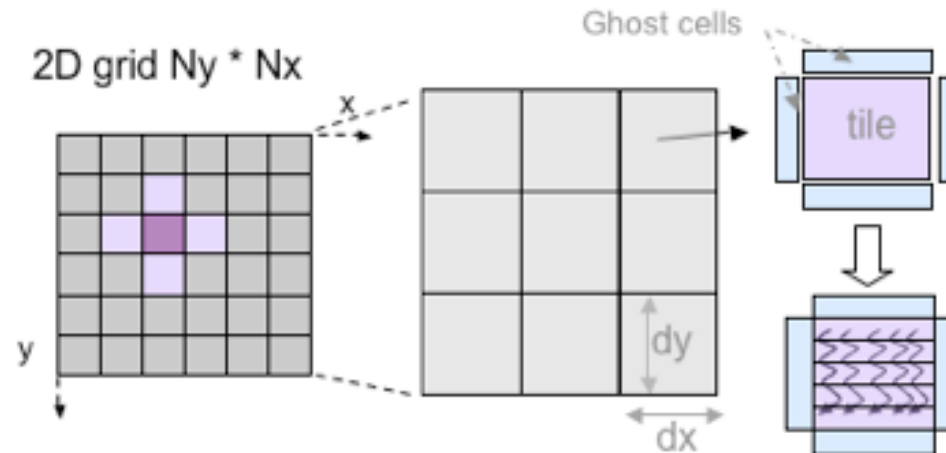
N=4K	1 CPU-only	1 GPU	Speedup over CPU
Single	2.36	124.48	52.84
Double	2.01	25.69	12.81
N=16K	4 CPUs-only	4 GPUs	Speedup over CPU
Single	7.79	454.47	58.32
Double	3.88	102.48	26.41

- Single Precision (22 Gflops w/o optimizations)
 - 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 (22 Gflops w/o optimizations)
 - 41.5% of the sustainable bandwidth
 - 1/3 of the peak double precision performance
 - Performance hurt by the division operation that appears in ODE

Today's lecture

- Stencil methods on the GPU
 - ◆ 2D
 - ◆ 3D
- Limits to Performance

Memory Accesses



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

Number of blocks = $N/d_x \times N/d_y$

Total Memory Accesses = $4N^2 + \frac{N^2}{d_x d_y} * 2(d_x + d_y)$

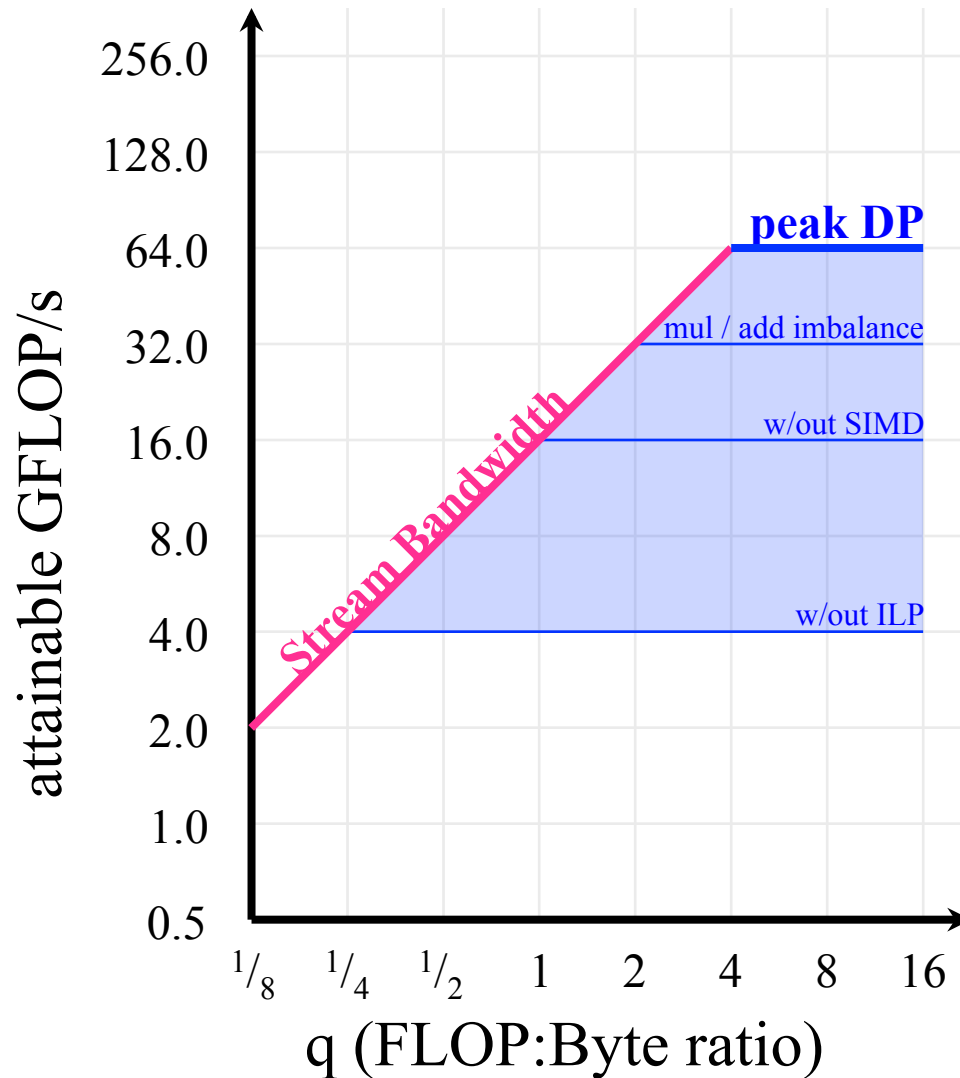
Estimated Kernel Time = Total Mem. Acces (bytes) / Empirical Device Bandwidth

Roofline model

- Establish performance bounds based on
 - ◆ Hardware capabilities
 - ◆ Arithmetic intensity (q): application requirements and impact of optimizations

$$\text{Attainable Performance}_{ij} = \min \left\{ \begin{array}{l} \text{FLOP/s with Optimizations}_{1-i} \\ q \times \text{Bandwidth with Optimizations}_{1-j} \end{array} \right.$$

Roofline model



- Peak performance as a function of q
- Top line corresponds to theoretical limit based on clock speed
- Lower rooflines correspond to successively degraded performance as we remove optimizations

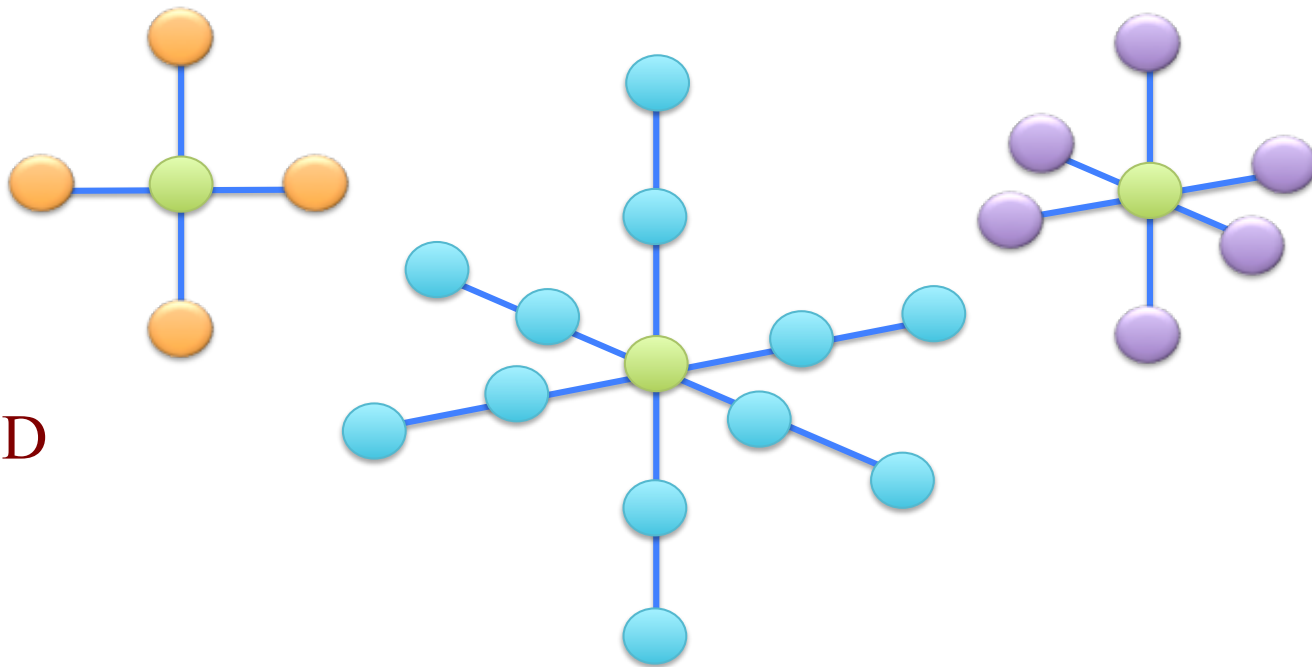
Today's lecture

- Stencil methods on the GPU
 - ♦ 2D
 - ♦ 3D

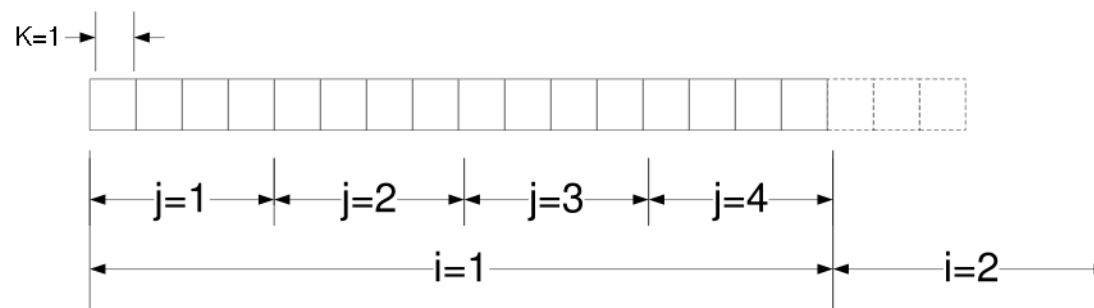
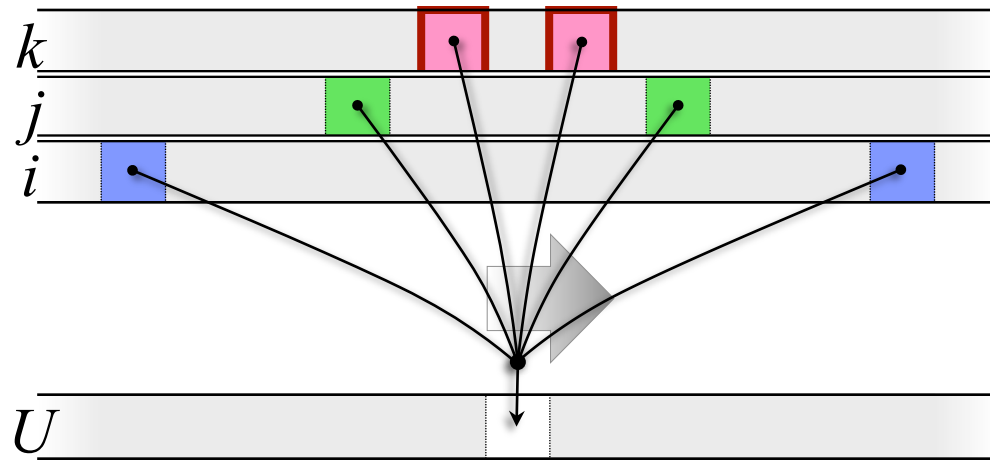
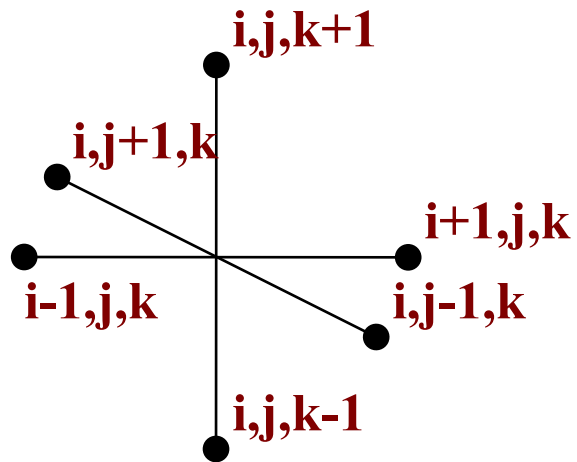
3D Stencils

- More demanding
 - ♦ Large strides
 - ♦ Curse of dimensionality

2D



Memory Strides

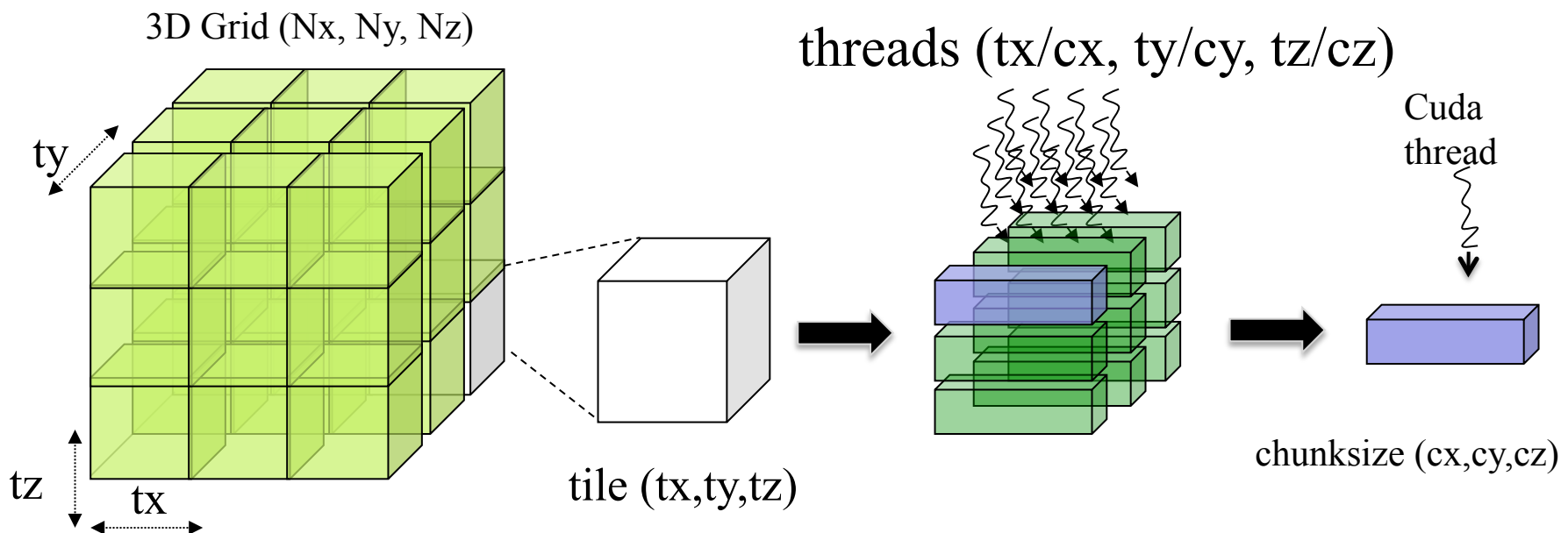


Sam Williams et al.

H. Das, S. Pan, L. Chen

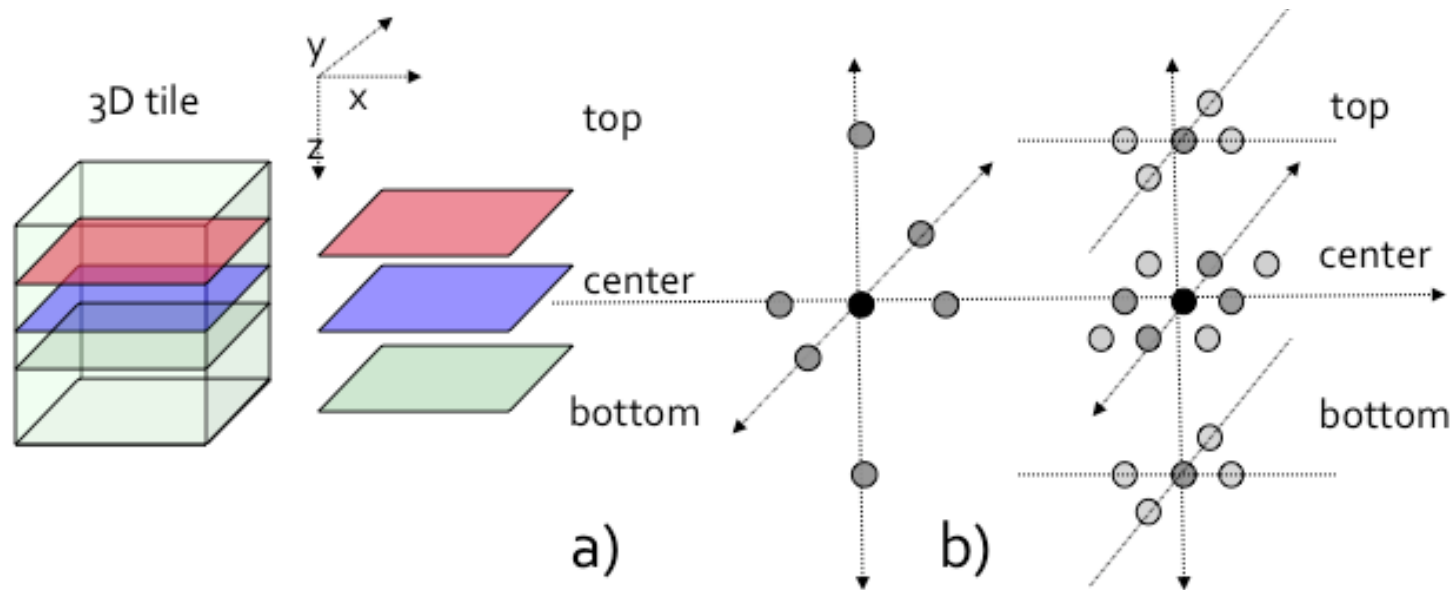
CUDA Thread Blocks

- Split mesh into 3D tiles
- Divide elements in a tile over a thread block



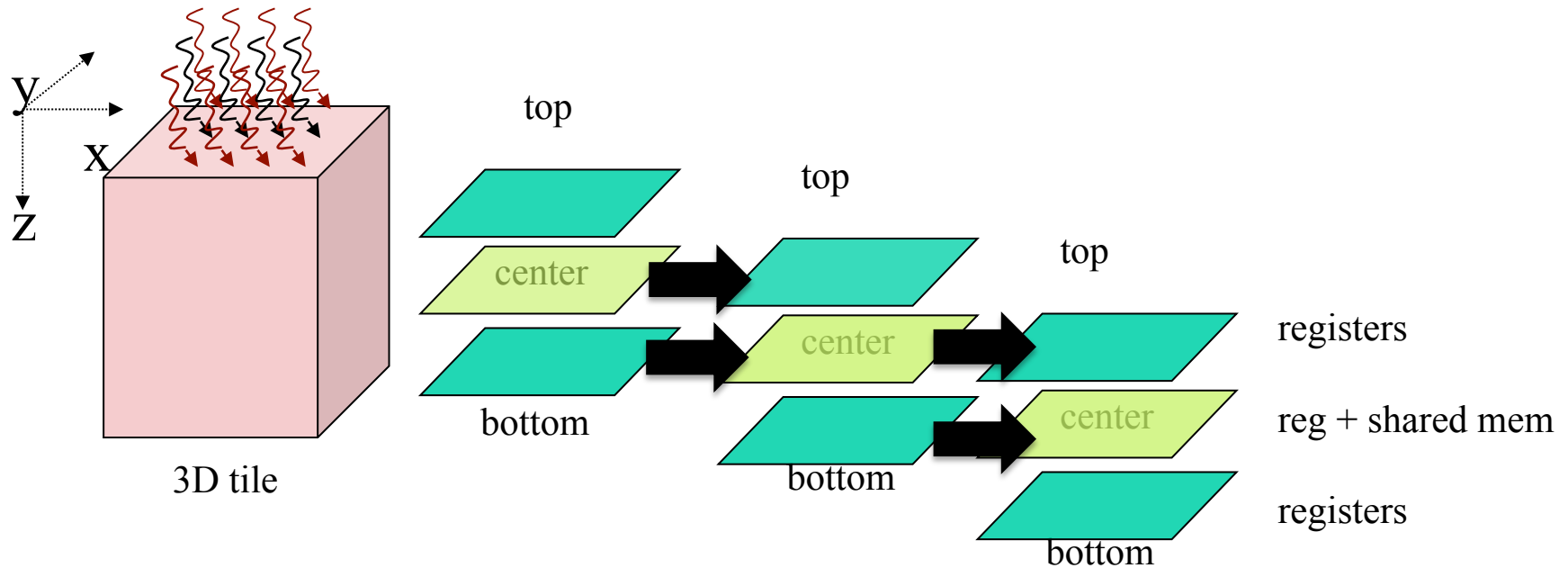
On chip memory optimization

- Copy center plane into shared memory
- Store others in registers
- Move in and out of registers



Rotating planes

- Copy center plane into shared memory
- Store others in registers
- Move in and out of registers



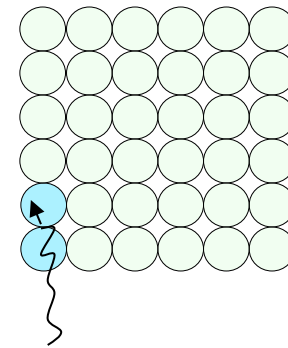
Performance Summary

- $N^3=256^3$, double precision

GFLOPS	Tesla 1060
Naïve	8.9
Shared Memory	15.5
Sliding Planes	20.7
Registers	23.6

Multiple Elements in Y-dim

- If we let a thread compute more than one plane, we can assign more than one row in the slowest varying dimension
- Reduces index calculations
 - ◆ But requires more registers
- May be advantageous in handling ghost cells

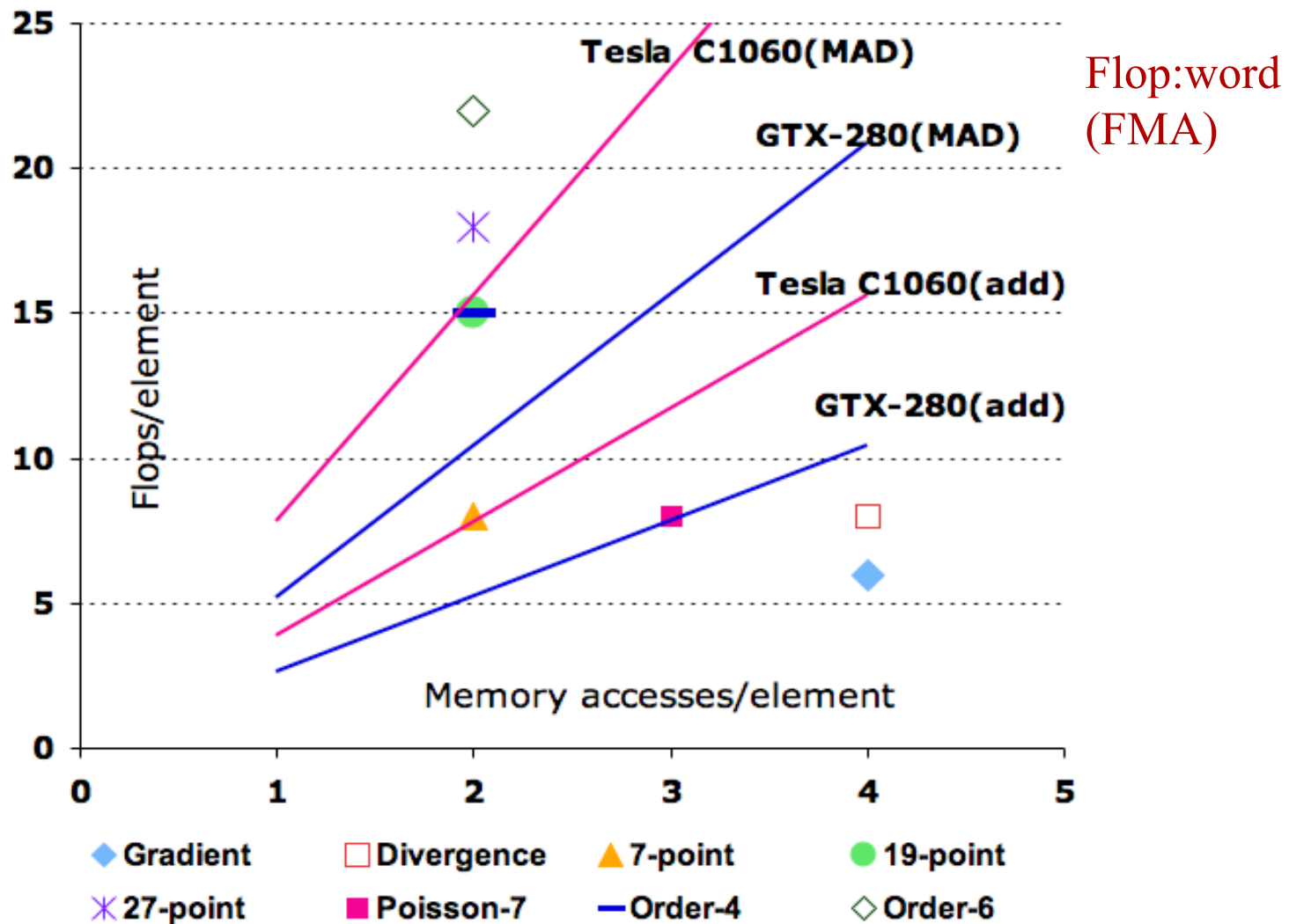


Contributions to Performance

- $N^3=256^3$, double precision

GFLOPS	Tesla 1060
Naïve	8.9
Shared Memory	15.5
Sliding Planes	20.7
Registers	23.6
MultipleY2	26.3
MultipleY4	26.2

Influence of memory traffic on performance



Generational changes in implementation strategy

- On Fermi, we do not see a large change in performance when we use shared memory! Cache helps!
 - ♦ C1060 (1.3) , cseclass01, cseclass05
 - ♦ SP: 22, 73, 34 Gflops
DP: 13, 45, 20 Gflops
- But on the next generation Kepler we do!
- Global memory references aren't cached as in Fermi
- Caching used mostly to handle register spills