Solving PDEs with CUDA

Jonathan Cohen

jocohen@nvidia.com

NVIDIA Research

PDEs (Partial Differential Equations)

- Big topic
- Some common strategies
- Focus on one type of PDE in this talk

- Poisson Equation
  - Linear equation => Linear solvers
  - Parallel approaches for solving resulting linear systems

Poisson Equation

Classic Poisson Equation:

$$^2p = \text{rhs} \quad \text{(p, rhs scalar fields)}$$

(Laplacian of *p* = sum of second derivatives)

$$\partial^2 p/\partial x^2 + \partial^2 p/\partial y^2 + \partial^2 p/\partial z^2 = \text{rhs}$$

$$\partial^2 p/\partial x^2 \approx \frac{\partial(p+\Delta \mathrm{x})/\partial x - \partial p/\partial x}{\Delta \mathrm{x}}$$

To compute Laplacian at P[4]:

$1/\Delta x^2$	P[0]	P[1]	P[2]	P[3]	P[4]	P[5]	P[6]	P[7]
				1	-2	1		

1st Derivatives on both sides:

$$\partial p/\partial x = \frac{P[4] - P[3]}{\Delta x} \qquad \partial (p+\Delta)/\partial x = \frac{P[5] - P[4]}{\Delta x}$$

Derivative of 1st Derivatives:

$$\frac{\frac{P[5] - P[4]}{\Delta x} - \frac{P[4] - P[3]}{\Delta x}}{\Delta x} \Rightarrow 1/\Delta x^2 \left(P[3] - 2P[4] + P[5] \right)$$

Poisson Matrix

Poisson Equation is a sparse linear system

-2	1						1		P[0]
1	-2	1							P[1]
	1	-2	1						P[2]
		1	-2	1					P[3]
			1	-2	1				P[4]
				1	-2	1			P[5]
					1	-2	1		P[6]
1						1	-2		P[7]

$= \Delta x^2 \text{ RHS}$

Approach 1: Iterative Solver

Solve M p = r, where M and r are known

Error is easy to estimate: E = M p’ - r

Basic iterative scheme:

- Start with a guess for p, call it p’
- Until | M p’ - r | < tolerance
  - p’ <= Update(p’, M, r)
- Return p’

Serial Gauss-Seidel Relaxation

Loop until convergence:

For each equation j = 1 to n

Solve for P[j]

E.g. equation for P[1]:

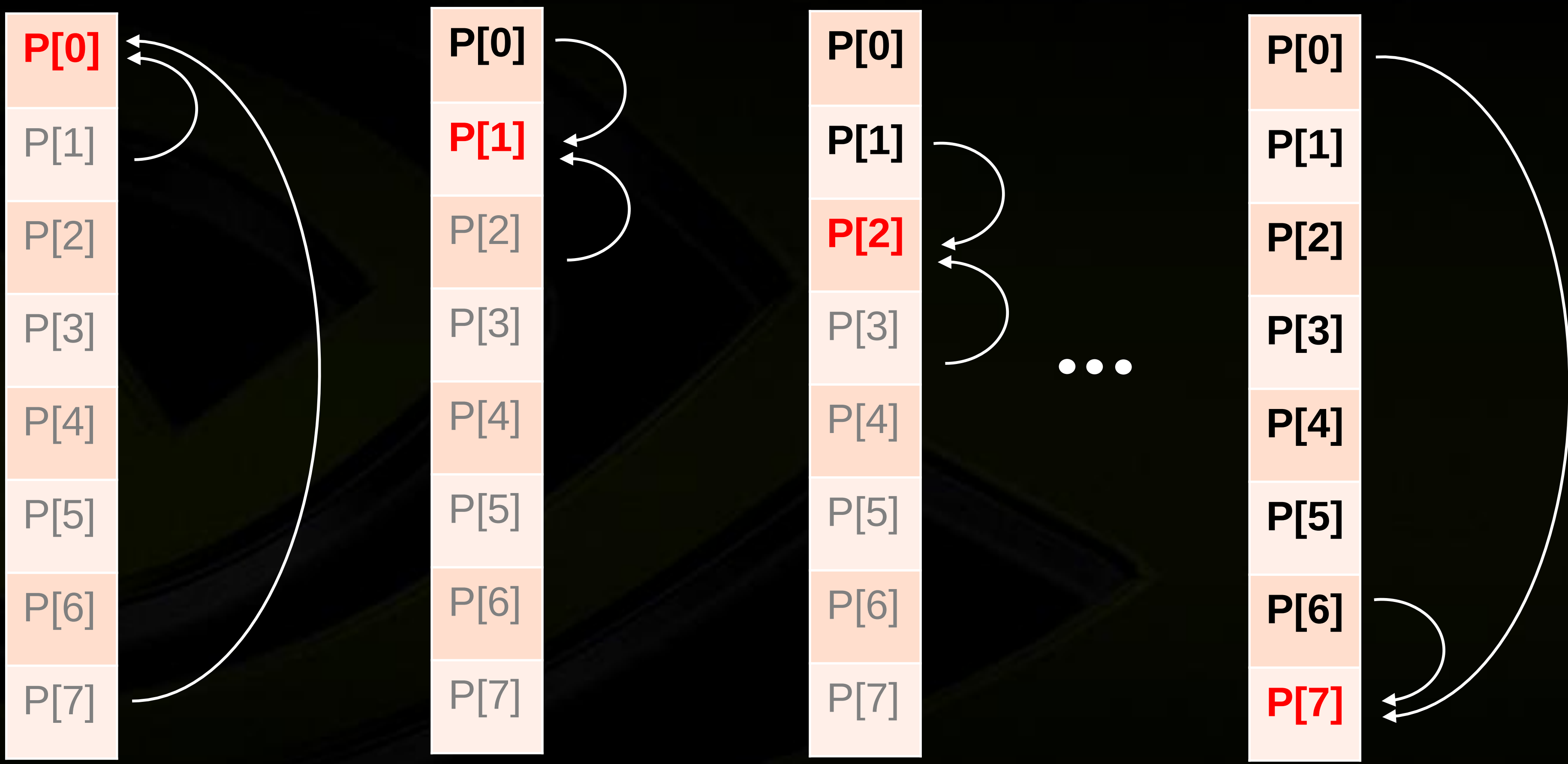
P[0] - 2P[1] + P[2] = h*h*RHS[1]

Rearrange terms:

$$P[1] = \frac{P[0] + P[2] - h*h*RHS[1]}{2}$$

One Pass of Serial Algorithm

$$P[0] = \frac{P[7] + P[1] - h*h*RHS[0]}{2} \qquad P[1] = \frac{P[2] + P[0] - h*h*RHS[1]}{2} \qquad P[2] = \frac{P[1] + P[3] - h*h*RHS[2]}{2} \qquad P[7] = \frac{P[6] + P[0] - h*h*RHS[7]}{2}$$



Red-Black Gauss-Seidel Relaxation

- Can choose any order in which to update equations
  - Convergence rate may change, but convergence still guaranteed
- "Red-black" ordering:

P[0]	P[1]	P[2]	P[3]	P[4]	P[5]	P[6]	P[7]

- Red (odd) equations independent of each other
- Black (even) equations independent of each other

Parallel Gauss-Seidel Relaxation

Loop n times (until convergence)

For each even equation j = 0 to n-1

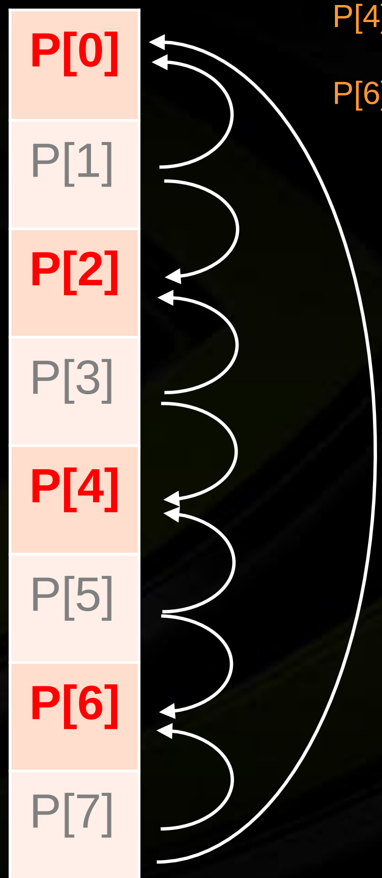
Solve for P[j]

For each odd equation j = 1 to n

Solve for P[j]

For loops are parallel – perfect for CUDA kernel

One Pass of Parallel Algorithm



$$P[0] = \frac{P[7] + P[1] - h*h*RHS[0]}{2}$$
$$P[2] = \frac{P[1] + P[3] - h*h*RHS[2]}{2}$$
$$P[4] = \frac{P[3] + P[5] - h*h*RHS[4]}{2}$$
$$P[6] = \frac{P[5] + P[7] - h*h*RHS[6]}{2}$$

$$P[1] = \frac{P[0] + P[2] - h*h*RHS[1]}{2}$$
$$P[3] = \frac{P[2] + P[4] - h*h*RHS[3]}{2}$$
$$P[5] = \frac{P[4] + P[6] - h*h*RHS[5]}{2}$$
$$P[7] = \frac{P[6] + P[0] - h*h*RHS[7]}{2}$$

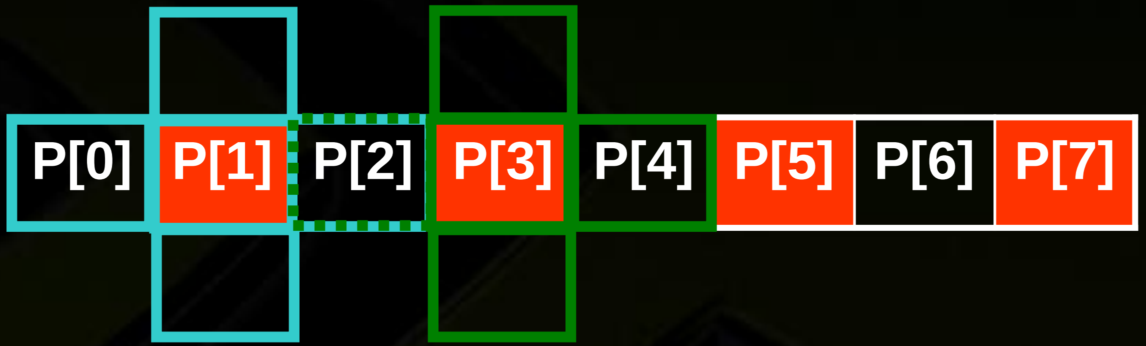
CUDA Pseudo-code

```
__global__ void RedBlackGaussSeidel(
  Grid P, Grid RHS, float h, int red_black)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  int j = blockIdx.y*blockDim.y + threadIdx.y;
  i*=2;
  if (j%2 != red_black) i++;
  int idx = j*RHS.jstride + i*RHS.istride;
  P.buf[idx] = 1.0/6.0*(-h*h*R.buf[idx] +
       P.buf[idx + P.istride] + P.buf[idx - P.istride] +
       P.buf[idx + P.jstride] + P.buf[idx - P.jstride]);
}

// on host:
for (int i=0; i < 100; i++) {
  RedBlackGaussSeidel<<<Dg, Db>>>(P, RHS, h, 0);
  RedBlackGaussSeidel<<<Dg, Db>>>(P, RHS, h, 1);
}
```

Optimizing the Poisson Solver

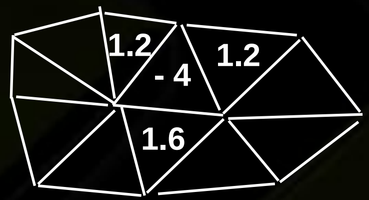
- Red-Black scheme is bad for coalescing
  - Read every other grid cell => half memory bandwidth
  - Lots of reuse between adjacent threads (blue and green)



- Texture cache (Fermi L1 cache) improves performance by 2x
  - Lots of immediate reuse, very small working set
  - In my tests, (barely) beats software-managed shared memory

Generalizing to non-grids

- What about discretization over non-Cartesian grids?
  - Finite element, finite volume, etc.
- Need discrete version of differential operator (Laplacian) to this geometry



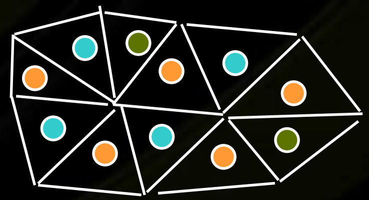
- Works out to same thing:

  L p = r

  where L is matrix of Laplacian discretizations

Graph Coloring

- Need partition into non-adjacent sets
  - Classic ‘graph coloring’ problem
  - Red-black is special case, where 2 colors suffice
  - Too many colors => not enough parallelism within each color
  - Not enough colors => hard coloring problem



Until convergence:

Update green terms in parallel

Update orange terms in parallel

Update blue terms in parallel

Back to the Poisson Matrix…

1D Poisson Matrix has particular sparse structure:

- 3 non-zeros per row, around the diagonal

-2	1						1
1	-2	1					
	1	-2	1				
		1	-2	1			
			1	-2	1		
				1	-2	1	
					1	-2	1
1						1	-2

P[0]
P[1]
P[2]
P[3]
P[4]
P[5]
P[6]
P[7]

$= \Delta x^2\ \mathrm{RHS}$

Approach 2: Direct Solver

- Solve the matrix equation directly
- Exploit sparsity pattern – all zeroes except diagonal, 1 above, 1 below = “tridiagonal” matrix

- Many applications for tridiagonal matrices
  - Vertical diffusion (adjacent columns do not interact)
  - ADI methods (e.g. separate 2D blur x blur, y blur)
  - Linear solvers (multigrid, preconditioners, etc.)

- Typically many small tridiagonal systems => per-CTA algorithm

What is a tridiagonal system?

$$\begin{pmatrix} b_1 & c_1 & & & & \\ a_2 & b_2 & c_2 & & & \\ & a_3 & b_3 & c_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & \ddots & \ddots & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ \vdots \\ d_n \end{pmatrix}$$

A Classic Sequential Algorithm

- **Gaussian elimination in tridiagonal case (Thomas algorithm)**

$$
\begin{pmatrix}
1 & c'_1 & & & \\
0 & 1 & c'_2 & & \\
 & 0 & 1 & c'_3 & \\
 & & 0 & 1 & c'_4 \\
 & & & 0 & 1
\end{pmatrix}
\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix}
=
\begin{pmatrix} d'_1 \\ d'_2 \\ d'_3 \\ d'_4 \\ d'_5 \end{pmatrix}
$$

Phase 1: Forward Elimination

Phase 2: Backward Substitution

Cyclic Reduction: Parallel algorithm

Basic linear algebra:
Take any row, multiply by scalar, add to another row => Solution unchanged

B1	C1						A1		X1		R1
A2	B2	C2							X2		R2
	A3	B3	C3						X3		R3
		A4	B4	C4					X4		R4
			A5	B5	C5				X5	=	R5
				A6	B6	C6			X6		R6
					A7	B7	C7		X7		R7
C8						A8	B8		X8		R8

Scale Equations 3 and 5

	B1	C1						A1	X1		R1
	A2	B2	C2						X2		R2
-A4/B3 *		A3	B3	C3					X3		R3
			A4	B4	C4				X4		R4
-C4/B5 *				A5	B5	C5			X5	=	R5
					A6	B6	C6		X6		R6
						A7	B7	C7	X7		R7
	C8						A8	B8	X8		R8

Add scaled Equations 3 & 5 to 4

B1	C1						A1
A2	B2	C2					
	A3’	-A4	C3’				
		A4	B4	C4			
			A5’	-C4	C5’		
				A6	B6	C6	
					A7	B7	C7
C8						A8	B8

X1
X2
X3
X4
X5
X6
X7
X8

=

R1
R2
R3
R4
R5
R6
R7
R8

Zeroes entries 4,3 and 4,5

B1	C1						A1	X1		R1
A2	B2	C2						X2		R2
	A3’	-A4	C3’					X3		R3
	A3’		B4’		C5’			X4	=	R4’
			A5’	-C4	C5’			X5		R5
				A6	B6	C6		X6		R6
					A7	B7	C7	X7		R7
C8						A8	B8	X8		R8

Repeat operation for all equations

B1’		C2’				A8’		X1		R1’
	B2’		C3’				A1’	X2		R2’
A2’		B3’		C4’				X3		R3’
	A3’		B4’		C5’			X4		R4’
		A4’		B5’		C6’		X5	=	R5’
			A5’		B6’		C7’	X6		R6’
C8’				A6’		B7’		X7		R7’
	C1’				A7’		B8’	X8		R8’

Permute – 2 independent blocks

Odd	B1’	C2’		A8’					X1		R1’
	A2’	B3’	C4’						X3		R3’
		A4’	B5’	C6’					X5		R5’
	C8’		A6’	B7’					X7	=	R7’
Even					B2’	C3’		A1’	X2		R3’
					A3’	B4’	C5’		X4		R4’
						A5’	B6’	C7’	X6		R6’
					C1’		A7’	B8’	X8		R8’

Cyclic Reduction Ingredients

- Apply this transformation (pivot + permute)
- Split (n x n) into 2 independent (n/2 x n/2)
- Proceed recursively

- Two approaches:
  - Recursively reduce both submatrices until n 1x1 matrices obtained. Solve resulting diagonal matrix.
  - Recursively reduce odd submatrix until single 1x1 system. Solve system. Reverse process via back-substitution.

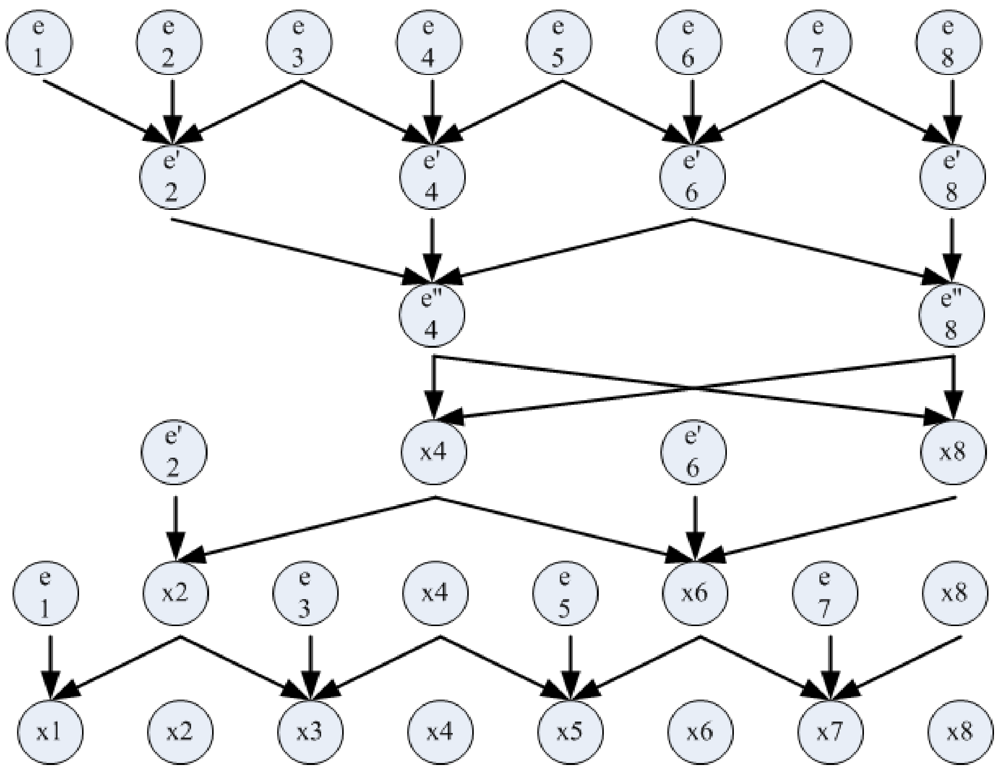
Cyclic Reduction (CR)

4 threads working

1 thread working

Forward Reduction

Backward Substitution



8-unknown system

4-unknown system

2-unknown system

Solve 2 unknowns

Solve the rest 2 unknowns

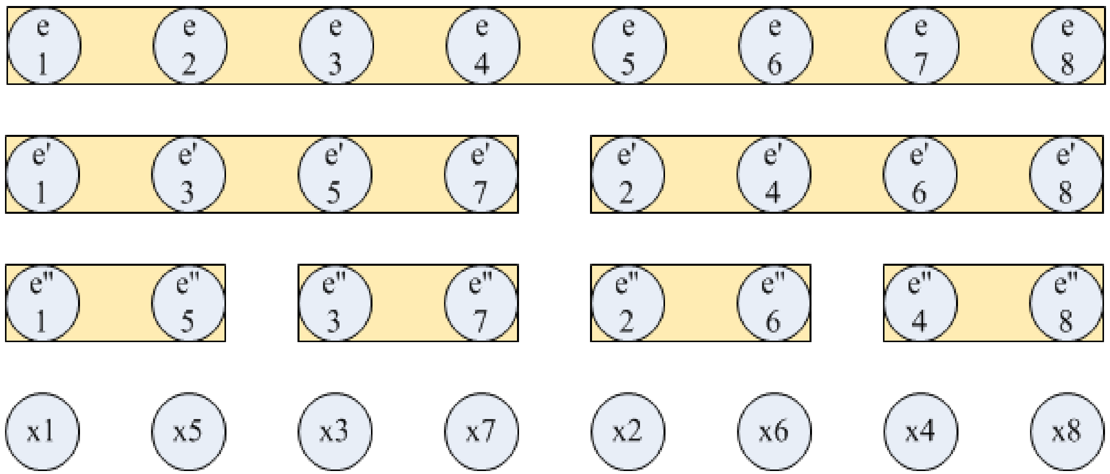
Solve the rest 4 unknowns

$2*\log_2(8)-1 = 2*3-1 = 5$ steps

Parallel Cyclic Reduction (PCR)

4 threads working

Forward Reduction



One 8-unknown system

Two 4-unknown systems

Four 2-unknown systems

Solve all unknowns

$\log_2 (8) = 3$ steps

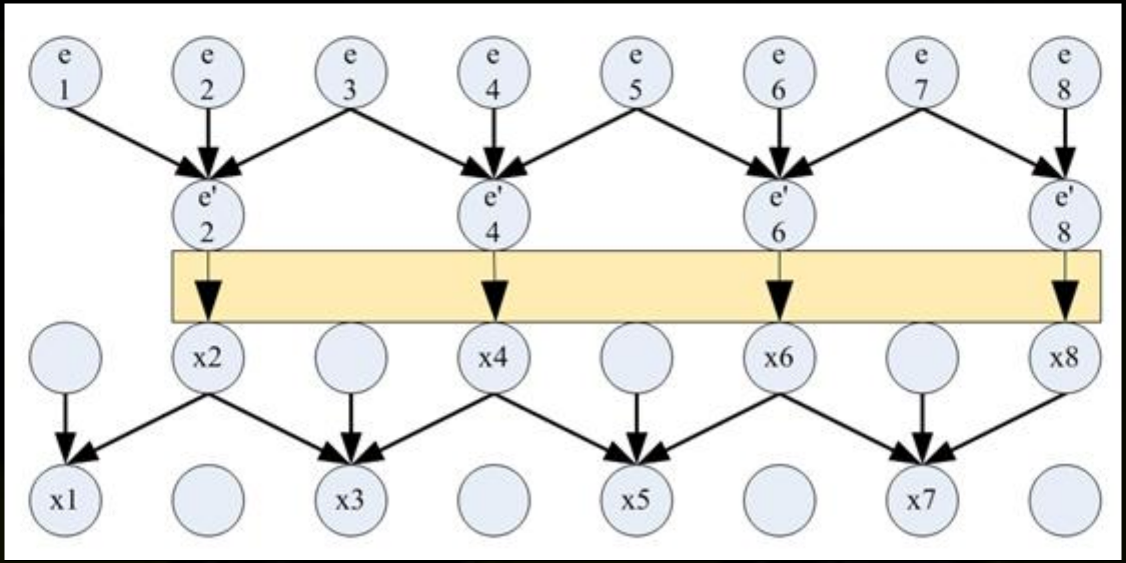
Work vs. Step Efficiency

- CR does O(n) work, requires 2 log(n) steps
- PCR does O(n log n) work, requires log(n) steps

- Smallest granularity of work is 32 threads: performing fewer than 32 math ops = same cost as 32 math ops

- Here’s an idea:
  - Save work when > 32 threads active (CR)
  - Save steps when < 32 threads active (PCR)

Hybrid Algorithm



Switch to PCR

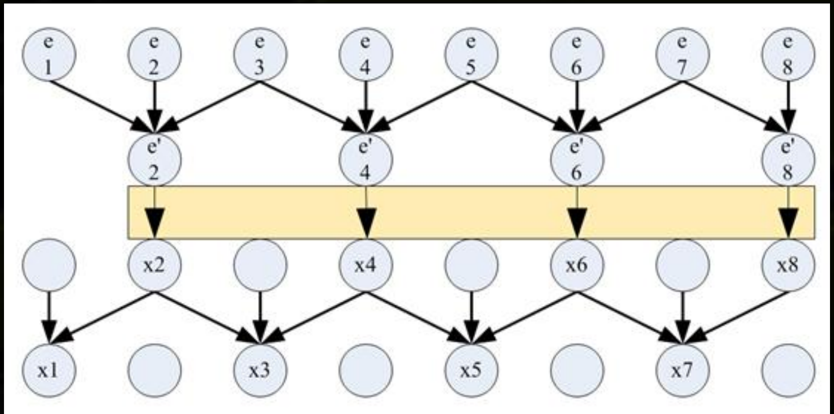
Switch back to CR

System size reduced at the beginning
No idle processors
Fewer algorithmic steps

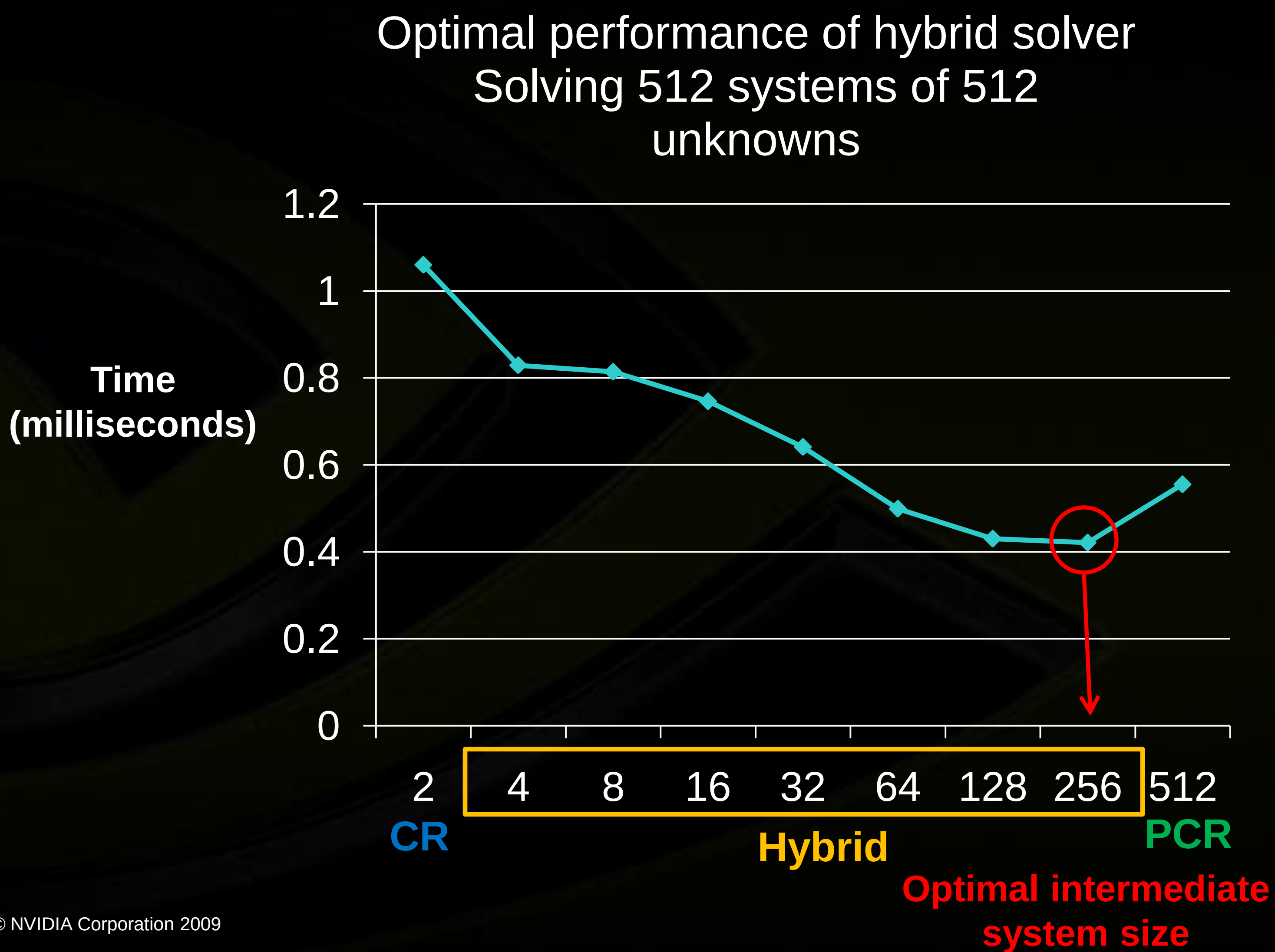
Even more beneficial because of:
bank conflicts
control overhead

PCR vs Hybrid

- Make tradeoffs between the computation, memory access, and control
  - The earlier you switch from CR to PCR
    - The fewer bank conflicts, the fewer algorithmic steps
    - But more work

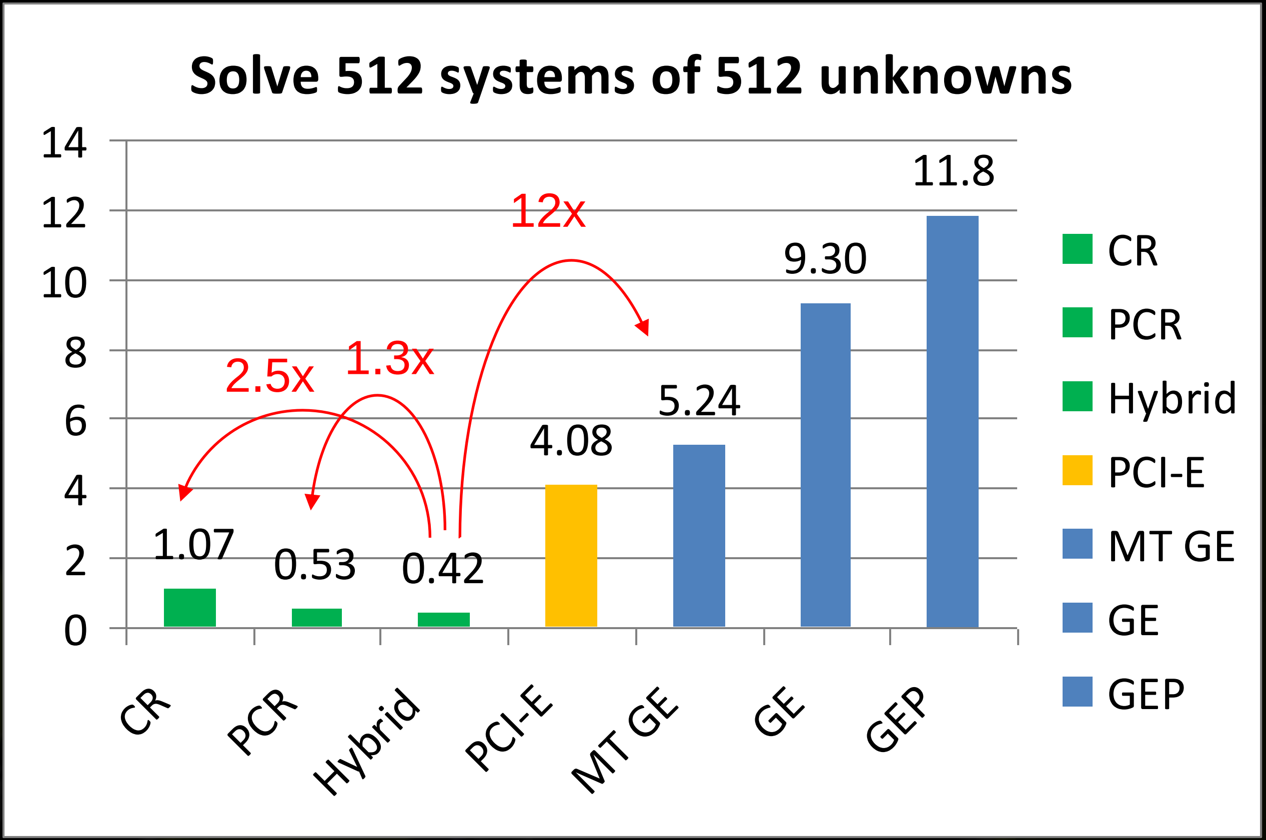


Hybrid Solver – Optimal cross-over



Results: Tridiagonal Linear Solver

Time (milliseconds)



PCI-E: CPU-GPU data transfer
MT GE: multi-threaded CPU Gaussian Elimination
GEP: CPU Gaussian Elimination with pivoting (from LAPACK)

From Zhang *et al.*, "Fast Tridiagonal Solvers on GPU." PPoPP 2010.

Summary

- Linear PDE => Linear solver (e.g. Poisson Equation)
- 2 basic approaches: Iterative vs. Direct
- Parallel iterative solver (Red Black Gauss Seidel)
  - Design update procedure so multiple terms can be updated in parallel
- Parallel direct solver (Cyclic Reduction)
  - Exploit structure of matrix to solve using parallel operations
- ‘General purpose’ solvers largely mythical. Most people use special purpose solvers => Lots of good research potential mining this field