

CSE 262

Lecture 10

Multigrid
GPU Implementation of stencil methods (I)

Announcements

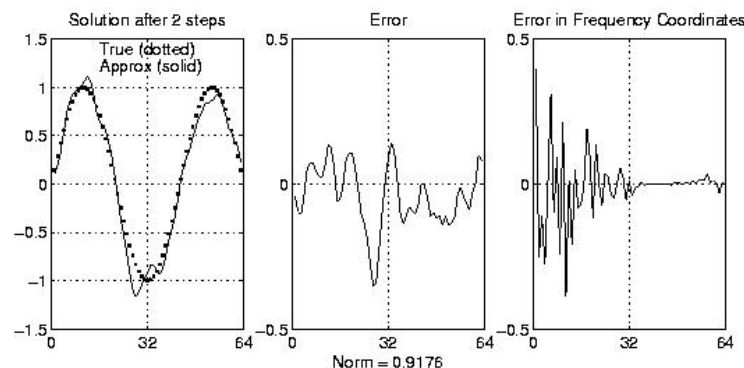
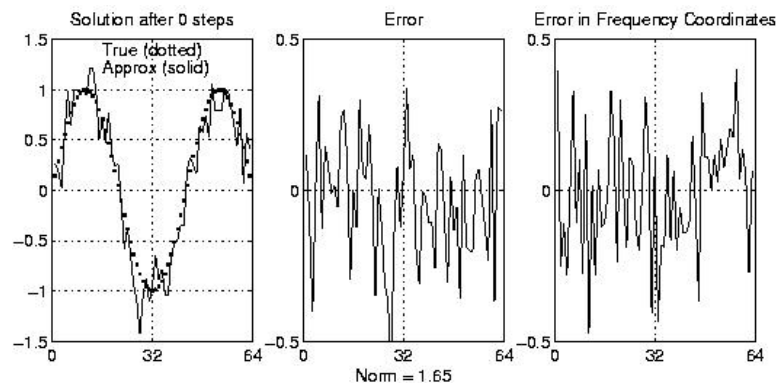
- Final presentations
 - ◆ Friday March 13th, 10:30 AM to 1:00PM
 - ◆ Room 3217, CSE Building (EBU3B)

Today's lecture

- Multigrid
- GPU Implementation of Stencil methods

Multigrid

- Recall that the image smoother converges slowly: $O(n^2)$ iterations for a mesh with n unknowns, $n=m^2$, m^3
- The smoother damps the high frequency error components much faster than low frequency ones



The idea behind multigrid

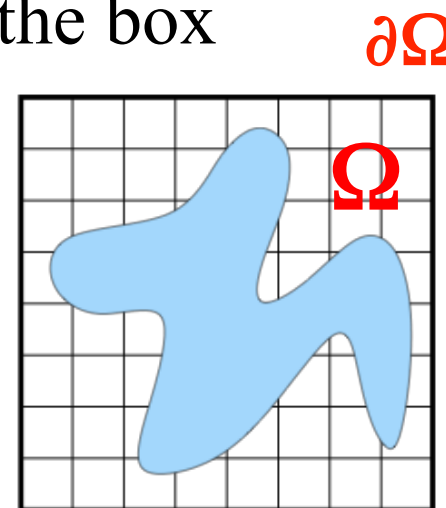
- If we can make low frequencies appear to be high frequencies, we can speed convergence
- Coarsening the mesh uses half as many points, doubling the frequency
- Cancel out the fine mesh errors using coarse mesh information
- Multigrid provides the glue between levels
- Another angle: numerical information communicated at multiple length scales
- We can improve the communication rate via multigrid

Multigrid

- Maintain a hierarchy of grids, each coarser than the one below it
- Find an approximate solution on the coarser grid, and do this recursively
- Use each coarse grid approximation as an initial guess for the finer grid

Some preliminaries

- We'll solve the discrete Poisson Equation in 2D
- $\Delta \mathbf{u} = \mathbf{f}(\mathbf{x}, \mathbf{y})$ within a square box, $\mathbf{x}, \mathbf{y} \in [0, 1]$
 $\mathbf{u}(\mathbf{x}, \mathbf{y}) = \mathbf{f}(\mathbf{x})$ on $\partial\Omega$, the boundary of the box
($\Delta \mathbf{u} = \partial^2 \mathbf{u} / \partial \mathbf{x}^2 + \partial^2 \mathbf{u} / \partial \mathbf{y}^2$)
- Smoother:
Red/Black Gauss-Seidel

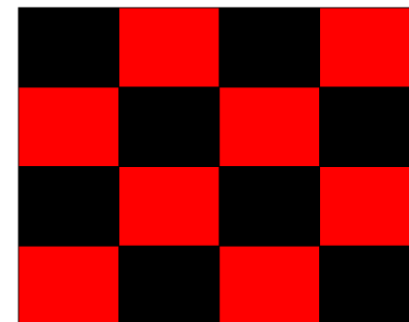


for (i,j) in 0:N-1 x 0:N-1 on the red points

$$u[i,j] = (u[i-1,j] + u[i+1,j] + u[i,j-1] + u[i,j+1]) / 4$$

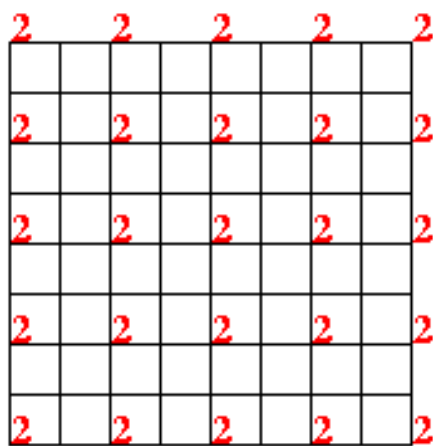
for (i,j) in 0:N-1 x 0:N-1 on the black points

$$u[i,j] = (u[i-1,j] + u[i+1,j] + u[i,j-1] + u[i,j+1]) / 4$$

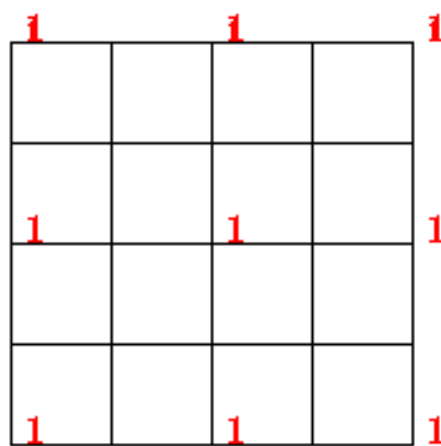


The algorithm (following Demmel)

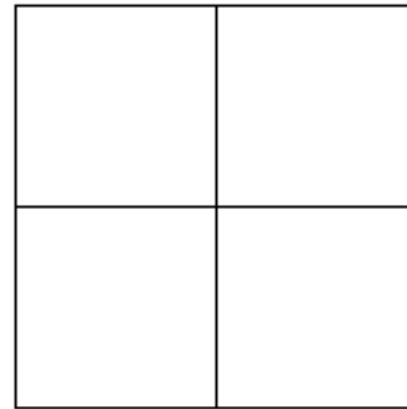
- Consider a $2m+1$ grid in 1D ($2m+1 \times 2m+1$ grid in 2D)
- Let $U(i)$ be the problem of solving the discrete Poisson equation on a 2^i+1 grid in 1D ($2^i+1 \times 2^i+1$ grid in 2D)
- $U^{(m)}, U^{(m-1)}, \dots, U^{(1)}$: sequence of problems from fine to coarse
- **Red points** are part of the next coarser grid



U^3



U^2



U^1

Accuracy

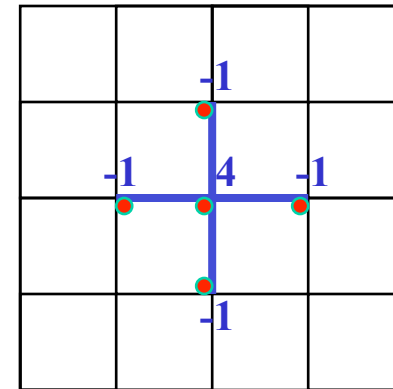
- We require that the coarse grid solution be a reasonable approximation to the fine grid solution
- Each level suppresses the error within the upper half the frequency spectrum
- The width of the frequency spectrum shrinks by one half at each coarser level

Some preliminaries (II)

- We rewrite Poisson's equation, with $A = \Delta$
 $Au = f$

$$A = \begin{pmatrix} 4 & -1 & & & & \\ -1 & 4 & -1 & & & \\ & -1 & 4 & & & \\ -1 & & & 4 & -1 & -1 \\ & -1 & -1 & 4 & -1 & \\ & & -1 & -1 & 4 & -1 \\ & & & -1 & & 4 & -1 \\ & & & & -1 & -1 & 4 & -1 \\ & & & & & -1 & -1 & 4 \end{pmatrix}$$

Graph and “stencil”



The error equations

- Poisson's equation, with $A = \Delta$
 $Au = f$
- In the discrete equation we have $A^h u^h = f^h$, where A^h is the discrete analog of A
- What if we knew the exact error e^h in the computed solution?
 - By definition $e^h = u^h - u$
 - We just form $u^h - e^h$ which gives us the exact solution
- Define the *residual* $r = Au - f^h$, and its discrete form
 $r^h = A^h u^h - f^h$

The error equations

- By the definition of e^h we now have

$$A^h(e^h + u^h) = f^h = A^h e^h + A^h u^h$$

- But $A^h u^h - f^h = r^h$: we arrive at the **residual equation**
 $A^h e^h = -r^h$

- We can obtain an approximation to e^h on a coarse grid by solving

$$A^{2h} e^{2h} = -r^{2h}$$

And adding the result to our computed solution u^h

- This is where multigrid comes in: we recursively solve e^{2h} on a coarser grid and then interpolate to u^h

Multigrid Operators

- For problem $U^{(i)}$

$f(i)$ is the RHS and $u(i)$ is the current estimated solution
both live on grids of size 2^i-1

$A(i)$ is implicit in the operators below

- All the following operators just average values on neighboring grid points

Neighboring grid points on coarse problems are far away in fine problems, so information moves quickly on coarse problems

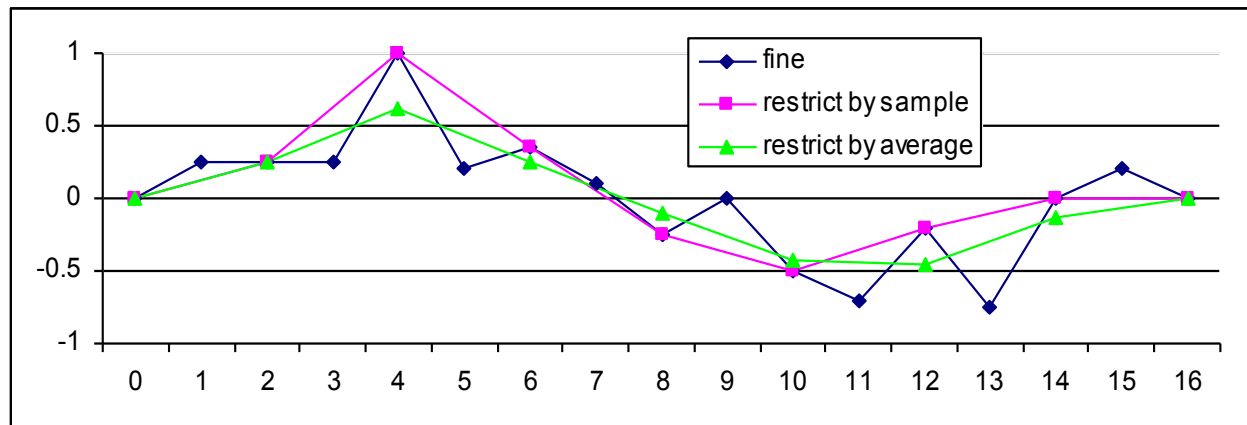
Multigrid Operators

- The **solution operator $S(i)$** takes $U^{(i)}$ and computes an improved solution $u_{\text{improved}}^{(i)}$ on same grid
Uses red-black Gauss Seidel
 $u_{\text{improved}}^{(i)} = S(i) (f(i), u(i))$
- The **restriction operator $R(i)$** maps $U^{(i)}$ to $U^{(i-1)}$
Restricts problem on fine grid $U^{(i)}$ to coarse grid $U^{(i-1)}$ by sampling or averaging **both live on grids of size 2^{i-1}**
 $f^{(i-1)} = R(i) (f(i))$
- The **prolongation (interpolation) operator $P(i-1)$** maps an approximate solution $U^{(i-1)}$ to $U^{(i)}$
Interpolates solution on coarse grid $U^{(i-1)}$ to fine grid $U^{(i)}$
 $U^{(i)} = P(i-1) U^{(i-1)}$

The Restriction Operator $R(i)$

- The restriction operator, $R(i)$, takes
 - ♦ a problem $U^{(i)}$ with RHS $f^{(i)}$ and
 - ♦ maps it to a coarser problem $U^{(i-1)}$ with RHS $f^{(i-1)}$
 - ♦ Averaging or sampling
- Average values of neighbors

$$u_{\text{coarse}}(i) = \frac{1}{4}u_{\text{fine}}(i-1) + \frac{1}{2}u_{\text{fine}}(i) + \frac{1}{4}u_{\text{fine}}(i+1)$$



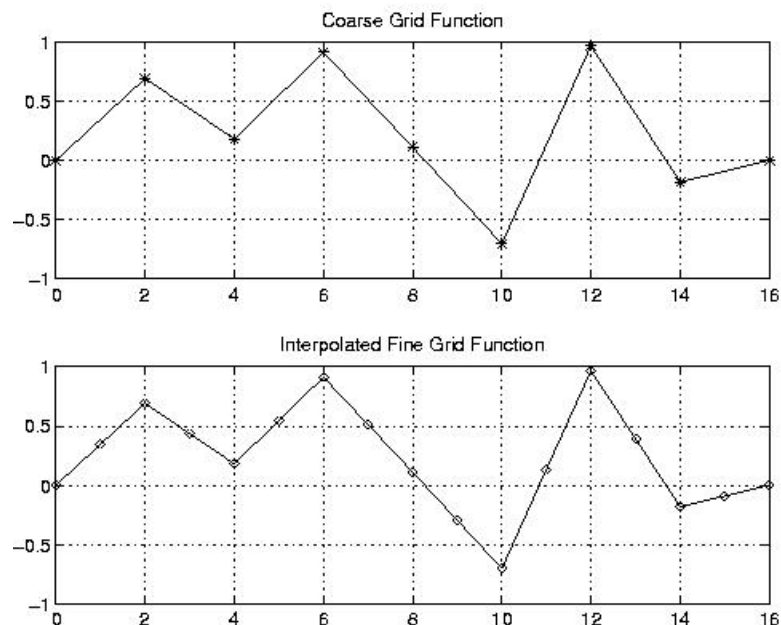
Courtesy Jim Demmel and
Katherine Yelick

Prolongation Operator $P(i)$

- The prolongation operator $P(i-1)$ converts a coarse grid solution $U^{(i-1)}$ to a fine grid $U^{(i)}$
- In 1D: linearly interpolate nearest coarse neighbors

$u_{\text{fine}}(i) = u_{\text{coarse}}(i)$ if the fine grid point i is also a coarse one,
else

$$u_{\text{fine}}(i) = \frac{1}{2}(u_{\text{coarse}}(\text{left of } i) + u_{\text{coarse}}(\text{right of } i))$$



Courtesy Jim Demmel

Multigrid V-Cycle Algorithm

Function MG_V ($f(i)$, $u(i)$)

... Solve $A(i) * u(i) = f(i)$ given $f(i)$ and an initial guess for $u(i)$

... return an improved $u(i)$

if ($i = 1$)

compute exact solution $v(1)$ of $U^{(1)}$ only 1 unknown

return $u(1)$

else

$u(i) = S(i) (f(i), u(i))$

improve solution by

damping high frequency error

$r(i) = A(i) * u(i) - f(i)$

compute residual

$d(i) = U(i-1) (MG_V (R(i) (r(i)), 0))$

solve $A(i) * d(i) = r(i)$ recursively

$u(i) = u(i) - d(i)$

correct fine grid solution

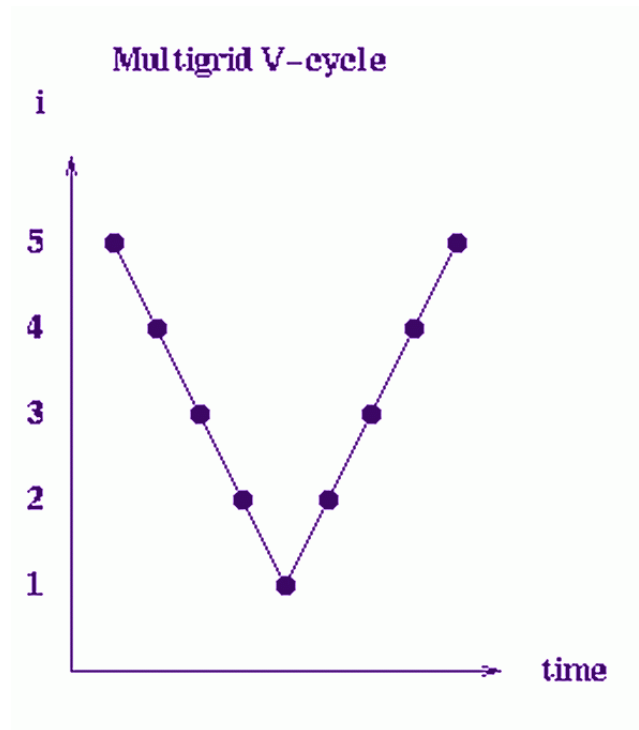
$u(i) = S(i) (f(i), u(i))$

improve solution again

return $u(i)$

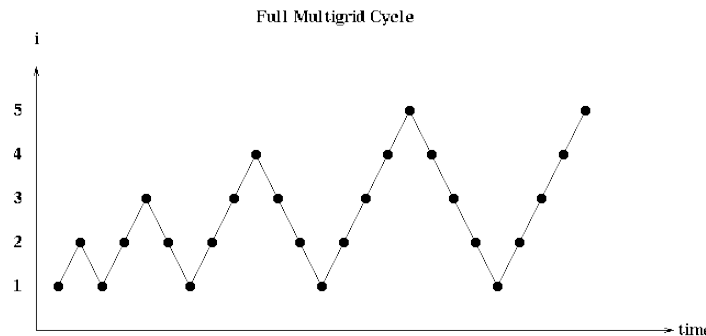
The V-Cycle

- The call graph by level has the shape of the letter 'V'



Complexity of a V-Cycle

- Work at each point in a V-cycle is $O(\text{the number of unknowns})$
- Cost of Level i is $(2^i - 1)^2 = O(4^i)$
- If finest grid level is m , total time is:
$$\sum_{i=1}^m O(4^i) = O(4^m) = O(\# \text{ unknowns})$$
- There is also Full Multigrid, see the reader for details

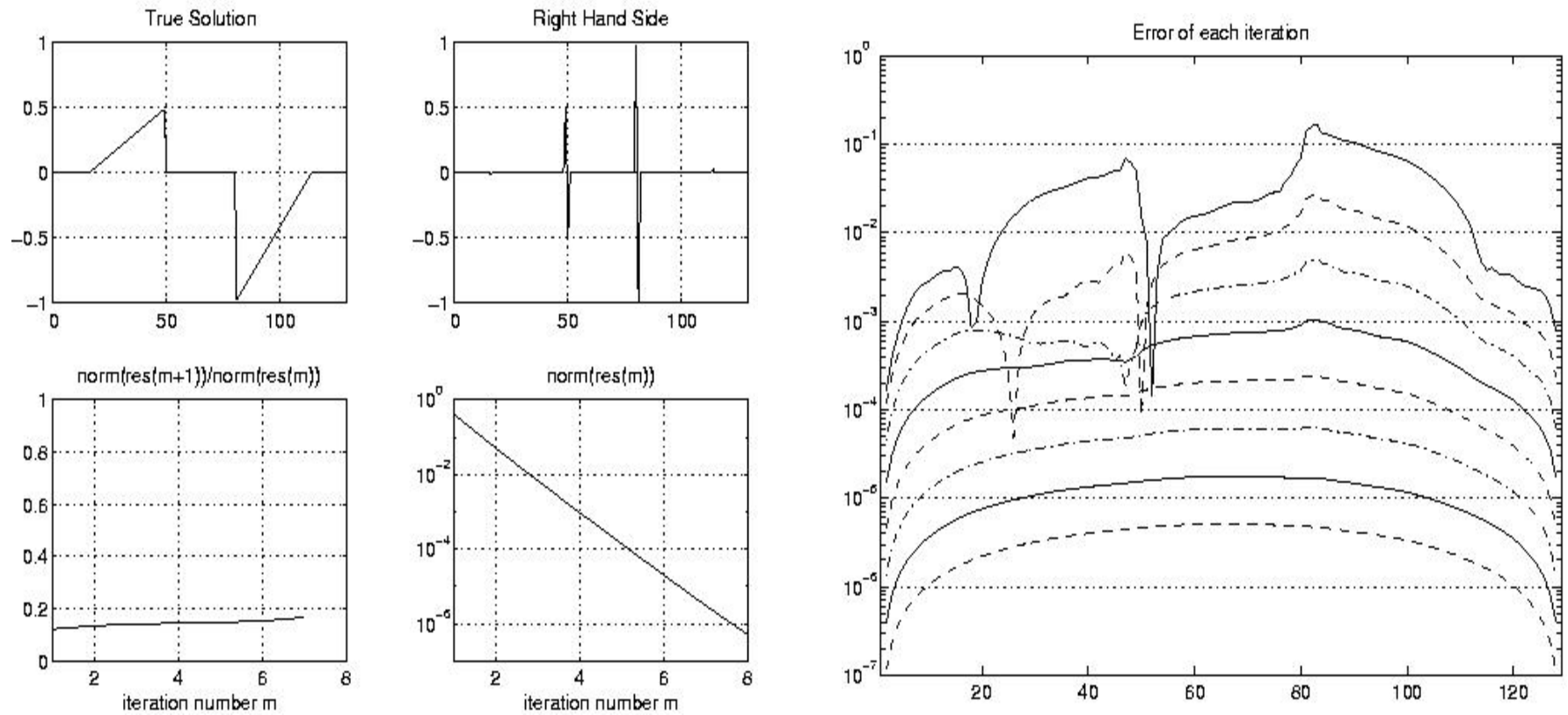


Courtesy Jim Demmel

Constant convergence rate in Multigrid

- Theorem: each iteration of *full* multigrid reduces the error by at least a factor of two, *independent of the number of unknowns*
- We can make the error smaller than any given tolerance in a **fixed** number of steps, independent of size of the grid
- This distinguishes MG from other iterative methods, which converge more slowly for large grids

Convergence of Multigrid in 1D



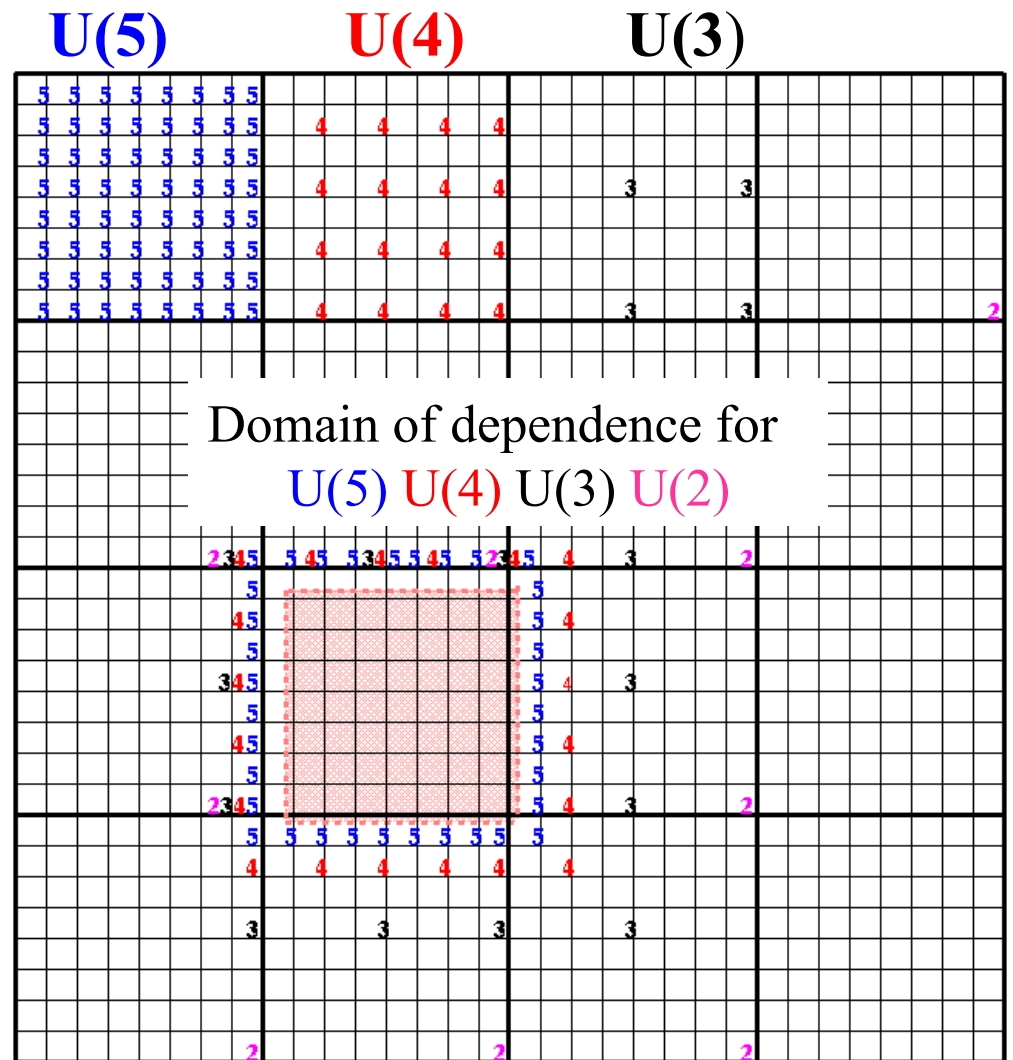
Courtesy Jim Demmel

Parallel 2D Multigrid

- Multigrid on 2D requires nearest neighbor (up to 8) computation at each level of the grid
- Start with $n=2^{m+1}$ by 2^m+1 grid (here $m=5$)

33 × 33 mesh

4 × 4 processors



Courtesy Jim Demmel

Performance of parallel 2D Multigrid

- Assume 2^{m+1} by 2^{m+1} grid of unknowns, $n = 2^{m+1}$, $N = n^2$
- Let $p = 4^k$ processors, arranged in 2^k by 2^k grid
 - ♦ Each processor has a 2^{m-k} by 2^{m-k} subgrid
- V-cycle starting at level m
 - ♦ At levels m through k , each processor does some work
 - ♦ At levels $k-1$ through 1 , some processors are idle, because a 2^{k-1} by 2^{k-1} grid of unknowns is insufficient to keep all processors busy
- Sum over all levels in all V-cycles in FMG to get complexity

Performance of parallel 2D Multigrid

- Cost of one level in V-cycle

- ♦ If level $j \geq k$, then cost =

- $O(4^{j-k})$ Flops, proportional to number of mesh points/processor

- $+ O(1) \alpha$ Send a constant # messages to neighbors

- $+ O(2^{j-k}) \beta$ Number of words sent

- ♦ If level $j < k$, then cost =

- $O(1)$ Flops, proportional to number of mesh points/processor

- $+ O(1) \alpha$ Send a constant # messages to neighbors

- $+ O(1) \beta$ Number of words sent

The curse of dimensionality

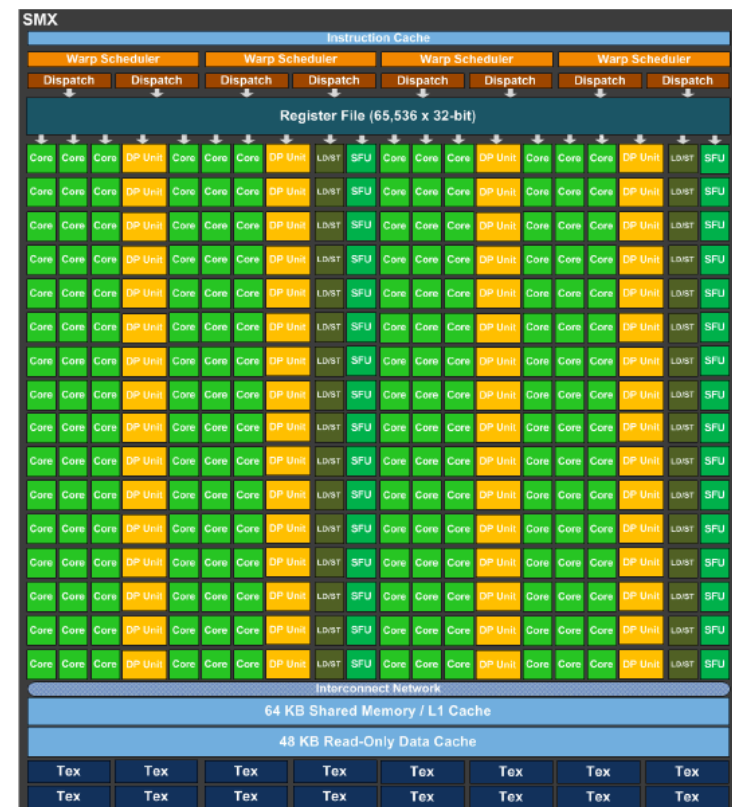
- In practice, we don't coarsen to a 1×1 mesh
- In sequential code, the coarsest grids are negligibly cheap, but on a parallel machine they are not.
 - ♦ Consider 1000 points per processor
 - ♦ In 2D, the surface to communicate is $4\sqrt{1000} \approx 128$, or 13%
 - ♦ In 3D, the surface is $6(1000)^{2/3} \approx 600$, or 60%
- We reduce the amount of parallelism as we coarsen the mesh
- Replicate computation: reduces communication

Today's lecture

- Multigrid
- Hardware in perspective
- GPU Implementation of Stencil methods

Hardware in perspective

- Each SMX contains 12 groups of 16 cores
- An SMX executes a warp in 2 cycles on a group of 16 cores (integer and single precision)
- At most 4 of the groups can be executing double precision
- Each warp runs as an independent thread of SIMD instructions
- A warp acts as a 32 element double precision SIMD instruction
- There can be 4 active simultaneously



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

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;
```

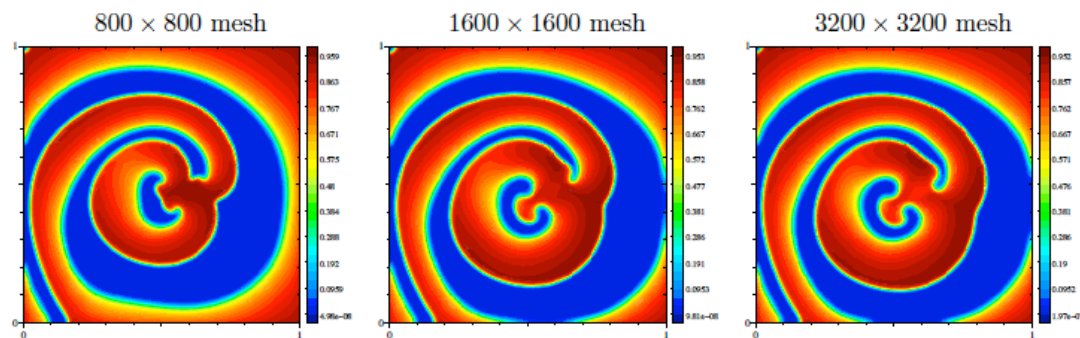
```
for j = 0 to N-1 by VL  
    for i = j to min(N, j+VL) - 1  
        a[i] = b[i] + c[i];
```

Today's lecture

- Multigrid
- Hardware in perspective
- GPU Implementation of Stencil methods

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

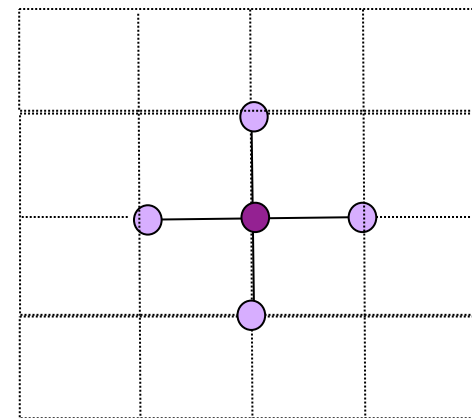
- Two parts
 - ♦ 2 Ordinary Differential Equations
 - Kinetics of reactions occurring at every point in space
 - ♦ Partial Differential Equation
 - Spatial diffusion of reactants
- First-order explicit numerical scheme

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

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

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));
  }
}

```

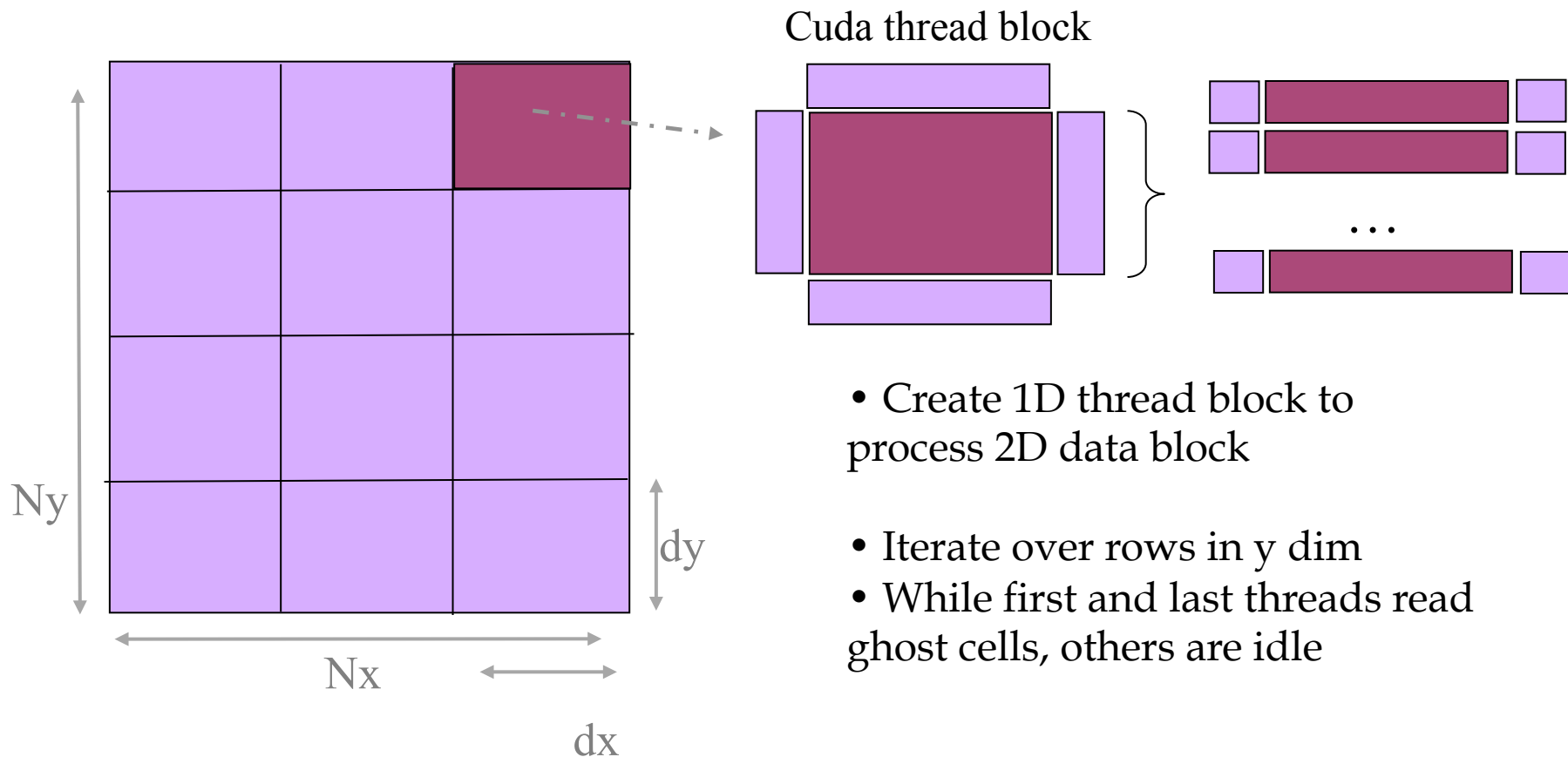
Naïve CUDA Implementation

- 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)

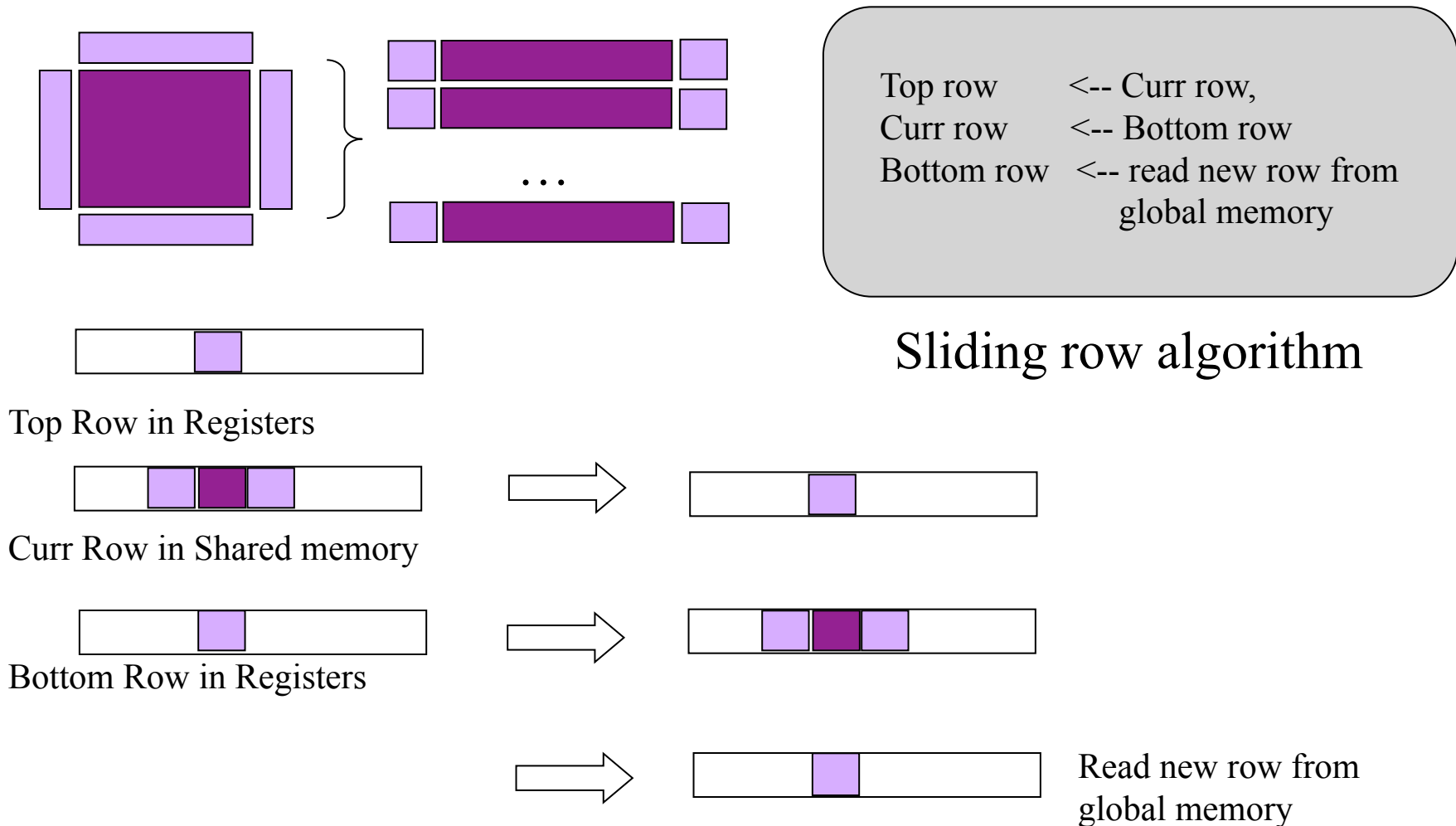


Compared to a 2D thread blocking, 1D thread blocks provide a 12% improvement in double precision and 64% improvement in single precision

Didem Unat

Sliding rows

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



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();
```

