technische universität dortmund

NVIDIA

ARCS 2008

Data parallel algorithms, algorithmic building blocks, precision vs. accuracy

Robert Strzodka

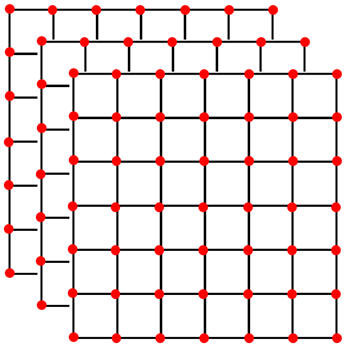
ARCS 2008 - Architecture of Computing Systems
GPGPU and CUDA Tutorials
Dresden, Germany, February 25 2008

Overview

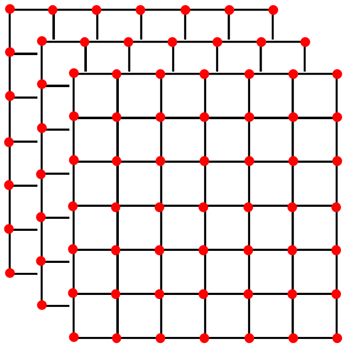
- Parallel Processing on GPUs
- Types of Parallel Data Flow
- Parallel Prefix or Scan
- Precision and Accuracy

The GPU is a Fast, Parallel Array Processor

Input Arrays:
1D, 3D,
2D (typical)

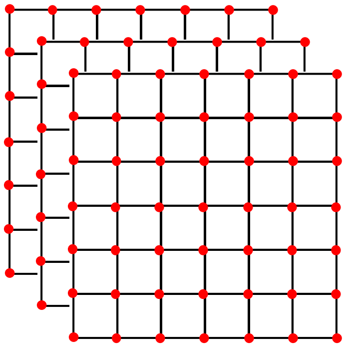


Output Arrays:
1D, 3D (slice),
2D (typical)

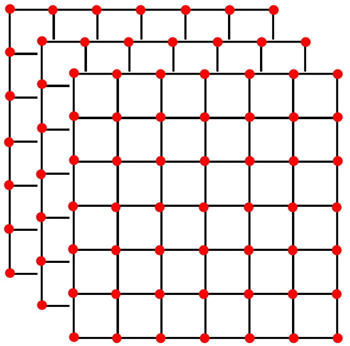


The GPU is a Fast, Parallel Array Processor

<u>Input Arrays</u>:
1D, 3D,
<u>2D</u> (typical)



<u>Output Arrays</u>:
1D, 3D (slice),
<u>2D</u> (typical)

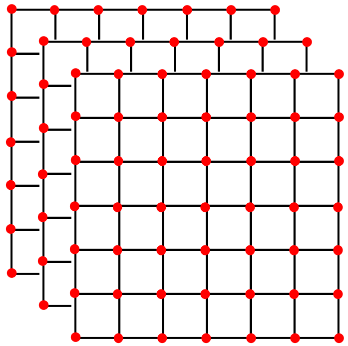


<u>Vertex Processor (VP)</u>

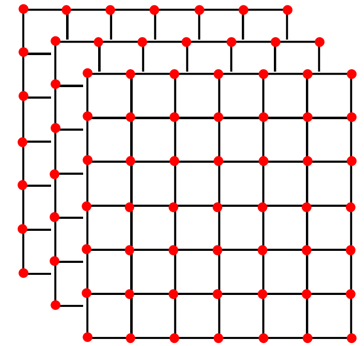
<u>Kernel</u> changes index **<u>regions</u>** of output arrays

The GPU is a Fast, Parallel Array Processor

Input Arrays:
1D, 3D,
2D (typical)



Output Arrays:
1D, 3D (slice),
2D (typical)



Vertex Processor (VP)

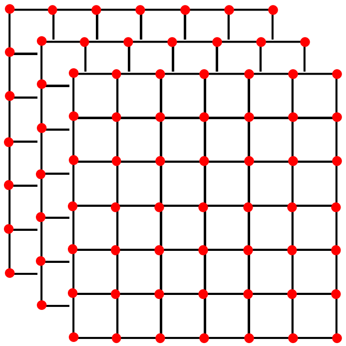
Kernel changes index **regions** of output arrays

Rasterizer

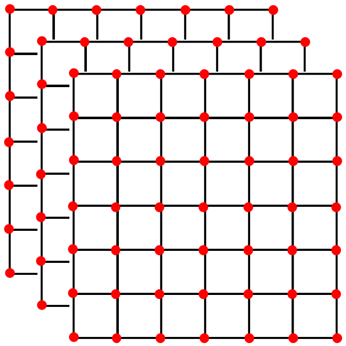
Creates data streams from index **regions**

The GPU is a Fast, Parallel Array Processor

Input Arrays:
1D, 3D,
2D (typical)



Output Arrays:
1D, 3D (slice),
2D (typical)



Vertex Processor (VP)

Kernel changes index **regions** of output arrays

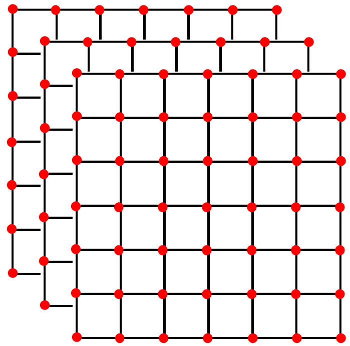
Rasterizer

Creates data streams from index **regions**

Stream of array elements, order unknown

The GPU is a Fast, Parallel Array Processor

Input Arrays:
1D, 3D,
2D (typical)



Vertex Processor (VP)

Kernel changes index **regions** of output arrays

Rasterizer

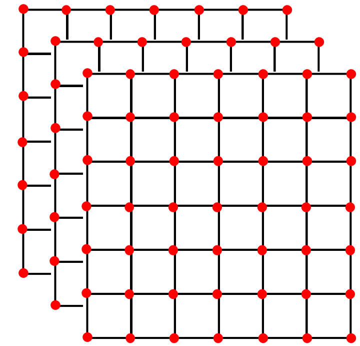
Creates data streams from index **regions**

Stream of array elements, order unknown

Fragment Processor (FP)

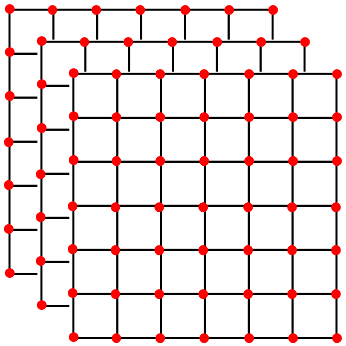
Kernel changes each datum independently, reads more input arrays

Output Arrays:
1D, 3D (slice),
2D (typical)

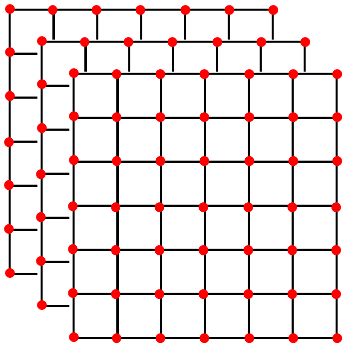


The GPU is a Fast, Parallel Array Processor

Input Arrays: 1D, 3D, **2D** (typical)



Output Arrays: 1D, 3D (slice), **2D** (typical)



Vertex Processor (VP)

Kernel changes index **regions** of output arrays

Rasterizer

Creates data streams from index **regions**

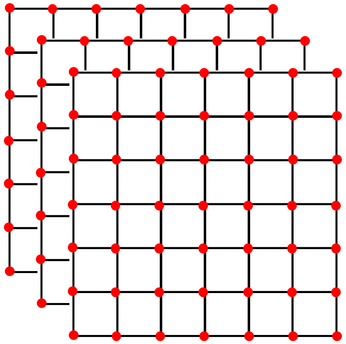
Stream of array elements, order unknown

Fragment Processor (FP)

Kernel changes each datum independently, reads more input arrays

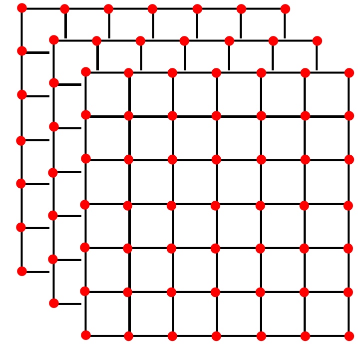
The GPU is a Fast, Highly Multi-Threaded Processor

Input Arrays:
nD



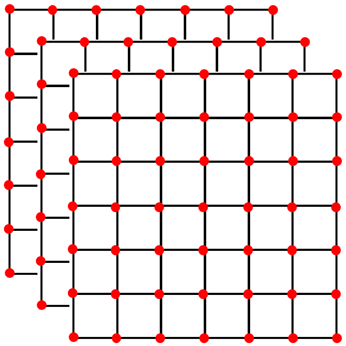
Start thousands of parallel threads in **groups of m, e.g. 32**

Output Arrays:
nD

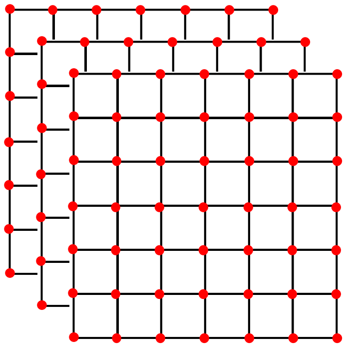


The GPU is a Fast, Highly Multi-Threaded Processor

Input Arrays:
nD



Output Arrays:
nD

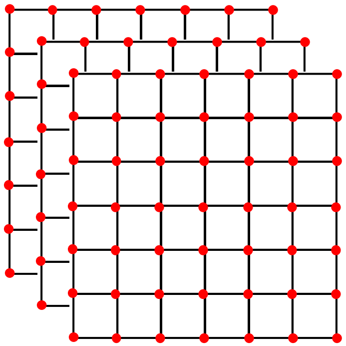


Start thousands of parallel threads in **groups of m, e.g. 32**

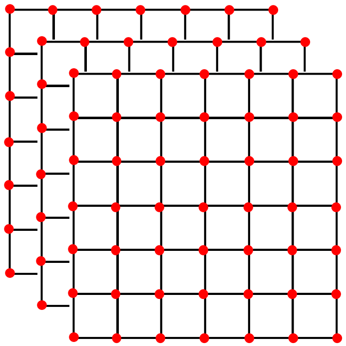
Each group operates in a **SIMD** fashion, with predication if necessary

The GPU is a Fast, Highly Multi-Threaded Processor

Input Arrays:
nD



Output Arrays:
nD



Start thousands of parallel threads in **groups of m, e.g. 32**

Each group operates in a **SIMD** fashion, with predication if necessary

In general all threads are independent but certain collections of groups may use **local memory to exchange** data

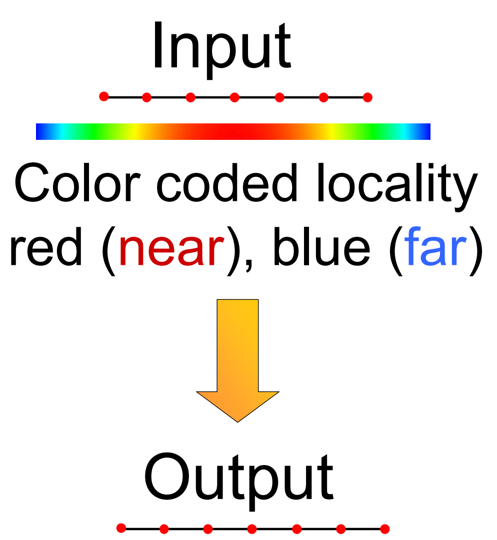
Native Memory Layout – Data Locality

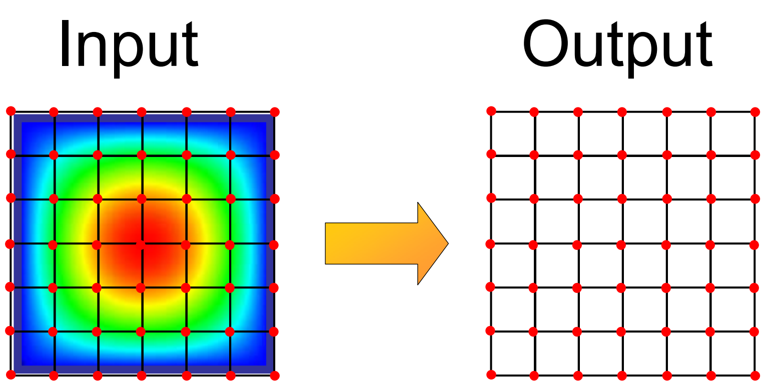
CPU

- 1D input
- 1D output
- Other dimensions with offsets

GPU

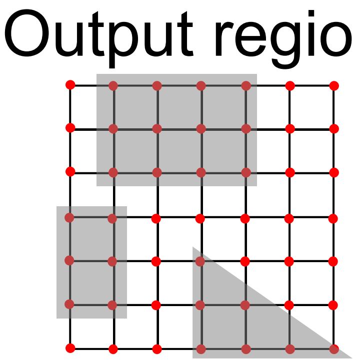
- 2D input
- 2D output
- Other dimensions with offsets

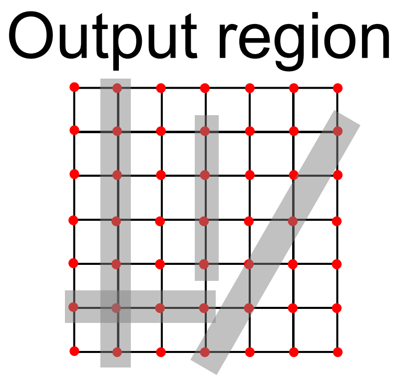


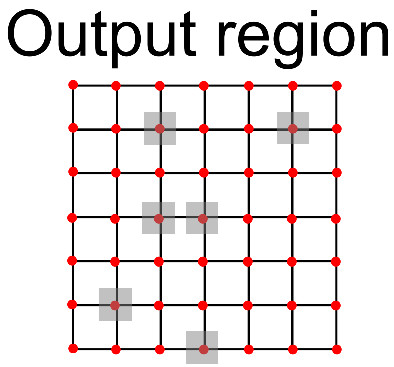


Primitive Index Regions in Output Arrays

- **Quads and Triangles**
  - Fastest option
- **Line segments**
  - Slower, try to pair lines to 2xh, wx2 quads
- **Point Clouds**
  - Slowest, try to gather points into larger forms

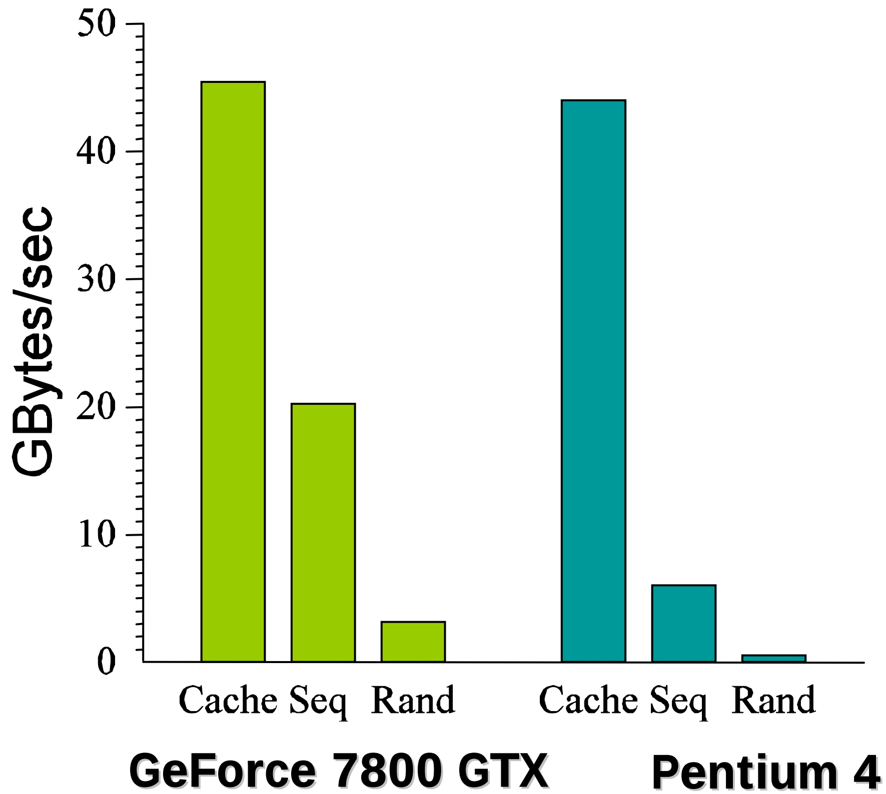






GPUs are Optimized for Local Data Access

Memory access types: **Cache**, **Seq**uential, **Rand**om



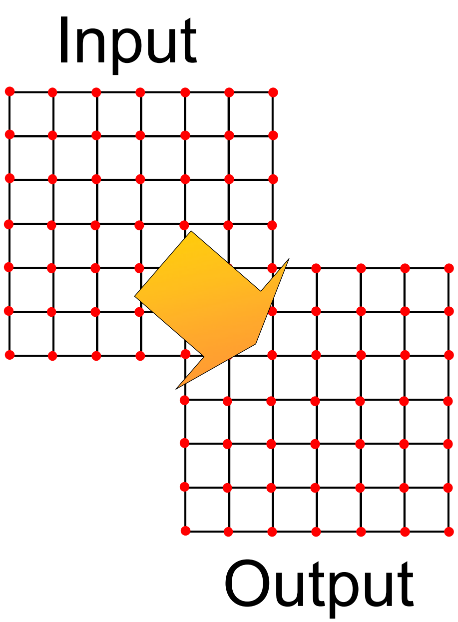
- CPU
  - **Large** cache
  - **Few** processing elements
  - Optimized for **spatial** and **temporal** data reuse
- GPU
  - **Small** cache
  - **Many** processing elements
  - Optimized for **sequential** (streaming) data access

chart courtesy of Ian Buck

Input and Output Arrays

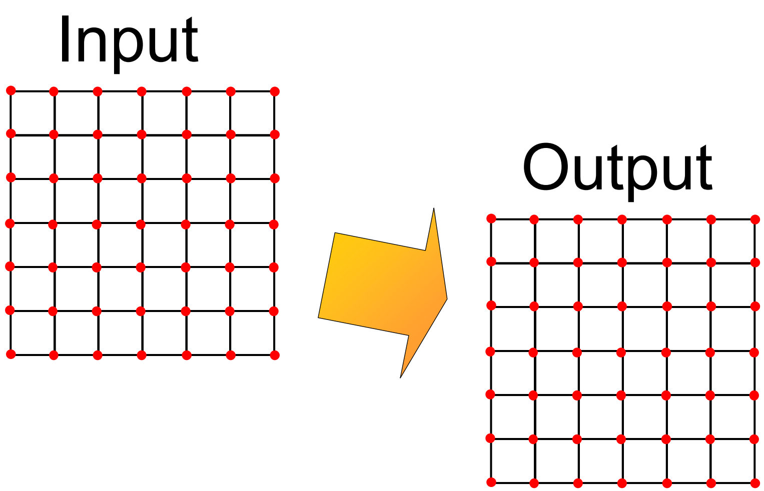
CPU

- Input and output arrays **may overlap**



GPU

- Input and output arrays **must not overlap**



Configuration Overhead

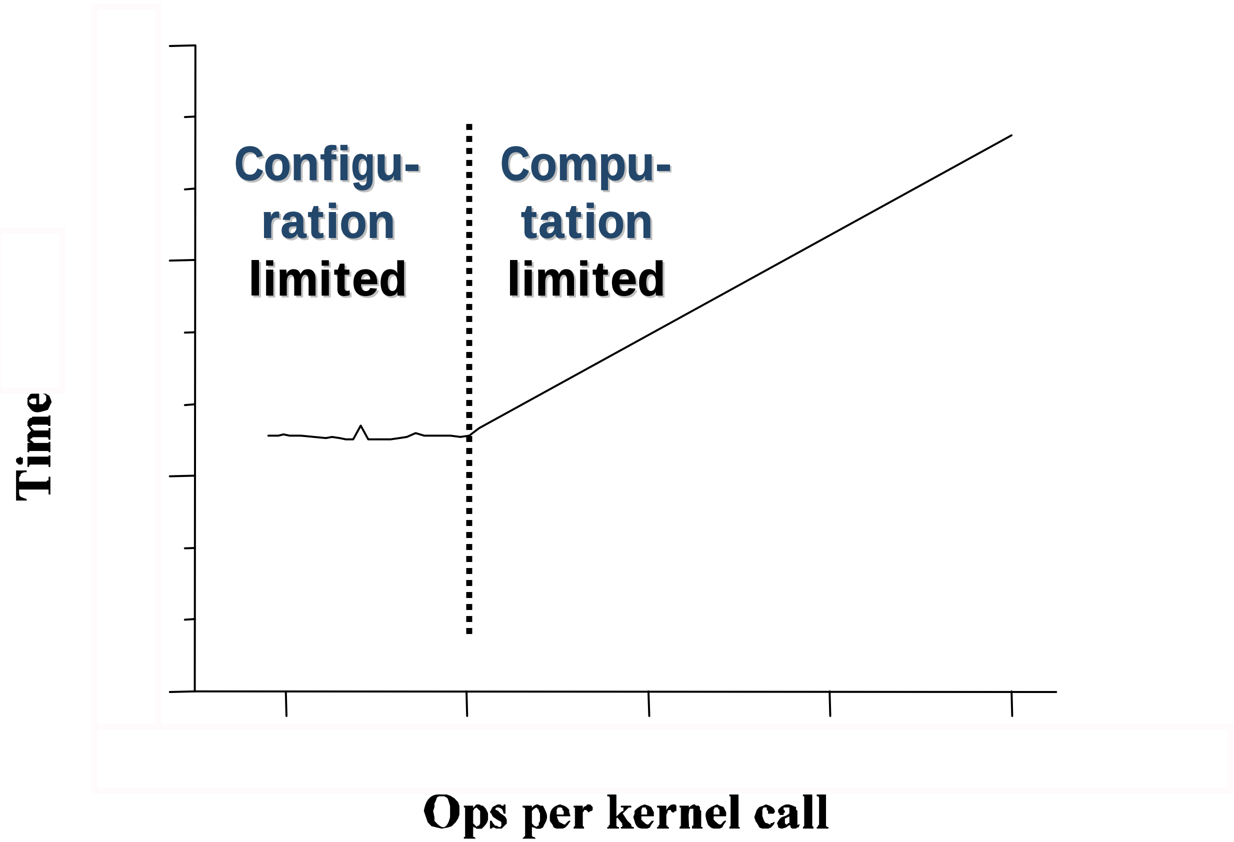


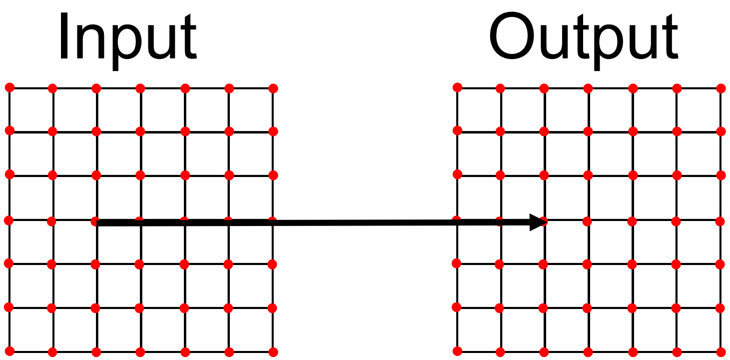
chart courtesy of Ian Buck

Overview

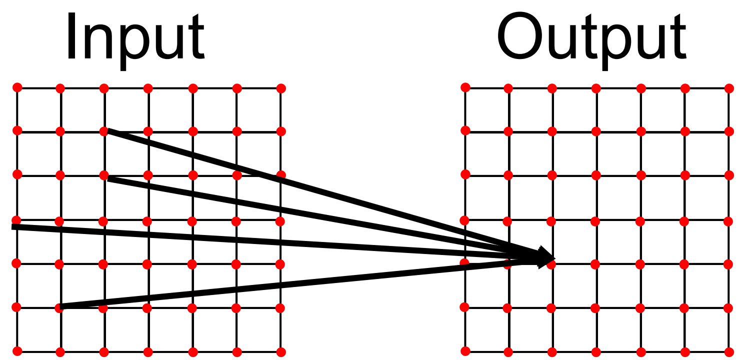
- Parallel Processing on GPUs
- Types of Parallel Data Flow
- Parallel Prefix or Scan
- Precision and Accuracy

Parallel Data-Flow: Map, Gather and Scatter

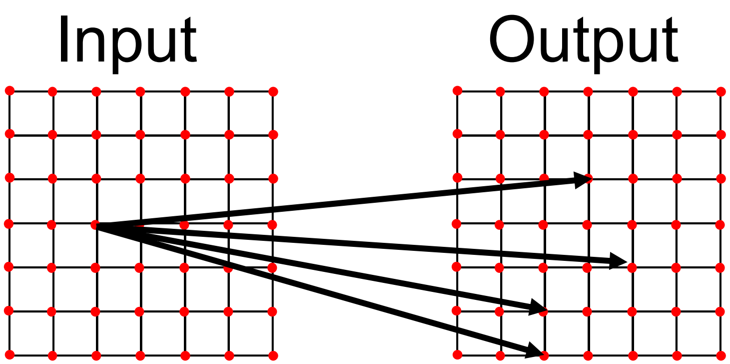
Map: x= f(a)



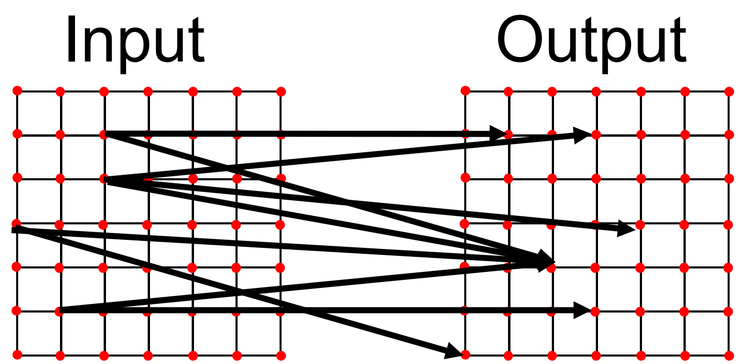
Gather: x= f(a(1,2), a(3,5), …)



Scatter: x(2,3)= f(a), x(6,7)= g(a), …



General: x(2,3)= f(a(1,2), a(3,5), …),
x(6,7)= f(a(6,2), a(7,5), …),



Performance of Gather and Scatter

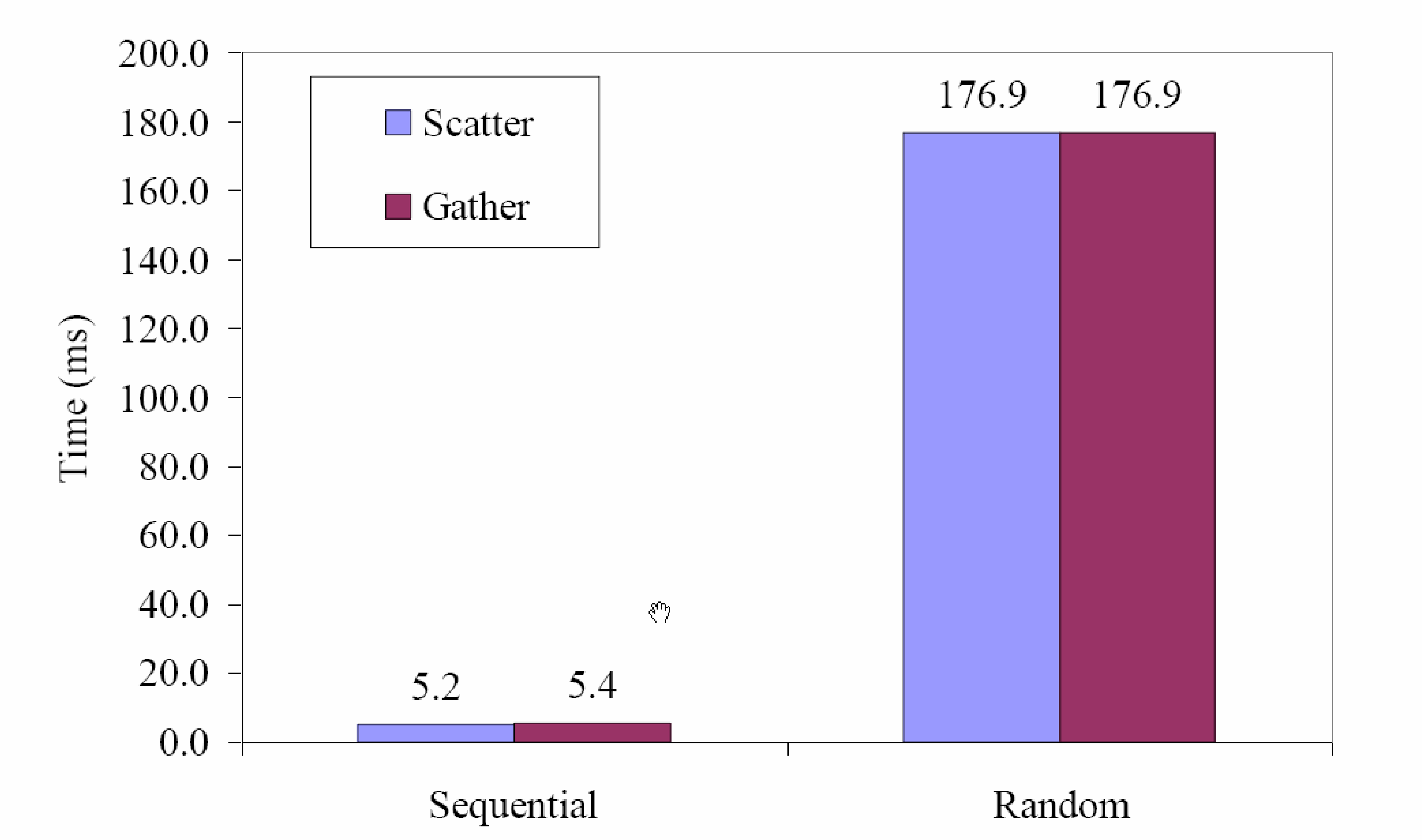


chart courtesy of
Naga Govindaraju

Parallel Reduction, e.g. Maximum of an Array

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

input

Parallel Reduction, e.g. Maximum of an Array

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

N/2 x N/2 output

Parallel Reduction, e.g. Maximum of an Array

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

gather 2x2 regions for each output

Parallel Reduction, e.g. Maximum of an Array

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

47	57	15	17
38	64	68	35
46	49	61	52
71	67	69	70

first output

Parallel Reduction, e.g. Maximum of an Array

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

47	57	15	17
38	64	68	35
46	49	61	52
71	67	69	70

maximum of 2x2 region

Parallel Reduction, e.g. Maximum of an Array

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

47	57	15	17
38	64	68	35
46	49	61	52
71	67	69	70

64	68
71	70

intermediates

Parallel Reduction, e.g. Maximum of an Array

input

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

intermediates

47	57	15	17
38	64	68	35
46	49	61	52
71	67	69	70

64	68
71	70

result

71

Parallel Reduction, e.g. Maximum of an Array

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

47	57	15	17
38	64	68	35
46	49	61	52
71	67	69	70

64	68
71	70

71

- **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/**

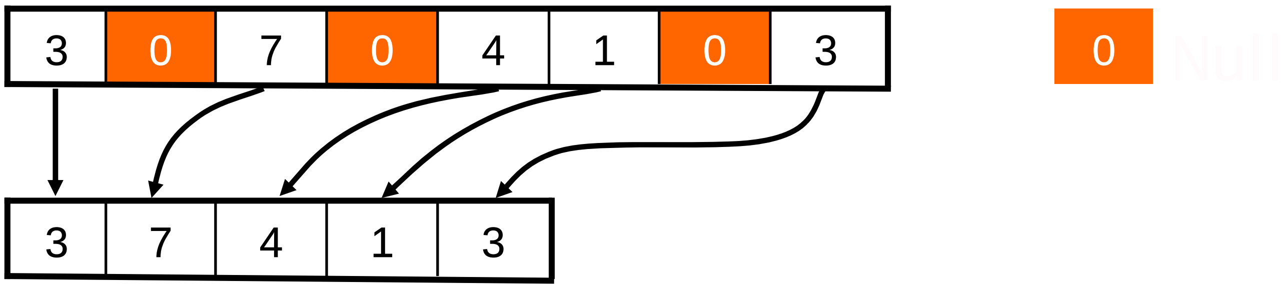
Overview

- Parallel Processing on GPUs
- Types of Parallel Data Flow
- Parallel Prefix or Scan
- Precision and Accuracy

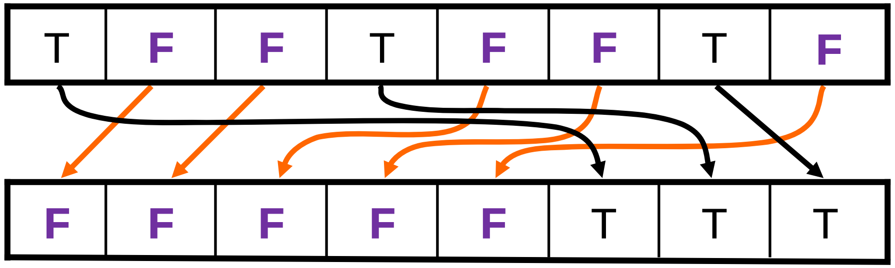
slides courtesy of
Shubho Sengupta

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

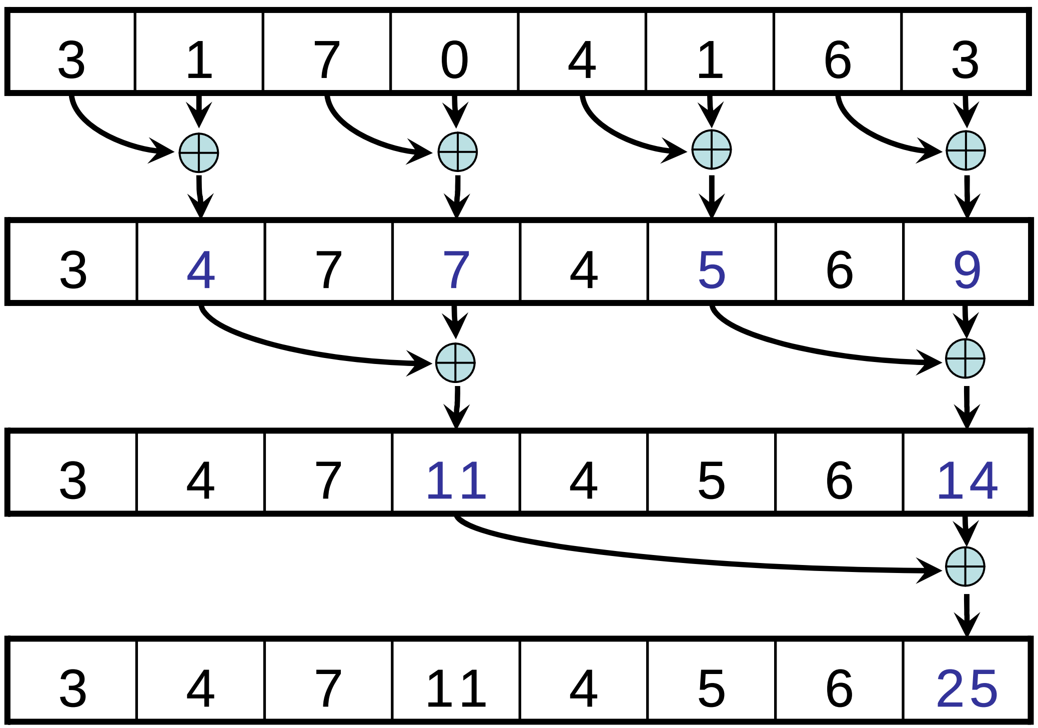
Scan – the past

- 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

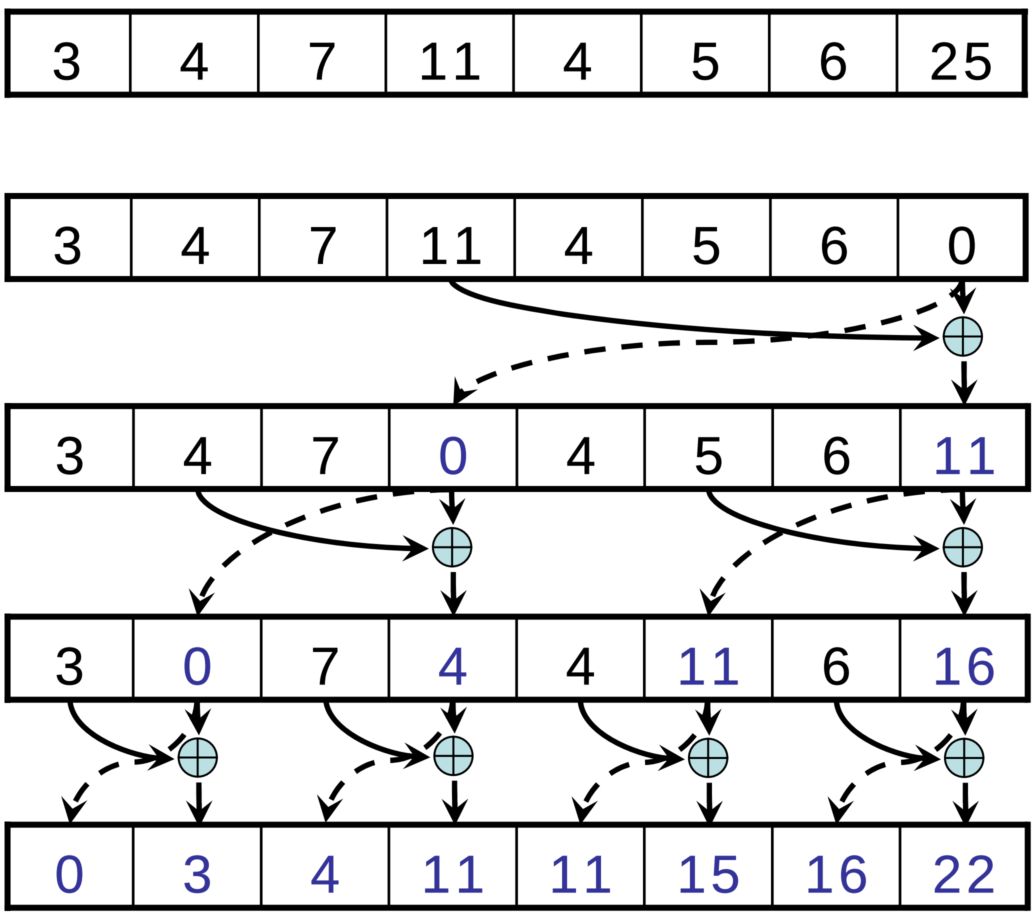
- **First GPU implementation by Daniel Horn (2004), $O(n \log n)$**
- **Subsequent GPU implementations by**
  - Hensley (2005) $O(n \log n)$, Sengupta (2006) $O(n)$, Greß (2006) $O(n)$ 2D
- **NVIDIA CUDA implementation by Mark Harris (2007), $O(n)$, space efficient**

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

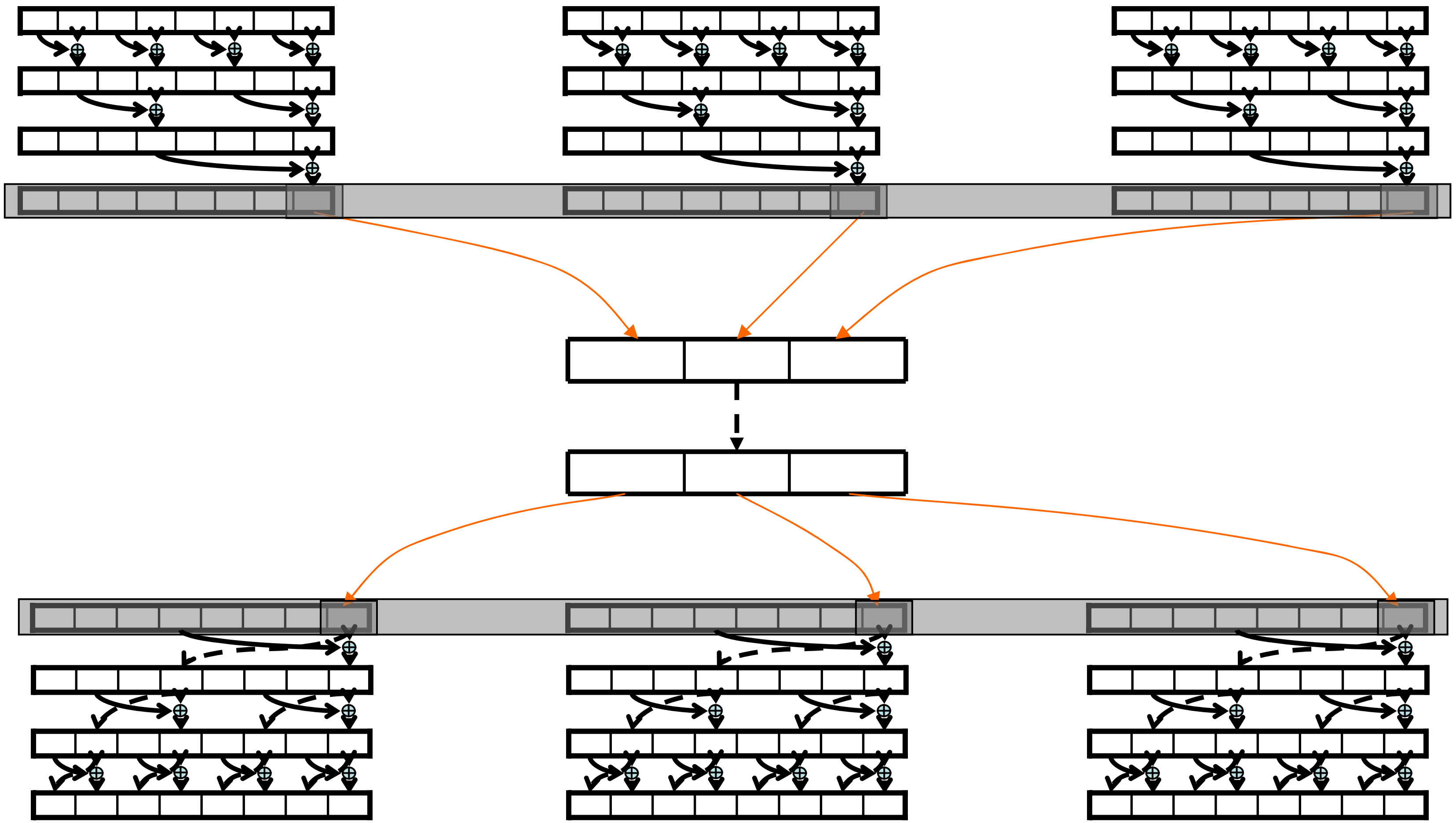
- Scan within each segment in parallel
- Output

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

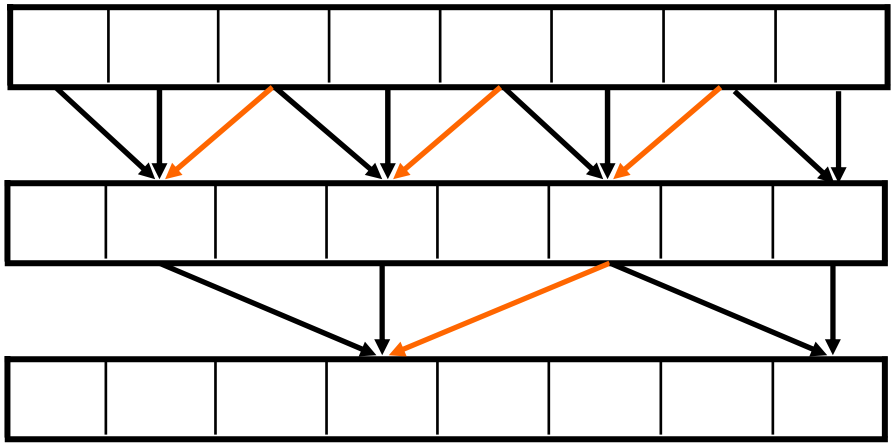


Segmented Scan – Advantages

- 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

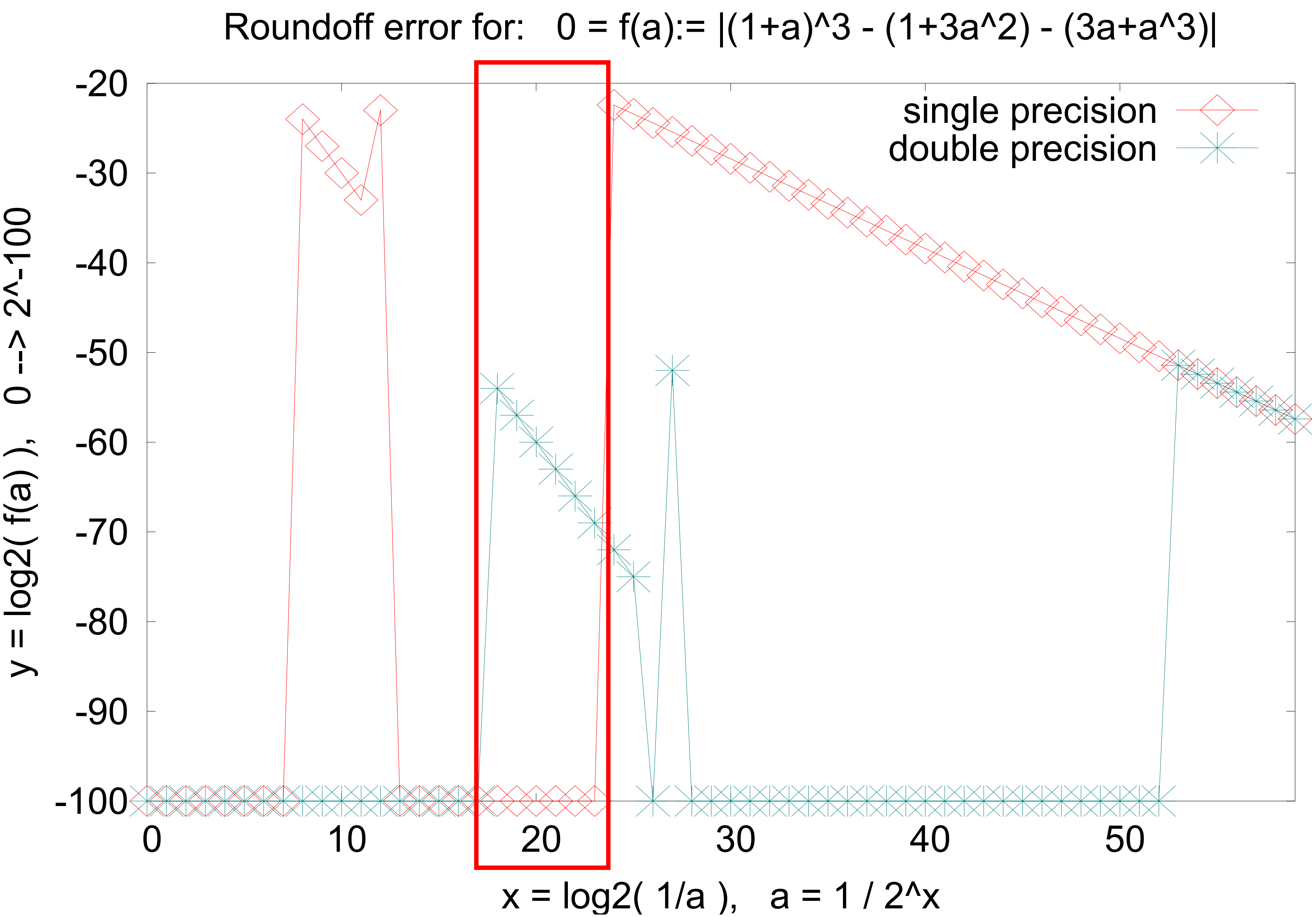


Overview

- Parallel Processing on GPUs
- Types of Parallel Data Flow
- Parallel Prefix or Scan
- Precision and Accuracy

The Erratic Roundoff Error

↓ Smaller is better ↓



Precision and Accuracy

- There is no monotonic relation between the computational precision and the accuracy of the final result.
- Increasing precision can decrease accuracy !
- The increase or decrease of precision in different parts of a computation can have very different impact on the accuracy.
- The above can be exploited to significantly reduce the precision in parts of a computation without a loss in accuracy.
- We obtain a mixed precision method.

Quantization - Preserving Accuracy

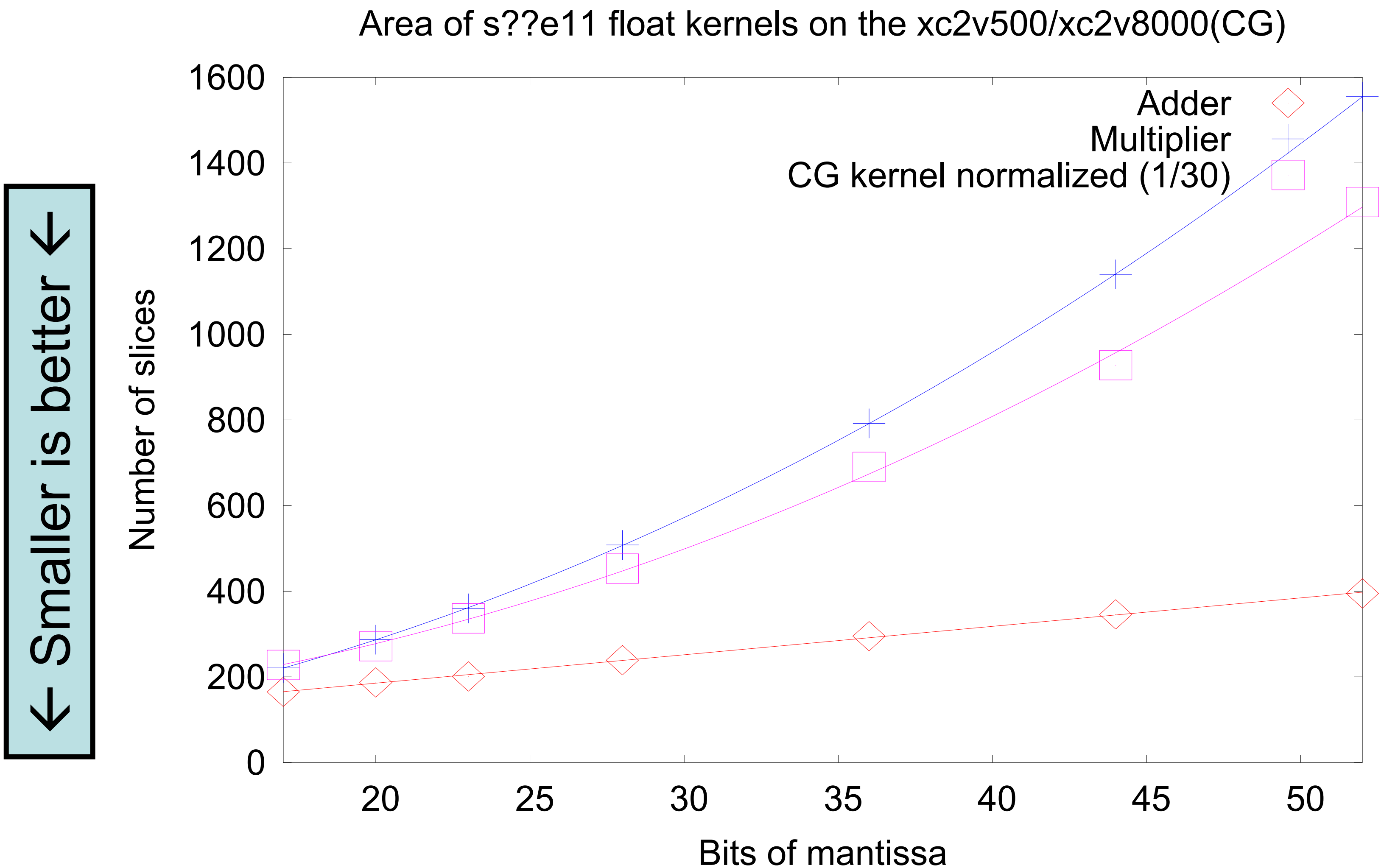
- Watch out for **cancellation**
  - $a \approx b$, $r = c*a-c*b$
  - $r = c(a-b)$
- Maximize operations on the **same scale**
  - $a \in [0,1]$, $b,c \in [10^{-3}, 10^{-4}]$, $r = a+b+c$
  - $r = a+(b+c)$
- Make **implicit relations** between constants explicit
  - $a_i=0.01$, $i=0..99$ $r = \Sigma_{i<100} a_i \neq 1$
  - $a_{99}=1-(\Sigma_{i<99} a_i)$, $r = \Sigma_{i<100} a_i = 1$
- Use **symmetric** intervals for multiplication
  - $a \sim [-1, 1]$, $r = 0.1134*(a+1)$
  - $r = 0.1134a+0.1134$
- Minimize the **number of multiplications**
  - $r = 0.25a + 0.1b + 0.15c$
  - $r = 0.1(a+b)+0.15(a+c)$

Resources for Signed Integer Operations

Operation	Area	Latency
min(r,0) max(r,0)	**b+1**	**2**
add(r_1,r_2) sub(r_1,r_2)	**2b**	**b**
add(r_1,r_2,r_3)$\rightarrow$add(r_4,r_5)	**2b**	**1**
mult(r_1,r_2) sqr(r)	**b(b-2)**	**b ld(b)**
sqrt(r)	**2c(c-5)**	**c(c+3)**

b: bitlength of argument, c: bitlength of result

FPGA Results: Conjugate Gradient (CG)



[Göddeke et al. *Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations*, IJPEDS 2007]

High Precision Emulation

- Given a ***m* x *m*** bit unsigned integer multiplier we want to build a ***n* x *n*** multiplier with a ***n*=*k**m*** bit result

$$\sum_{i=1}^{k} 2^{-im} a_i \cdot \sum_{j=1}^{k} 2^{-jm} b_j = \sum_{\substack{i,j=1 \\ i+j \le k+1}}^{k} 2^{-(i+j)m} a_i b_j + \sum_{\substack{i,j=1 \\ i+j > k+1}}^{k} 2^{-(i+j)m} a_i b_j$$

- The evaluation of the first sum requires $k(k+1)/2$ multiplications, the evaluation of the second depends on the rounding mode

- For floating point numbers additional operations are necessary because of the mantissa/exponent distinction

- A double emulation with two aligned s23e8 single floats is less complex than an exact s52e11 double emulation, achieves a s46e8 precision and still requires 10-20 single float operations

Precision – Performance Rough Estimates

- **Reconfigurable device, e.g. FPGA**
  - 2x float add ≈ 1x double add
  - 4x float mul ≈ 1x double mul

- **Hardware emulation (compute area limited), e.g. GPU**
  - 2x float add ≈ 1x double add
  - 5x float mul ≈ 1x double mul

- **Hardware emulation (data path limited), e.g. CPU**
  - 2x float add ≈ 1x double add
  - 2x float mul ≈ 1x double mul

- **Software emulation**
  - 10x float add ≈ 1x double add
  - 20x float mul ≈ 1x double mul

Mixed Precision Iterative Refinement Ax=b

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

$d_k = b - Ax_k$	**Compute** in high precision (cheap)
$Ac_k = d_k$	**Solve** in low precision (fast)
$x_{k+1} = x_k + c_k$	**Correct** in high precision (cheap)
$k = k+1$	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

CPU Results: LU Solver

→ Larger is better →

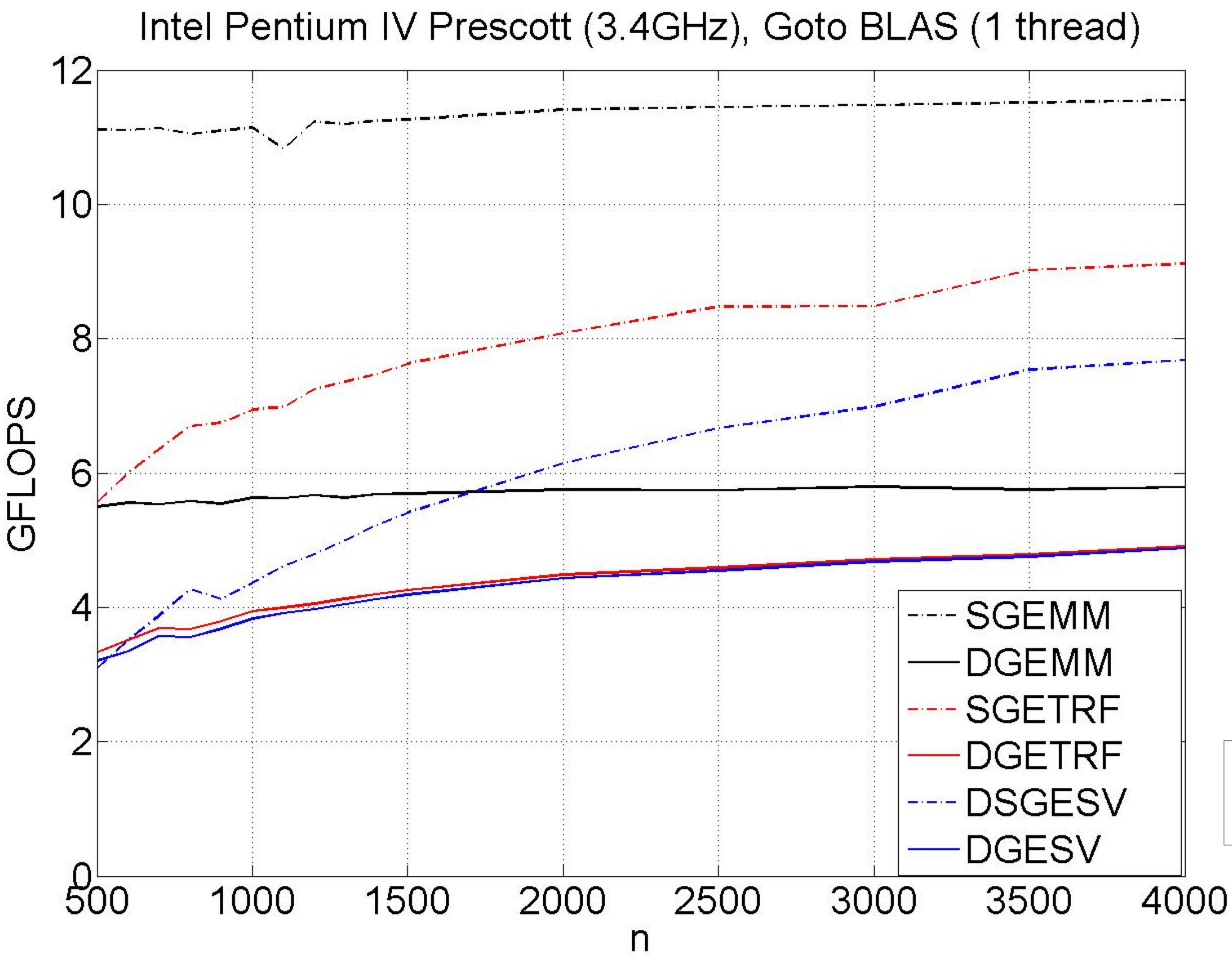
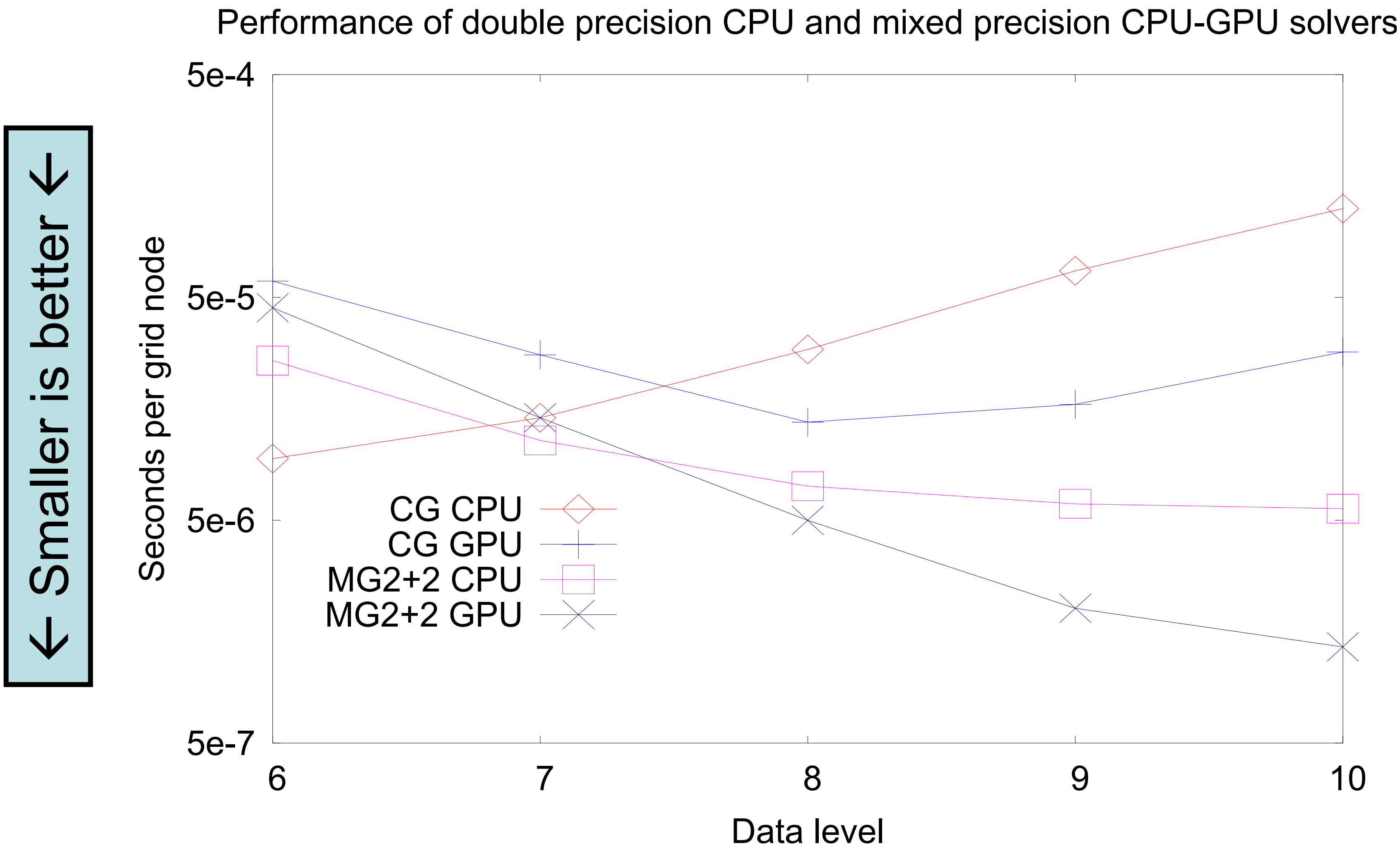


chart courtesy of Jack Dongarra

[Langou et al. *Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems),* SC 2006]

GPU Results: Conjugate Gradient (CG) and Multigrid (MG)



[Göddeke et al. *Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations*, IJPEDS 2007]

Conclusions

- Parallel Processing on GPUs is about identifying independent work and preserving data locality
- Map, gather, scatter are basic types of parallel data-flow.
- Parallel prefix (scan) enables the parallelization of many seemingly inherently sequential algorithms
- Precision ≠ accuracy! Mixed precision methods can reduce resource requirements quadratically.