

Aspects of Scientific Computing on GPUs

**Robert Strzodka (MPII),
Dominik Göddeke (TUDo), Dominik Behr (AMD)**

**PPAM 2009 - Conference on Parallel Processing and
Applied Mathematics
Wrocław, Poland, September 13-16, 2009**

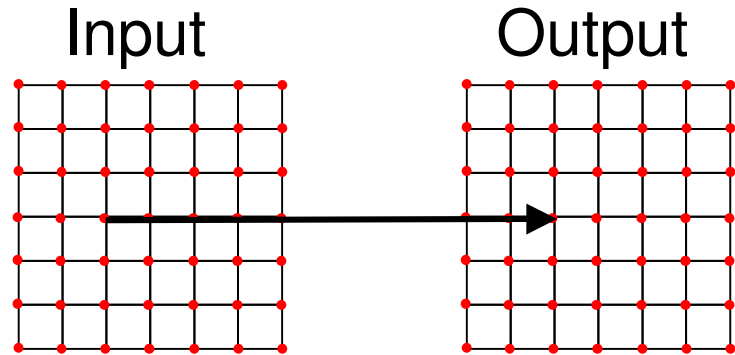
www.gpgpu.org/ppam2009

Overview

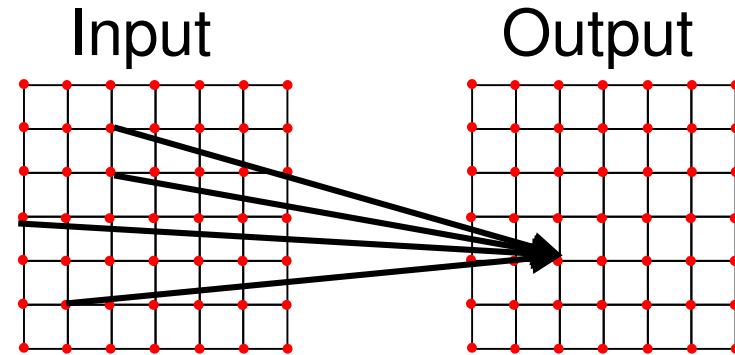
- **Types of Parallel Data Flow**
- **Parallel Prefix or Scan**
- **Precision and Accuracy**
- **Efficient PDE Solvers**
- **Mixed Precision Refinement**

Parallel Data-Flow: Map, Gather and Scatter

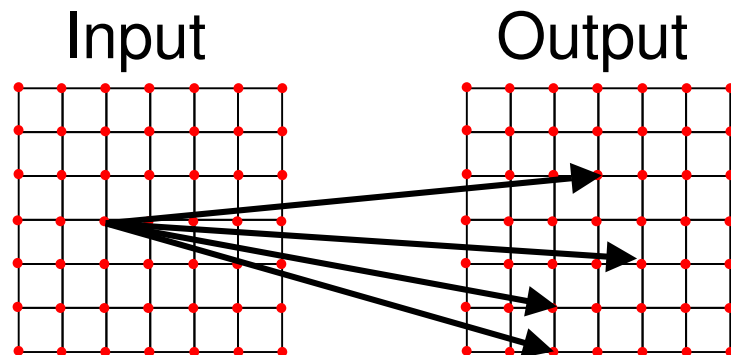
Map: $x = f(a)$



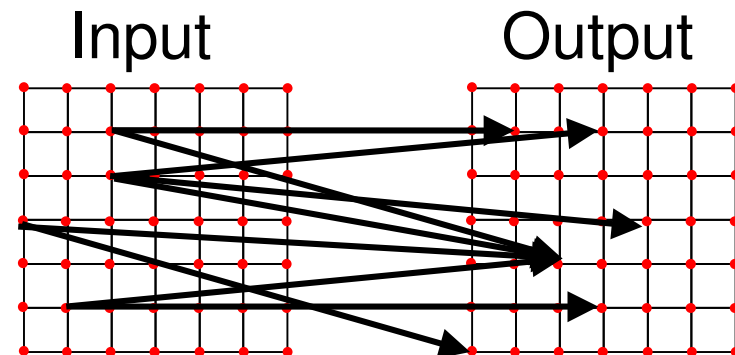
Gather: $x = f(a(1,2), a(3,5), \dots)$



Scatter: $x(2,3) = f(a), x(6,7) = g(a), \dots$



General: $x(2,3) = f(a(1,2), a(3,5), \dots)$,
 $x(6,7) = f(a(6,2), a(7,5), \dots)$



Performance of Gather and Scatter

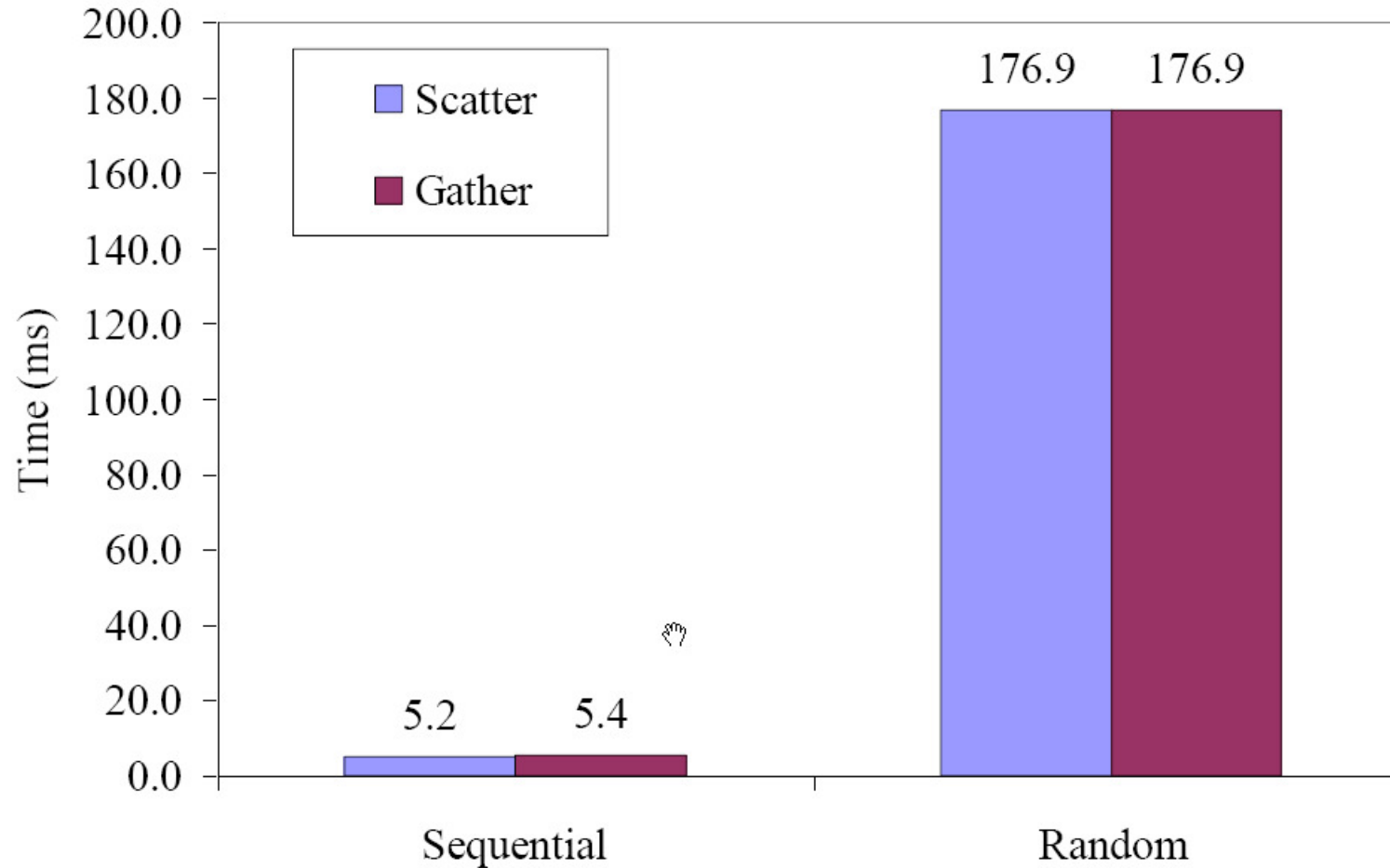
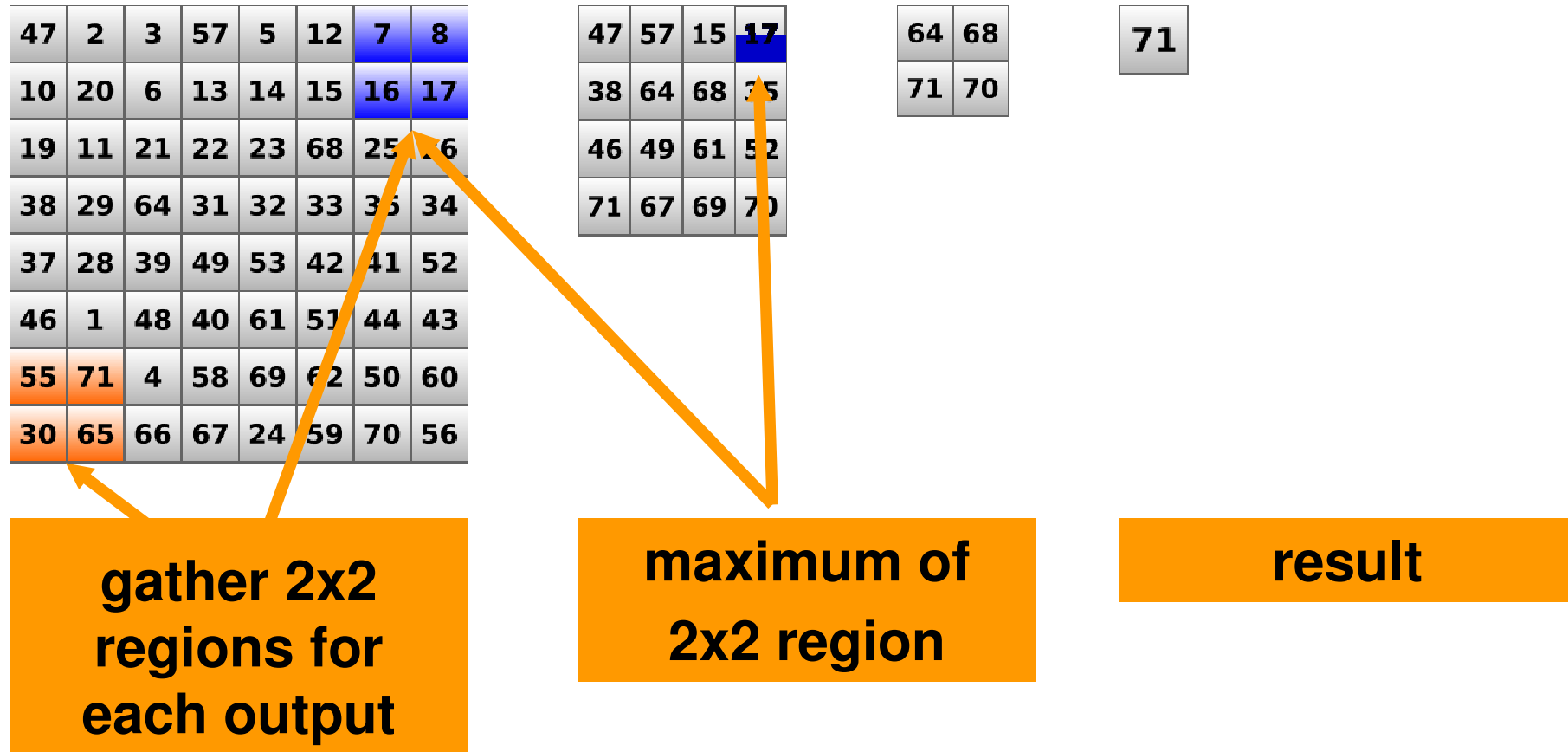


chart courtesy of
Naga Govindaraju

Parallel Reduction I, e.g. Maximum of an Array



- For commutative operators (e.g. +, *, max) this is encapsulated into a single function call.
- For a more detailed discussion see, Mark Harris' CUDA optimization talk from SC 2007: <http://www.gpgpu.org/sc2007/>

Parallel Reduction II

47	2	3	57	5	12	7	8
10	20	6	13	14	15	16	17
19	11	21	22	23	68	25	26
38	29	64	31	32	33	35	34
37	28	39	49	53	42	41	52
46	1	48	40	61	51	44	43
55	71	4	58	69	62	50	60
30	65	66	67	24	59	70	56

64	68
71	70

71

**gather at once
larger regions**

intermediates

result

- Region for gathering should fit the on-chip memory
- Last step gathering on CPU or GPU depending on where result is needed next

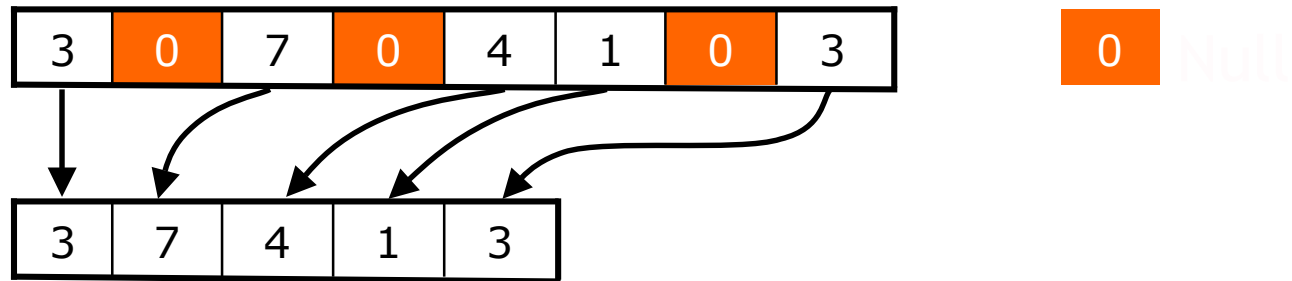
Overview

- Types of Parallel Data Flow
- **Parallel Prefix or Scan**
- Precision and Accuracy
- Efficient PDE Solvers
- Mixed Precision Refinement

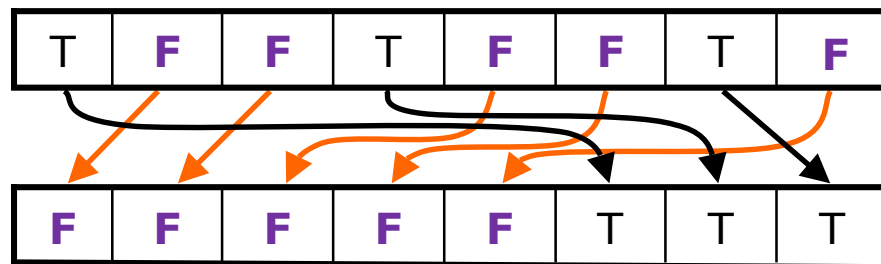
slides courtesy of
Shubho Sengupta

Motivation

- Stream Compaction



- Split



Motivation

- **Common scenarios in parallel computing**
 - Variable output per thread
 - Threads want to perform a split – radix sort, building trees
- **“What came before/after me?”**
- **“Where do I start writing my data?”**
- **Scan answers this question**

Scan

- Each element is a sum of all the elements to the left of it (Exclusive)
- Each element is a sum of all the elements to the left of it and itself (Inclusive)

3	1	7	0	4	1	6	3
---	---	---	---	---	---	---	---

Input

0	3	4	11	11	15	16	22
---	---	---	----	----	----	----	----

Exclusive

3	4	11	11	15	16	22	25
---	---	----	----	----	----	----	----

Inclusive

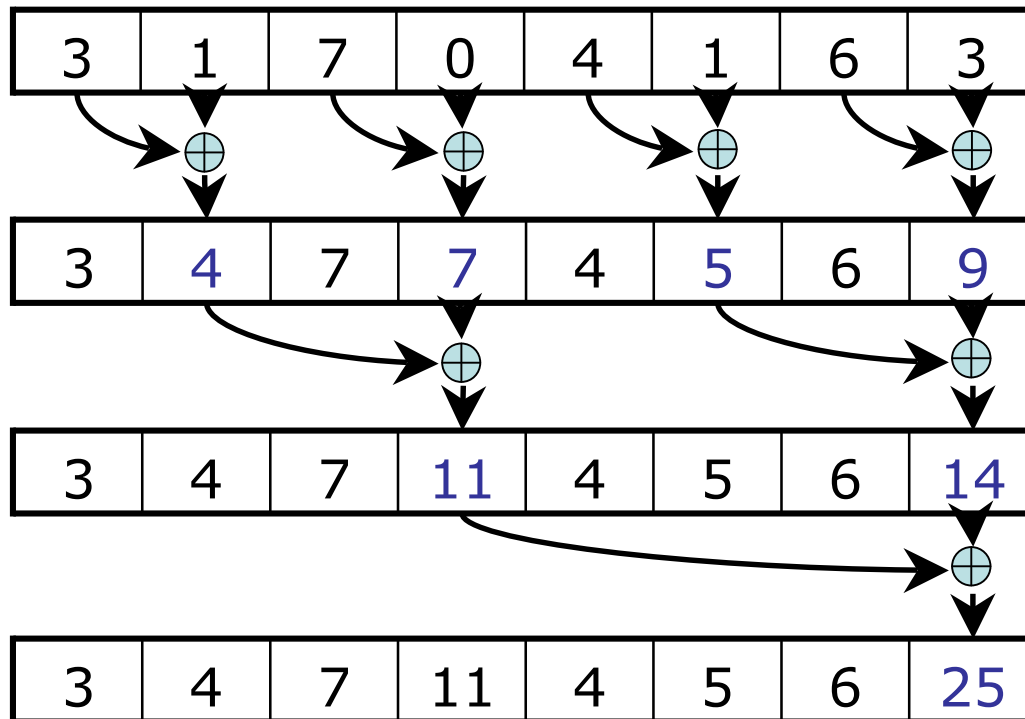
Scan – the past

- **First proposed in APL (1962)**
- **Used as a data parallel primitive in the Connection Machine (1990)**
- **Guy Blelloch used scan as a primitive for various parallel algorithms (1990)**

Scan – the present

- **First GPU implementation by Daniel Horn (2004), $O(n \log n)$**
- **Subsequent GPU implementations by**
 - Hensley (2005) $O(n \log n)$, Sengupta (2006) $O(n)$, Greß (2006) $O(n)$ 2D
- **NVIDIA CUDA implementation by Mark Harris (2007), $O(n)$, space efficient**

Scan - Reduce



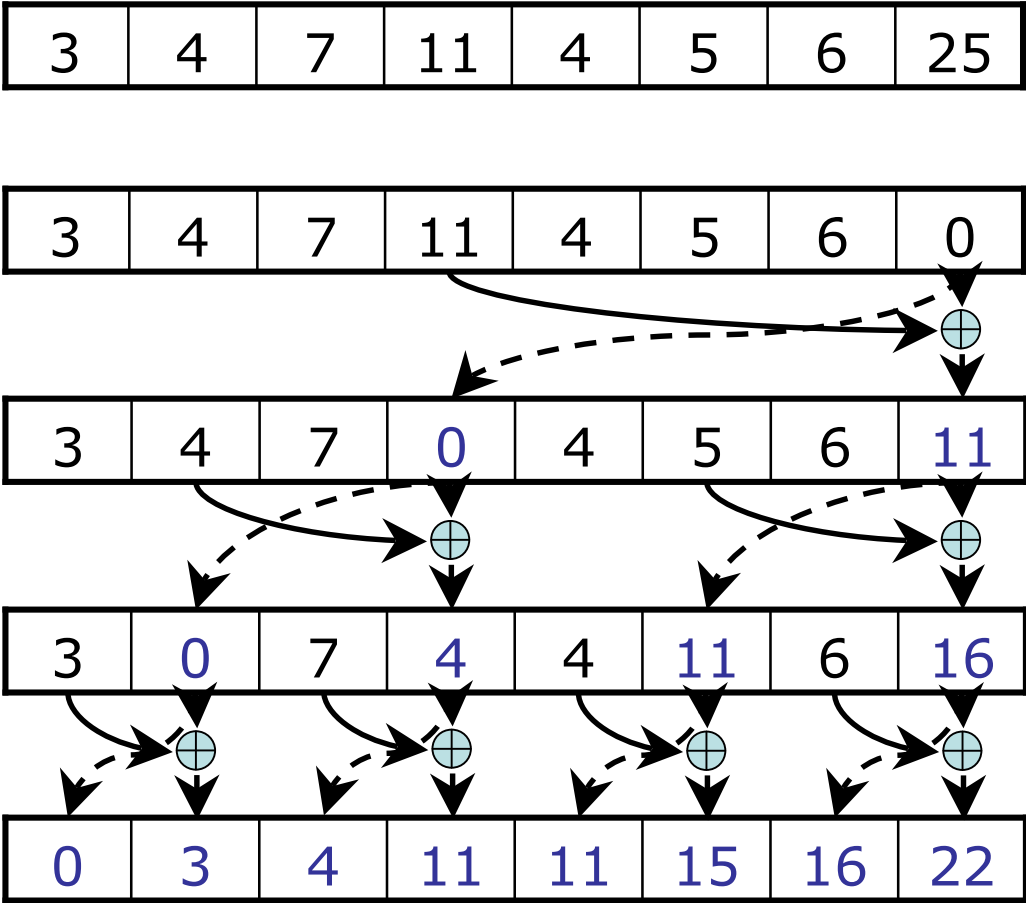
- $\log n$ steps

- Work halves each step

- $O(n)$ work

- In place, space efficient

Scan - Down Sweep



- log n steps

- Work doubles each step

- $O(n)$ work

- In place, space efficient

Segmented Scan

- **Input**

3	1	7	0	4	1	6	3
---	---	---	---	---	---	---	---

- **Scan within each segment in parallel**

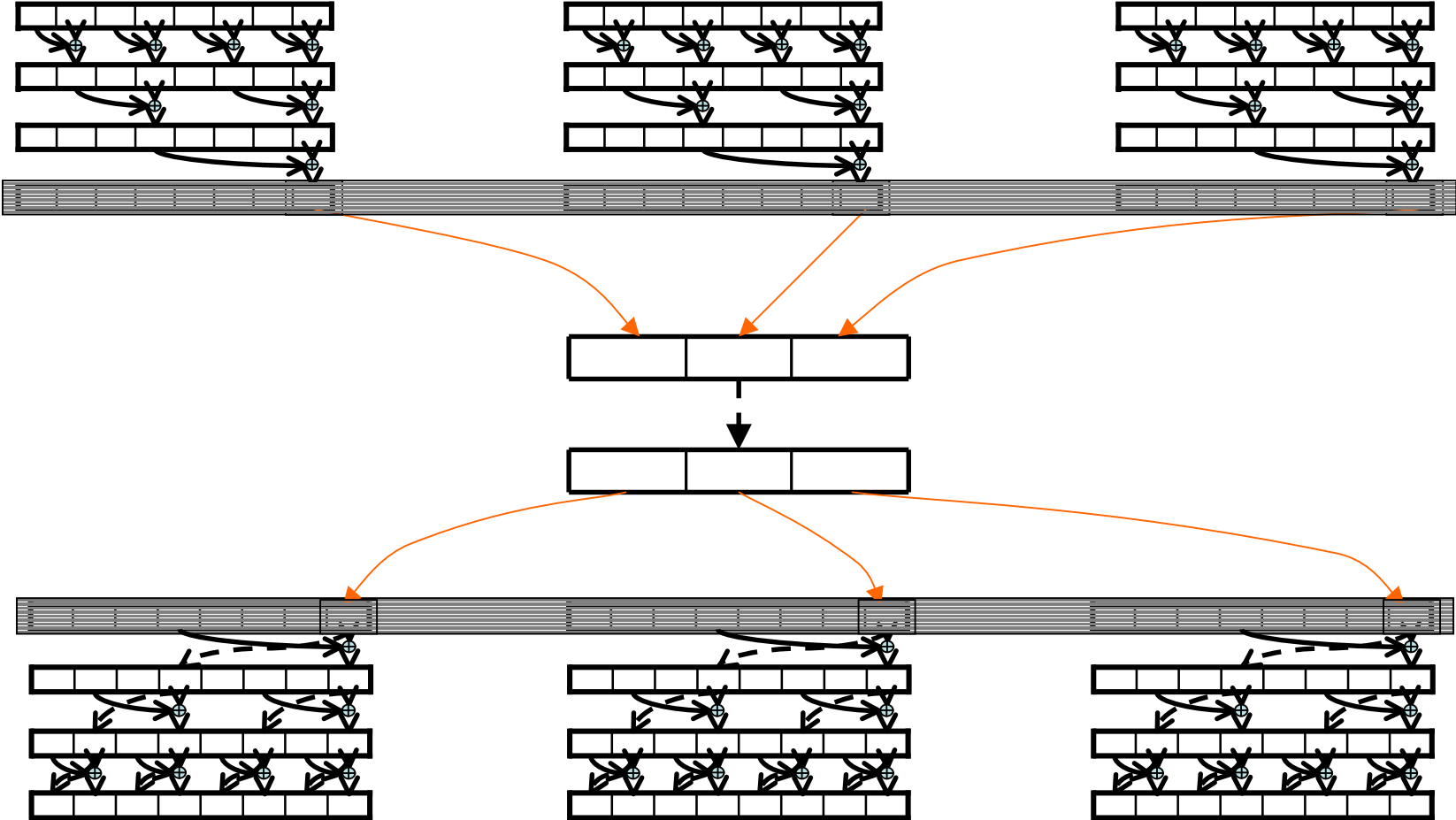
- **Output**

0	3	0	7	7	0	1	7
---	---	---	---	---	---	---	---

Segmented Scan

- **Introduced by Schwartz (1980)**
- **Forms the basis for a wide variety of algorithms**
 - Radixsort, Quicksort
 - Sparse Matrix-Vector Multiply
 - Convex Hull
 - Solving recurrences
 - Tree operations

Segmented Scan – Large Input

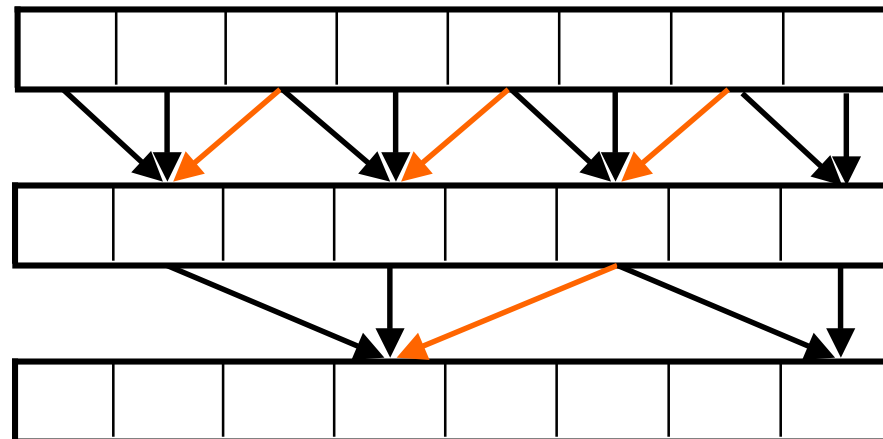


Segmented Scan – Advantages

- **Operations in parallel over all the segments**
- **Irregular workload since segments can be of any length**
- **Can simulate divide-and-conquer recursion since additional segments can be generated**

Segmented Scan Variant: Tridiagonal Solver

- Tridiagonal system of n rows solved in parallel
- Then for each of the m columns in parallel
- Read pattern is similar to but more complex than scan



Overview

- Types of Parallel Data Flow
- Parallel Prefix or Scan
- **Precision and Accuracy**
- Efficient PDE Solvers
- Mixed Precision Refinement

Roundoff and Cancellation

Roundoff examples for the **float s23e8** format

additive roundoff $a = 1 + 0.00000004 = 1.00000004 =_{fl} 1$
multiplicative roundoff $b = 1.0002 * 0.9998 = 0.99999996 =_{fl} 1$
cancellation $c \in \{a, b\}$ $(c - 1) * 10^8 = \pm 4 =_{fl} 0$

Cancellation promotes the small error 0.00000004 to the absolute error 4 and a relative error of order one.

Order of operations can be crucial:

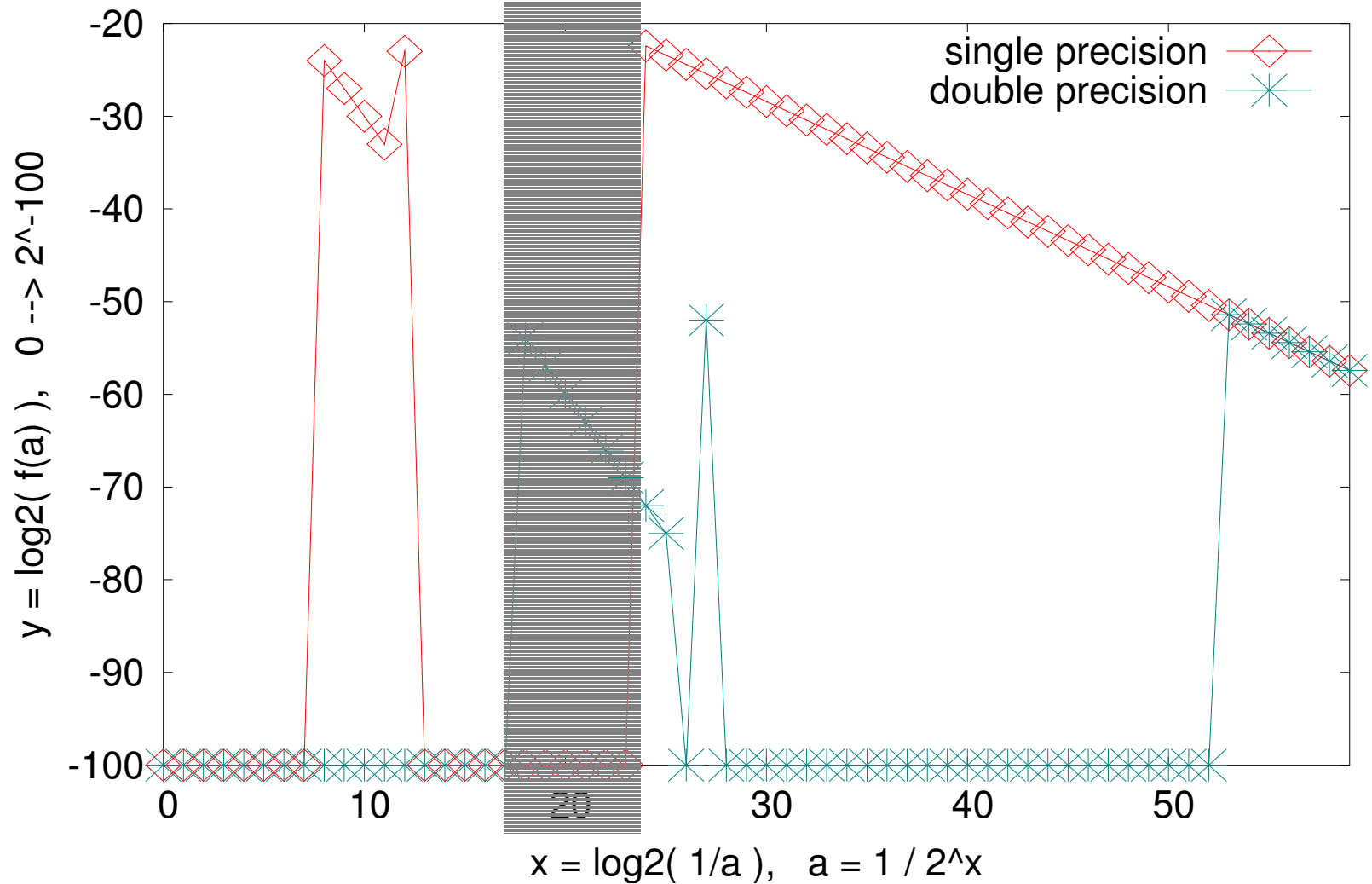
$$1 + 0.00000004 - 1 =_{fl} 0$$
$$1 - 1 + 0.00000004 =_{fl} 0.00000004$$

With the **double s52e11** format no problems above, but ...

The Erratic Roundoff Error

Roundoff error for: $0 = f(a) := |(1+a)^3 - (1+3a^2) - (3a+a^3)|$

← Smaller is better ←



The Dominant Data Error

- **Data error occurs when the exact value has to be truncated for storage in the binary format, e.g.**
 - π , $\sqrt{2}$, $\sin(2)$, $\exp(2)$, $1/3$, ...
 - In fact, any value, e.g. 0.1, except combinations of 2^b
- **So more precision is usually better because**
 - for float s23e8: $1 + 4e-8 =_{fl} 1$
 - for double s52e11: $1 + 4e-15 =_{fl} 1$
- **How can float be better than double then?**
 - There is **no data error** in the operands
 - Alternatively, the errors **cancel out** themselves favorably

Understanding Floating Point Operations

- **Number representation s23e8**

- $a = | 1 \text{ bit sign } s_a | 23 \text{ bit mantissa } m_a | 8 \text{ bit exponent } e_a |$

- **Multiplication $a * b$**

- **Operations:** $s_a * s_b$, $m_a * m_b$, $e_a + e_b$

- **Exact format:** $s46e9 = s23e8 * s23e8$

- **Main error:** Mantissa truncated from **46** bit to 23 bit

- **Addition $a + b$**

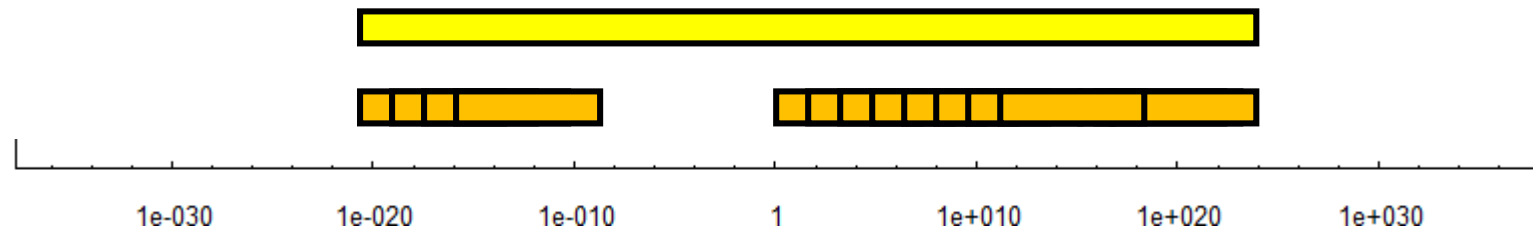
- **Operations:** $e_{\text{diff}} = e_a - e_b$, $m_a + (m_b \gg e_{\text{diff}})$, normalize

- **Exact format:** $s278e8 = s23e8 + s23e8$

- **Main error:** Mantissa truncated from **278** bit to 23 bit

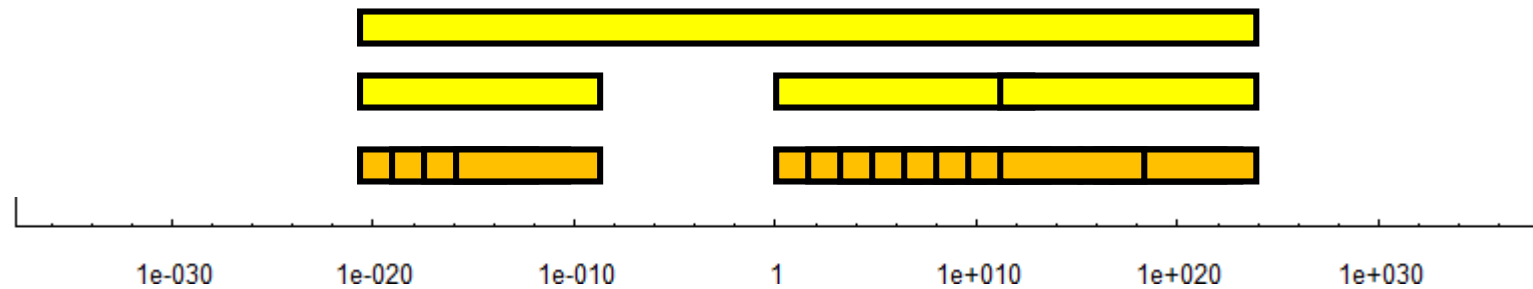
Commutative Summation

$$s = \sum_{i \in I} a_i$$



$$s = s_0 + s_1 + s_2$$

$$s_2 = \sum_{i \in I_2} a_i \quad s_1 = \sum_{i \in I_1} a_i \quad s_0 = \sum_{i \in I_0} a_i$$



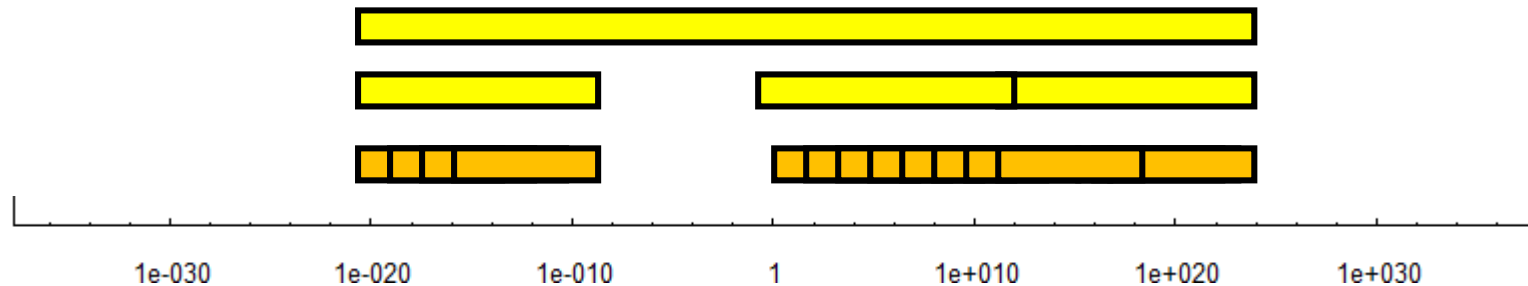
Commutative Summation Example

- $1 + 0.00000004 =_{\text{db}} 1.00000004 =_{\text{fl}} 1$
- In **float s23e8**
 $s = \sum a_i = 1/2 + 1/2 + 0.00000004 - 0.00000003 =_{\text{fl}} 1$
- In **double s52e11**
 $s = \sum a_i =_{\text{db}} 1.00000001$
- In **mixed double/float**
 $s_0 = \sum_0 a_i = 1/2 + 1/2 =_{\text{fl}} 1$
 $s_1 = \sum_1 a_i = 0.00000004 - 0.00000003 =_{\text{fl}} 0.00000001$
 $s = s_0 + s_1 =_{\text{db}} 1.00000001$

Dependent Summation

$$S =_{db} S_0 + S_1 + S_2$$

$$S_2 =_{fl} \sum_{i \in I_2} a_i \quad S_1 =_{fl} \sum_{i \in I_1} a_i \quad S_0 =_{fl} \sum_{i \in I_0} a_i$$



$$S =_{db} \sum_{i \in I} a_i, \quad a_i = f_{db}(a_{i-1}), \quad \text{with slow double precision } f_{db}()$$

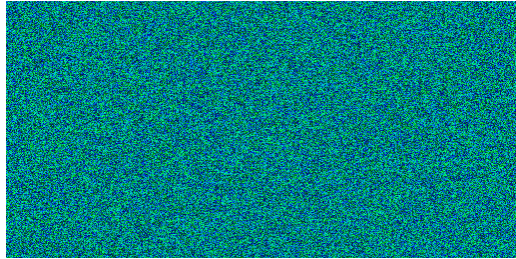
$$S_0 =_{fl} \sum_{i \in I_0} a_i, \quad S_1 =_{fl} \sum_{i \in I_1} a_i, \quad S_2 =_{fl} \sum_{i \in I_2} a_i, \quad S =_{db} S_0 + S_1 + S_2$$

$$a_i = f_{fl}(a_{i-1}), \quad \text{with fast single precision } f_{fl}()$$

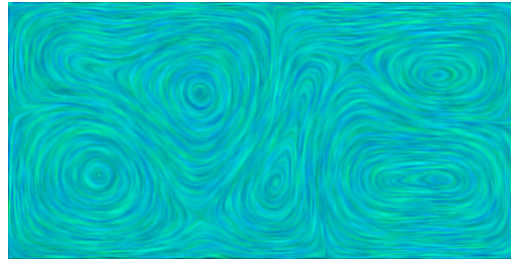
Overview

- Types of Parallel Data Flow
- Parallel Prefix or Scan
- Precision and Accuracy
- **Efficient PDE Solvers**
- Mixed Precision Refinement

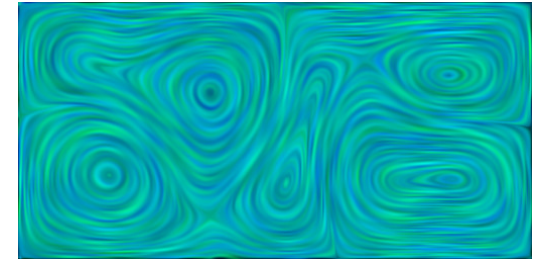
PDE Example – Anisotropic Diffusion



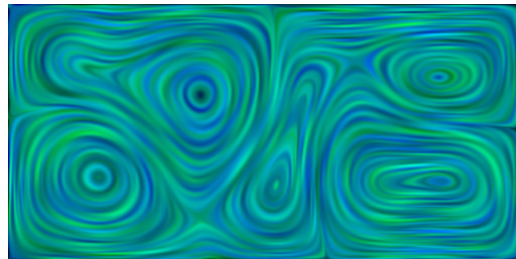
Initial image



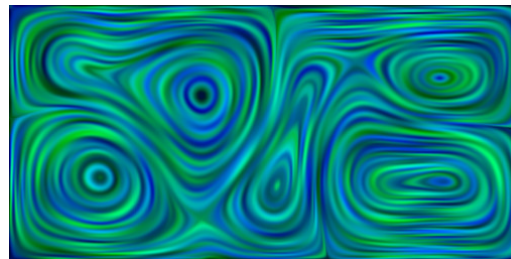
Step 1



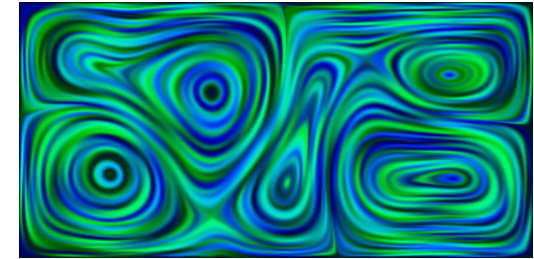
Step 2



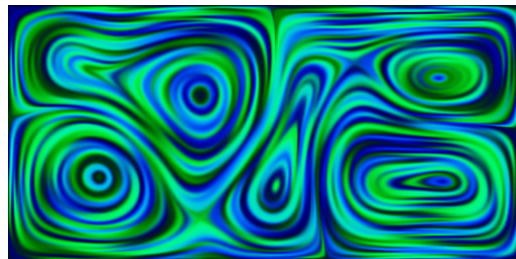
Step 3



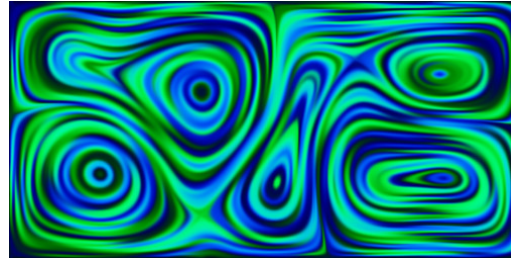
Step 4



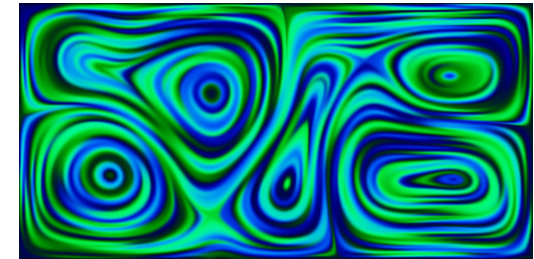
Step 5



Step 6



Step 7



Step 8

images courtesy of
Tobias Preusser

PDE Example – Anisotropic Diffusion

Initial image $u_0 : \Omega \rightarrow [0,1]$, unknown $u : (\Omega, \mathfrak{R}^+) \rightarrow \mathfrak{R}$

$$\partial_t u - \operatorname{div}(G(\nabla u_\sigma) \nabla u) = 0 \quad \text{in } \mathfrak{R}^+ \times \Omega$$

$$u(0) = u_0 \quad \text{in } \Omega$$

$$\partial_\nu u = 0 \quad \text{on } \mathfrak{R}^+ \times \partial\Omega$$

- linear

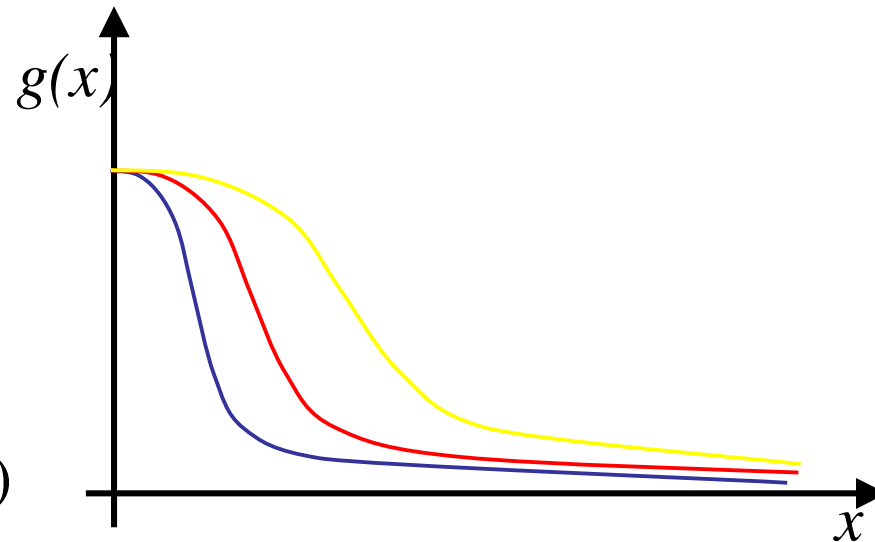
$$G(v) := 1$$

- isotropic non-linear

$$G(v) := g(\|v\|) \text{ scalar}$$

- anisotropic

$$G(v) := B^T(v) \begin{pmatrix} g_1(\|v\|) & 0 \\ 0 & g_2(\|v\|) \end{pmatrix} B(v)$$



Space Discretization

- **Finite Differences**

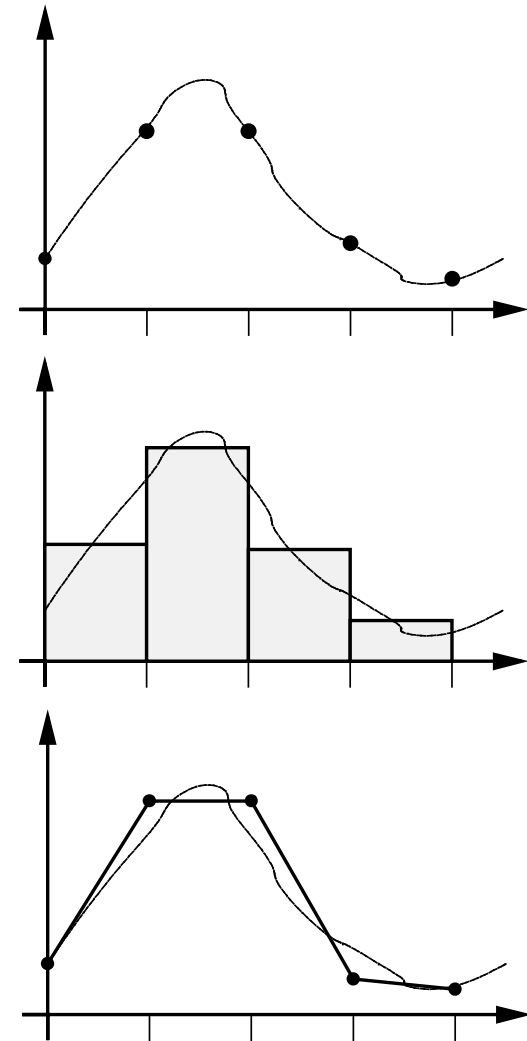
- **Interpolative** approach: simple and fast
- Usually interaction with direct neighbors

- **Finite Volumes**

- **Volumetric** approach: mass conservation
- Good at **discontinuities**, less for smooth data
- Interaction over element boundaries

- **Finite Elements**

- **Approximative** approach: error minimization
- Good handling of **deformed, unstructured** grids
- Interaction of basis functions (all neighbors)



Diffusion Example – Discretization

continuous model

$$\partial_t u - \operatorname{div}(G(\nabla u_\sigma) \nabla u) = 0$$

time disc. (semi-implicit)

$$u^{n+1} - \tau^n \operatorname{div}(G(\nabla u_\sigma^n) \nabla u^{n+1}) = u^n$$

space disc. (Finite Differences)

$$\bar{U}^{n+1} - \tau^n \operatorname{div}_h(G(\nabla_h \bar{U}_\sigma^n) \nabla_h \bar{U}^{n+1}) = \bar{U}^n$$

linear equation system

$$A[\nabla_h \bar{U}^n] \cdot \bar{U}^{n+1} = \bar{U}^n, \quad A[\nabla_h \bar{U}^n] := \mathbf{1} - \tau^n \operatorname{div}_h(G(\nabla_h \bar{U}_\sigma^n) \nabla_h \cdot)$$

● Typical situation in semi-implicit schemes

- Matrix A depends non-linearly on explicit data
- Linear equation system must be solved

● Solvers on GPUs have similar requirements as on parallel computers

- Parallel processing of matrix entries
- Examples: Jacobi solver, conjugate gradient, block-SOR, Multigrid

Diffusion Example – Linear Algebra

linear equation system

$$A[\nabla_h \bar{U}^n] \cdot \bar{U}^{n+1} = \bar{U}^n, \quad A[\nabla_h \bar{U}^n] := \mathbf{1} - \tau^n \operatorname{div}_h \left(G(\nabla_h \bar{U}_\sigma^n) \nabla_h \cdot \right)$$

two main computational tasks

1. Derivatives and nonlinear functions >>> Assembly of the matrix
2. Iterative linear equation solver >>> Matrix vector product

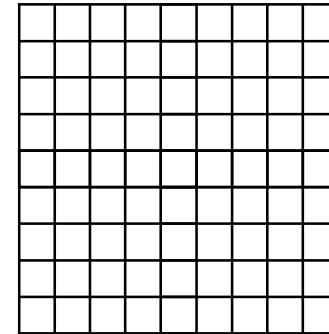
structure of the normal matrix vector product

$$\left(A \bar{U} \right)_\alpha = \sum_{\beta} A_{\alpha,\beta} \bar{U}_\beta = A_{\alpha,\beta_0} \bar{U}_{\beta_0} + A_{\alpha,\beta_1} \bar{U}_{\beta_1} + A_{\alpha,\beta_2} \bar{U}_{\beta_2} + \dots$$

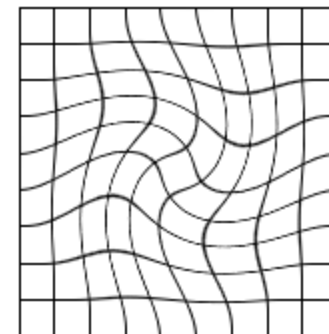
- For each pair of data items (A,U) 2 floats must be read
- And 2 operations computed: multiply and add
- Arithmetic intensity = 1
- Execution is bandwidth bound

Discretization Grids

- Equidistant grid
 - topology: implicit
 - geometry: implicit
 - access: **direct**



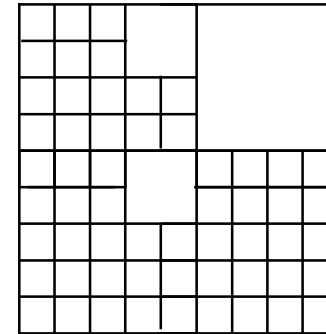
- Generalized tensor-product grid
 - topology: implicit
 - geometry: explicit
 - access: **direct**



Discretization Grids

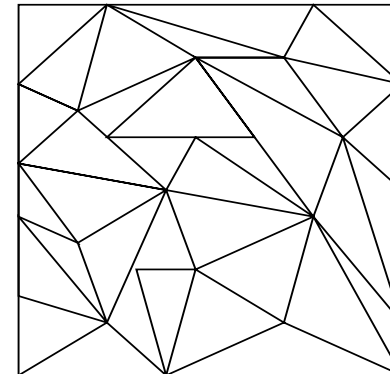
- **Adaptive grid**

- topology: **explicit**
- geometry: implicit/explicit
- access: hash, tree or page table



- **Unstructured grid**

- topology: **explicit**
- geometry: explicit
- access: index array



Structured and Unstructured SpMV

→ Larger is better →

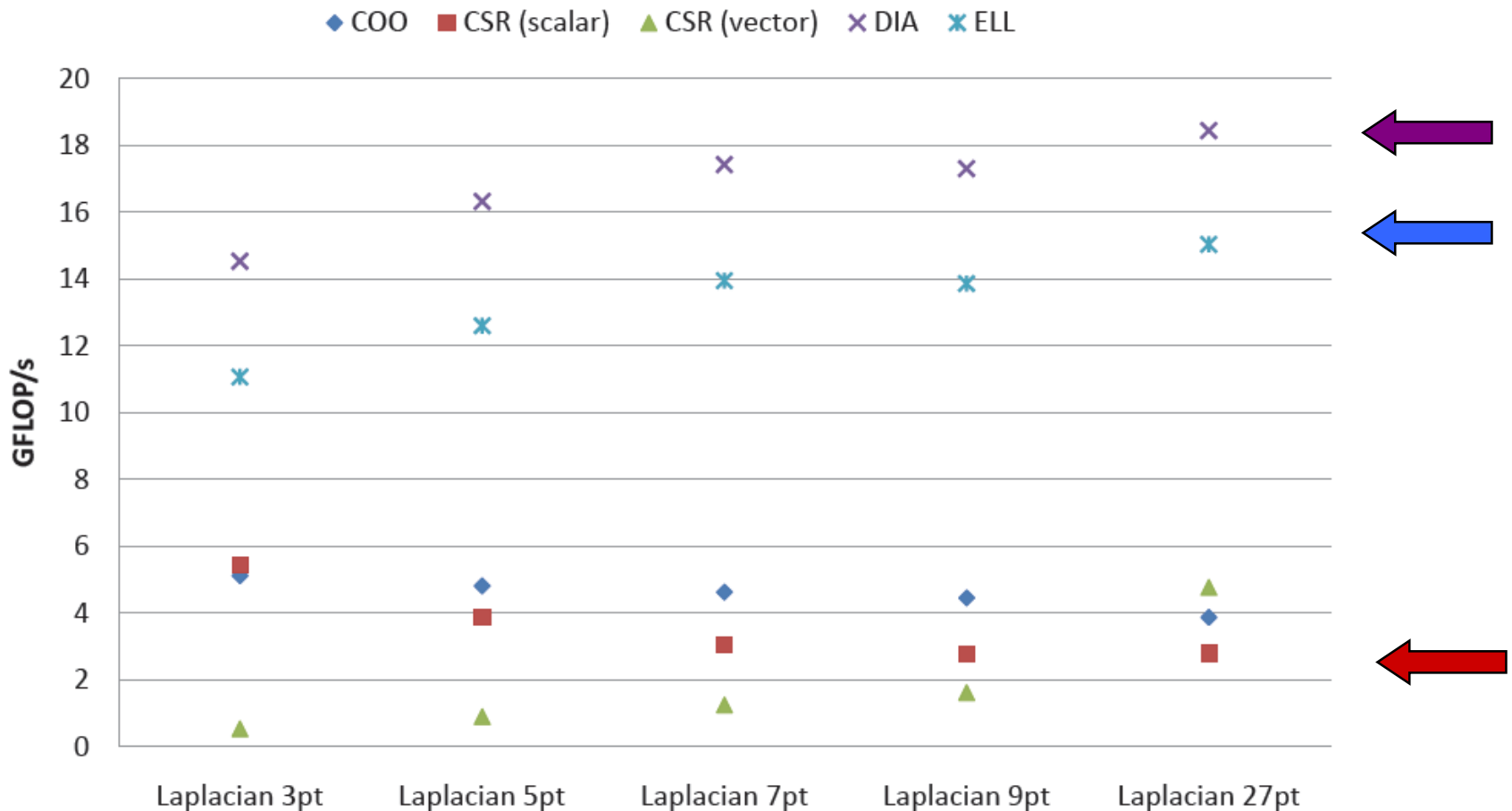


chart from [Bell and Garland SC 2009]

Basic Solvers

Linear equation system after discretization

$$A\vec{U} = \vec{R}$$

Iterative solvers are applied to approximate the solution

$$\vec{X}^{l+1} = F(\vec{X}^l), \quad \vec{X}^0 = \vec{R}$$

Jacobi-Solver

$$F(\vec{X}) = \vec{X} + D^{-1}(\vec{R} - A\vec{X}), \quad D := \text{diag}(A)$$

CG-Solver

$$\vec{r}^l = -(A^k \vec{X}^l - \vec{R})$$

$$\vec{p}^l = \vec{r}^l + \frac{\vec{r}^l \cdot \vec{r}^l}{\vec{r}^{l-1} \cdot \vec{r}^{l-1}} \vec{p}^{l-1}$$

$$F(\vec{X}^l) = \vec{X}^l + \frac{\vec{r}^l \cdot \vec{p}^l}{A^k \vec{p}^l \cdot \vec{p}^l} \vec{p}^l$$

Multigrid Idea

Linear equation system after discretization

$$A\vec{U} = \vec{R}$$

Observation: Basic solvers quickly reduce the high frequency error components, but struggle with low frequencies

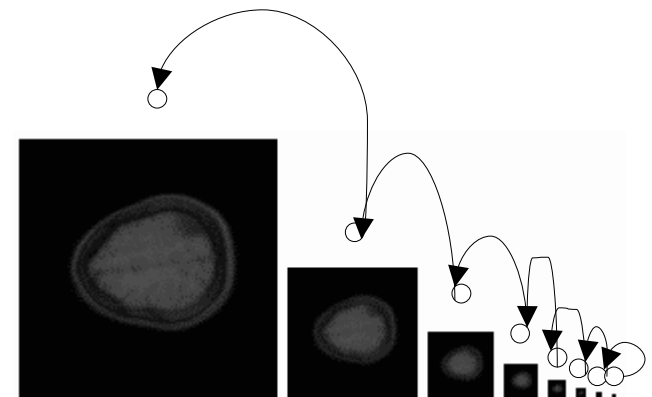
Idea: Solve the system on a pyramid of grids, thus dealing with different frequencies one after another

$$\begin{aligned} \mathbf{d}_k &= \mathbf{b} - \mathbf{A}\mathbf{x}_k \\ \mathbf{A}\mathbf{c}_k &= \mathbf{d}_k \\ \mathbf{x}_{k+1} &= \mathbf{x}_k + \mathbf{c}_k \end{aligned}$$

Fine grid

Coarse grid

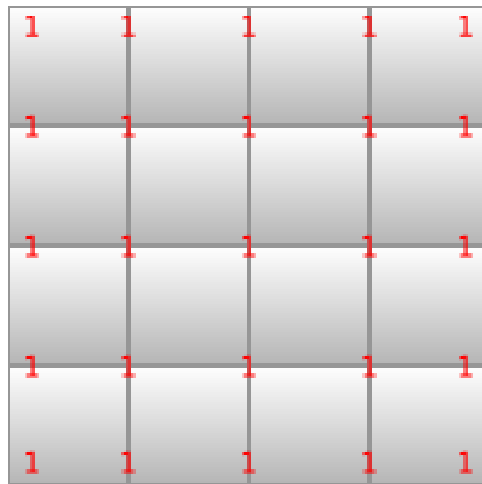
Back on **fine** grid



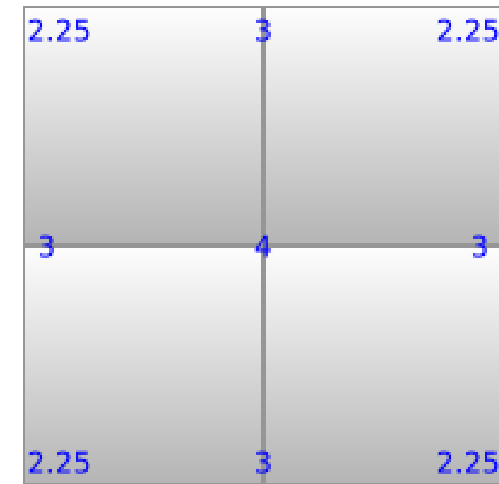
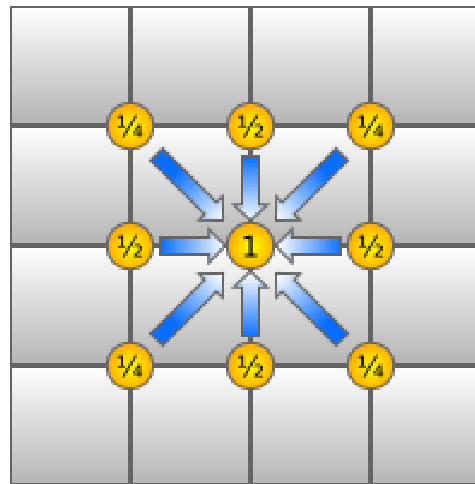
Multigrid Transfers

- **Restriction**

- Interpolate values from fine into coarse array
- Local neighborhood weighted gather on both CPU and GPU



fine

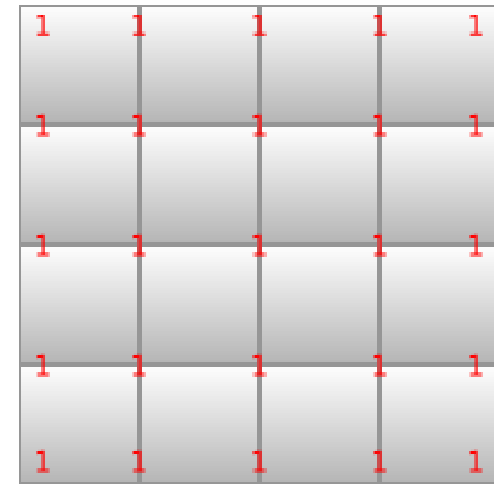
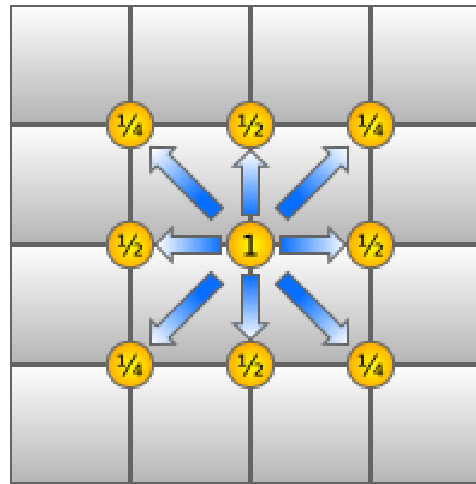
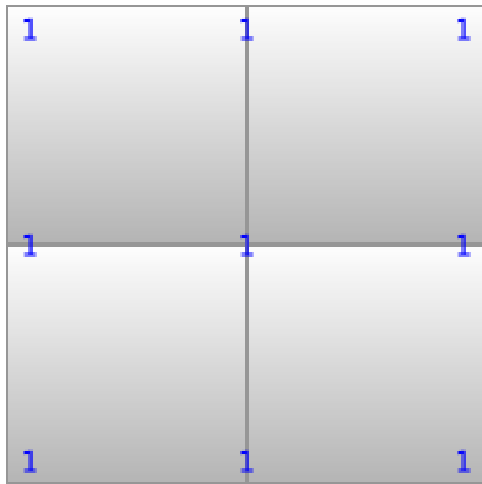


**coarse
result**

Multigrid Transfers

- **Prolongation**

- Scatter values from fine to coarse with weighting stencil
- Typical CPU implementation: loop over coarse array with stride-2 daxpy's



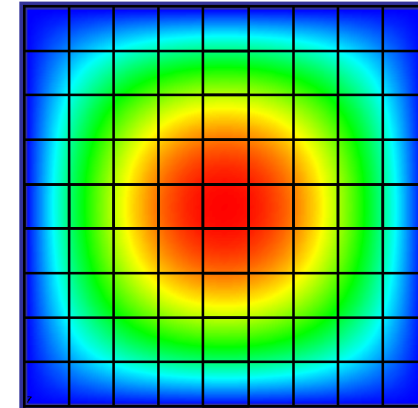
Overview

- Types of Parallel Data Flow
- Parallel Prefix or Scan
- Precision and Accuracy
- Efficient PDE Solvers
- **Mixed Precision Refinement**

PDE Example: Poisson Problem

- - $\Delta u = f$
- Unit square $[0,1]^2$
- Bilinear conforming FEs (Q1)
- Regular quadrilateral grid
- Zero Dirichlet BCs
- Analytic test function $x(1-x)y(1-y)$

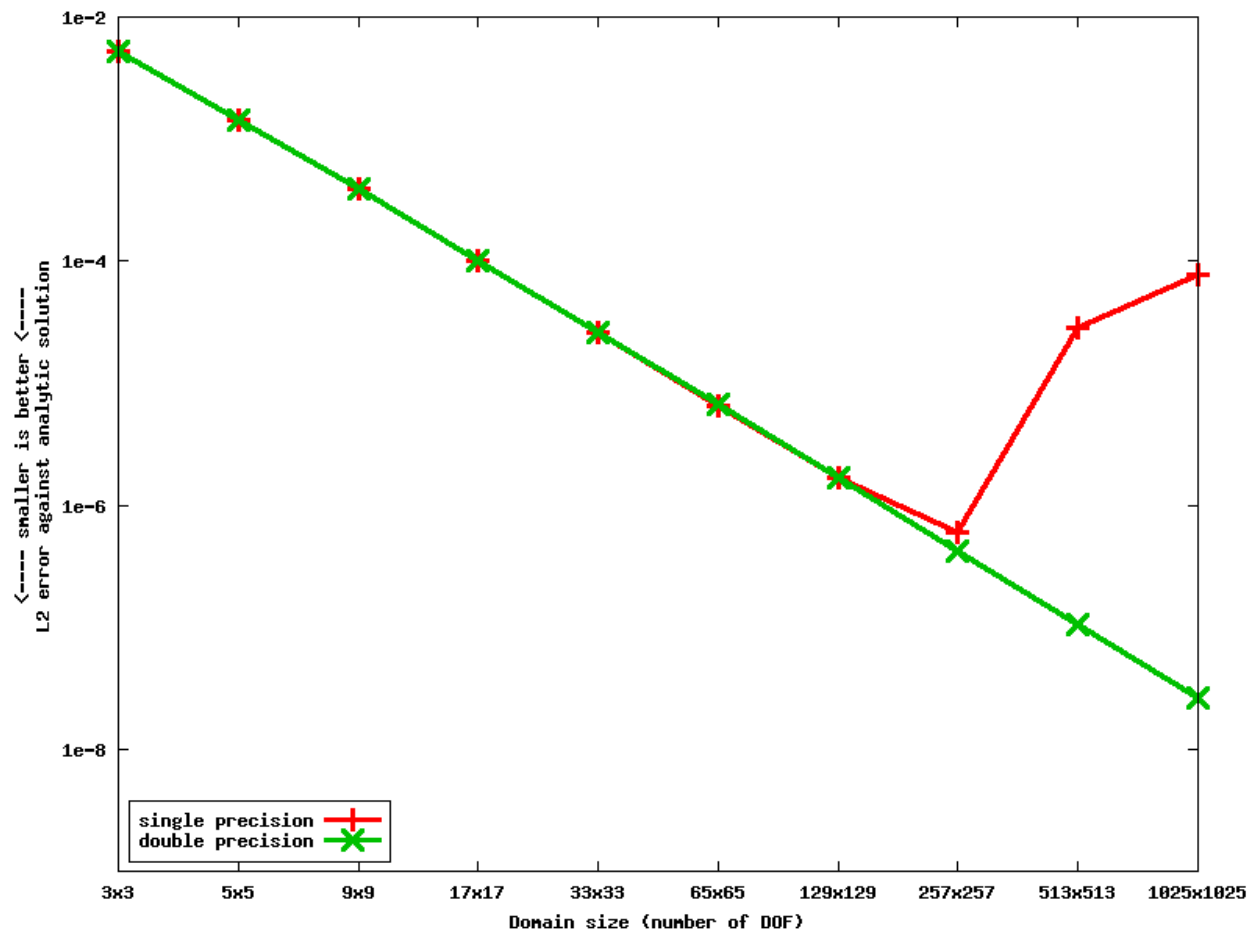
- Solved with multigrid until norms of residuals ***indicate*** convergence



PDE Example: Poisson Problem

- FEM theory: pure discretization error
- Expected error reduction of 4 (i.e. h^2) in each level

← Smaller is better ←



Mixed Precision Iterative Refinement $Ax=b$

- Exploit the **speed** of low precision and obtain a result of high **accuracy**

$$d_k = b - Ax_k$$

$$Ac_k = d_k$$

$$x_{k+1} = x_k + c_k$$

$$k = k + 1$$

Compute in high precision (cheap)

Solve in low precision (fast)

Correct in high precision (cheap)

Iterate until convergence in high precision

- **Low precision solution is used as a pre-conditioner in a high precision iterative method**
 - A is small and dense: Solve $Ac_k = d_k$ **directly**
 - A is large and sparse: Solve (approximately) $Ac_k = d_k$ with an **iterative** method itself

Direct Scheme for Small, Dense A

- **Algorithm**

- Compute $PA=LU$ once in single precision
- Use LU decomposition to solve $Ly=Pd_k, Uc_k=y$ in each step

- **Main reasons for speedup**

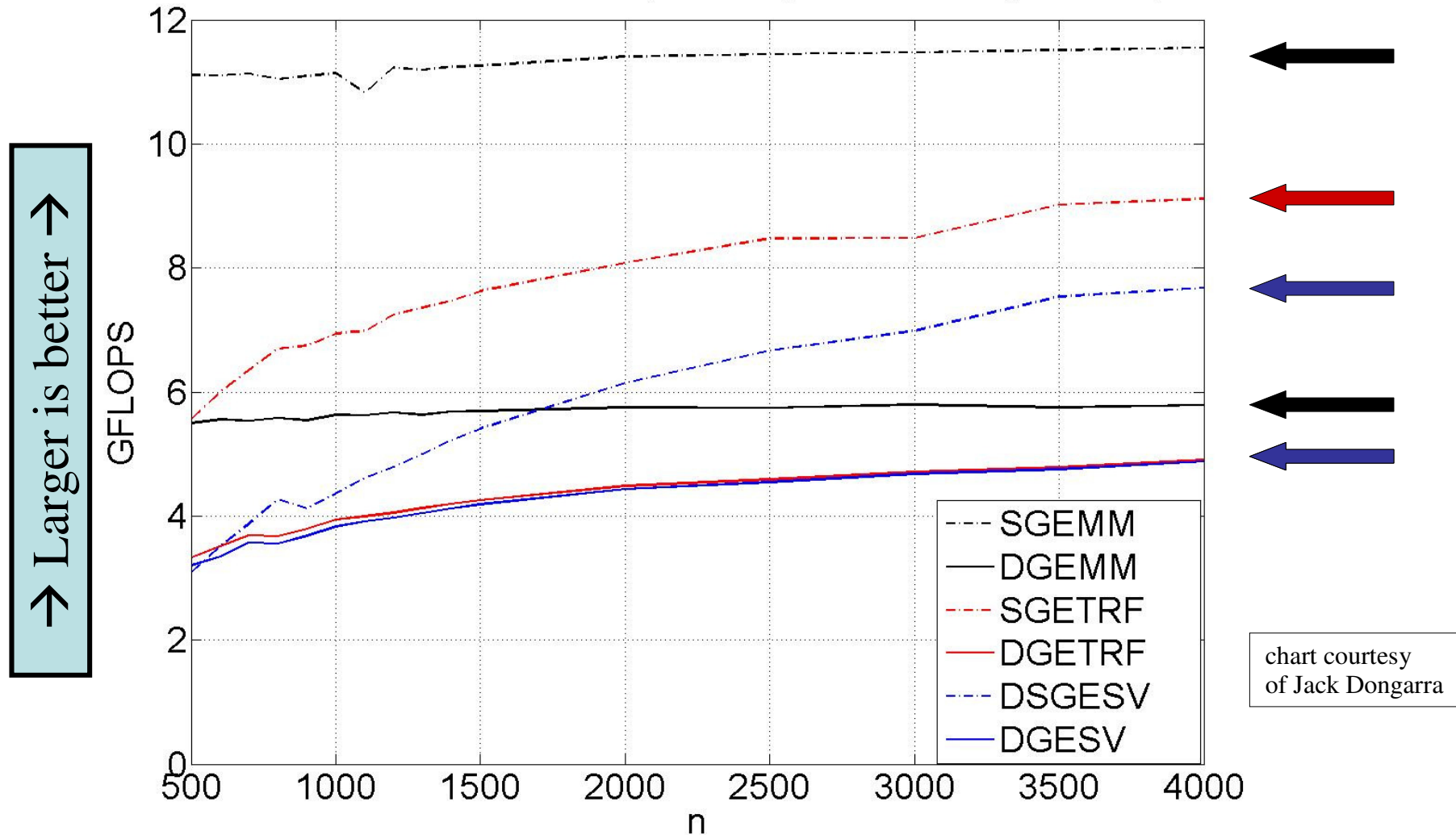
- Computation of LU decomposition is $O(n^3)$
- Computation of LU is much faster in single than in double
- Solution using LU for several RHS is only $O(n^2)$

- **Upper bound for iteration count**

- $\text{ceil}(t_d/(t_s-K))$, where K, t_d, t_s are \log_{10} of matrix condition and double and single precision (e.g. t_d approx 16)

CPU SSE Results: LU Solver

Intel Pentium IV Prescott (3.4GHz), Goto BLAS (1 thread)



Iterative Scheme for Large, Sparse A

- **Algorithm**

- Inner solver: **Conjugate Gradients, Multigrid**
- Correction loop can run on **CPU** or on **GPU** (old GPUs: emulated precision; new GPUs: true double precision)
- Terminate inner solver after **fixed number** of iterations, **fixed error reduction** or convergence

- **Main reason for speedup**

- Inner solver on the GPU runs almost at **peak bandwidth**

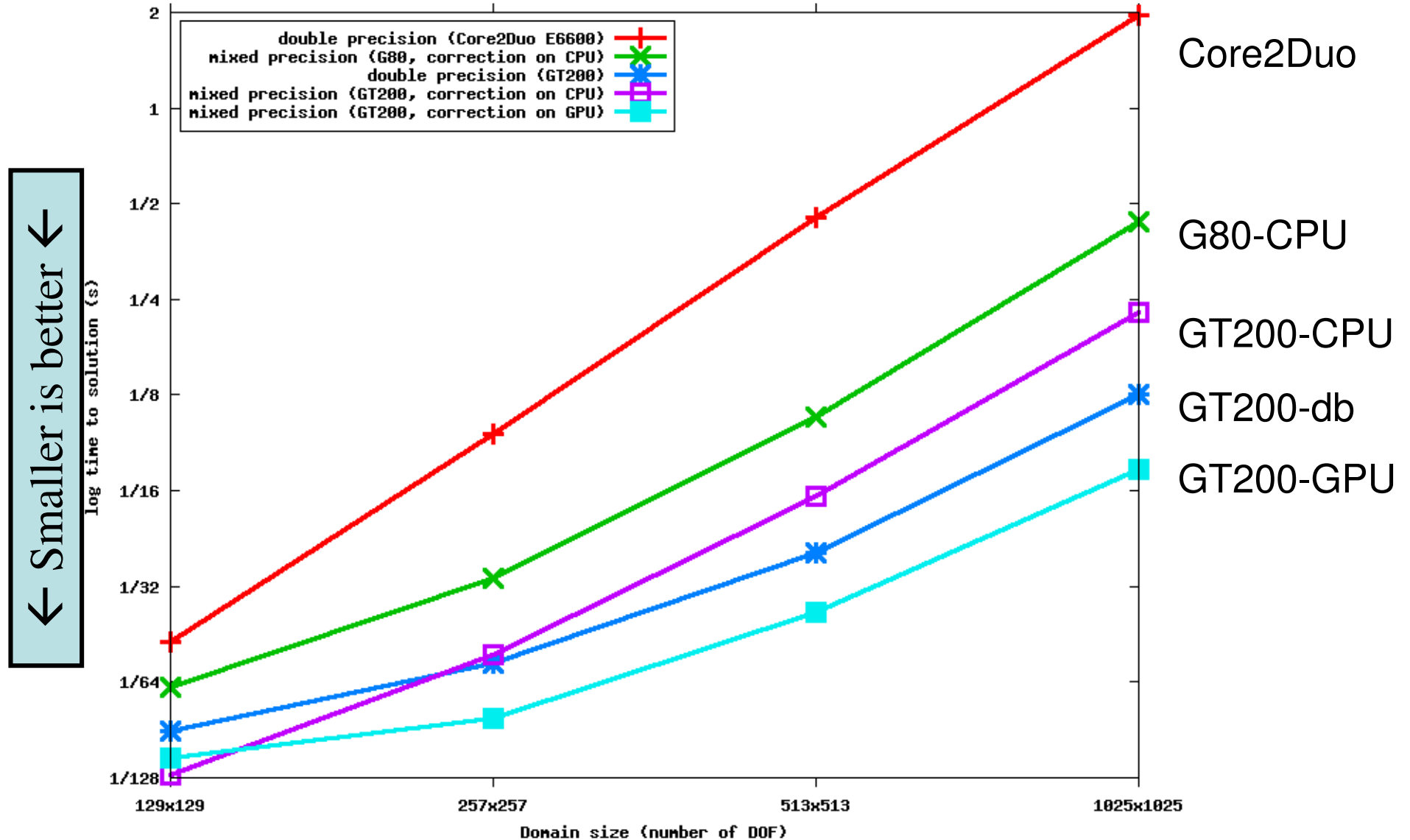
- **Applicability**

- Works even for very **ill-conditioned** matrices

GPU Performance Results

- **Test problem**
 - Poisson on unit square
 - Multigrid solver
 - $N=33^2$ to $N=1025^2$ DOF (=mesh points for Q1 FE)
- **Solver combination parameter space**
 - CPU implementation (Core2Duo E6600, SSE-optimized, double)
 - CUDA implementation (GeForce 8800 GTX and GeForce GTX 280)
 - Mixed precision, correction on CPU (G80 and GT200)
 - Native double precision (GT200 only)
 - Mixed precision, correction on GPU (GT200 only)

GPU Performance Results: CUDA



Conclusions

- **Scientific computing on GPUs is about identifying independent work and preserving data locality**
- **Parallel prefix (scan) enables the parallelization of many seemingly inherently sequential algorithms**
- **Precision \neq accuracy! Mixed precision methods give rise to very efficient PDE solvers**