Parallel Programming

Data-Parallel Primitives on the GPU

Gather and Scatter

`

Overview

- Data-Parallel Primitives
  - Map, Prefix Scan, Scatter, Gather, Split, Sort
  - Others: Reduce, Filter, Search…
- Optimizations on the GPU

Processing Large Data Sets

```
//sequential
   for (i = 0; i < N; i++)
            h_C[i] = h_A[i] + h_B[i];

//data-parallel
__global__ void VecAdd(int* A, int* B, int* C)
{
   int i = blockDim.x * blockIdx.x + threadIdx.x;
   C[i] = A[i] + B[i];
}
```

Map and Prefix Scan

Primitive: Map
Input: R_{in} [1, …, n], a map function fcn.
Output: R_{out}[1,…,n].
Function: $R_{out}[i]=fcn(R_{in}[i])$.

Primitive: Prefix Scan
Input: R_{in} [1, …, n], binary operator $\oplus$.
Output: R_{out}[1,…,n].
Function: $R_{out}[i]= \bigoplus_{j<i} R_{in}[j]$.

Scatter and Gather

Primitive: Scatter
Input: $R_{in}[1, \ldots, n]$, $L[1, \ldots, n]$.
Output: $R_{out}[1, \ldots, n]$.
Function: $R_{out}[L[i]]=R_{in}[i]$, $i=1, \ldots, n$.

Primitive: Gather
Input: $R_{in}[1, \ldots, n]$, $L[1, \ldots, n]$.
Output: $R_{out}[1, \ldots, n]$.
Function: $R_{out}[i]=R_{in}[L[i]]$, $i=1, \ldots, n$.

Split and Sort

Primitive: Split
Input: $R_{in}[1, \ldots, n]$, $func(R_{in}[i]) \in [1,\ldots,F]$, i=1, …, n.
Output: $R_{out}[1, \ldots, n]$.
Function: $\{R_{out}[i],\ i=1,\ldots,\ n\} = \{R_{in}[i],\ i=1,\ \ldots,\ n\}$ and $func(R_{out}[i]) \le func(R_{out}[j]), \forall i,j \in [1,..,n], i \le j$.

Primitive: Sort
Input: $R_{in}[1, \ldots, n]$.
Output: $R_{out}[1, \ldots, n]$.
Function: $\{R_{out}[i],\ i=1,\ldots,\ n\} = \{R_{in}[i],\ i=1,\ \ldots,\ n\}$ and $R_{out}[i] \le R_{out}[j], \forall i,j \in [1,..,n] \text{ and } i \le j$.

Map Example

```
// for all samples – all threads execute this code
neighbors[x][y] =
0.25f * (value[x-1][y]+
value[x+1][y]+
value[x][y+1]+
value[x][y-1]);
diff = (value[x][y] - neighbors[x][y]);
diff *= diff; // squared difference
```

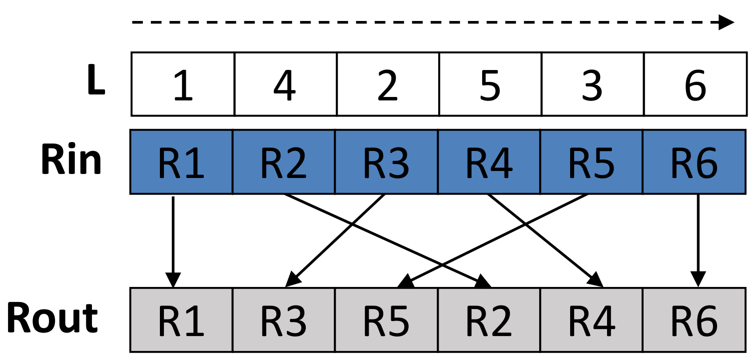
- Load from GPU memory, compute, store to GPU memory
- Make computation as dense as possible to amortize memory access cost
- Maximize number of concurrent threads

Scatter and Gather: Overview

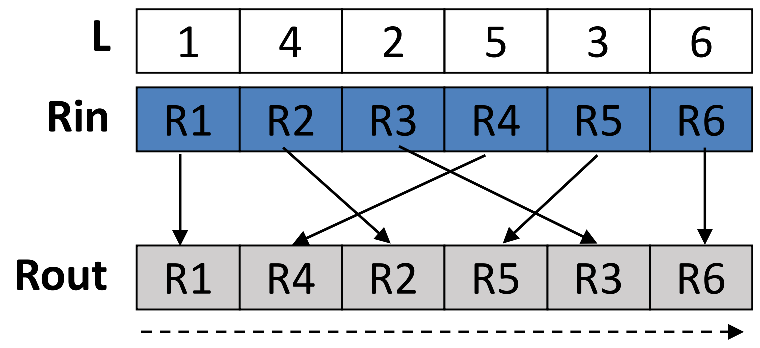
- Widely supported
  - Parallel programming languages, e.g., MPI, NESL, ZPL.
  - Supercomputers, e.g., Cray MTA, Stanford Merrimac
  - Commodity co-processors (IBM Cell, GPUs)
- Irregular access patterns
  - Sparse matrix computations, hashing, searching, etc.
- Performance is memory bandwidth limited
  - Require high bandwidth architectures
  - HPC benchmarks (HPC Challenge, NAS PB, etc.)

Access Patterns

- Scatter: sequential reads and random writes.
- Gather: random reads and sequential writes.



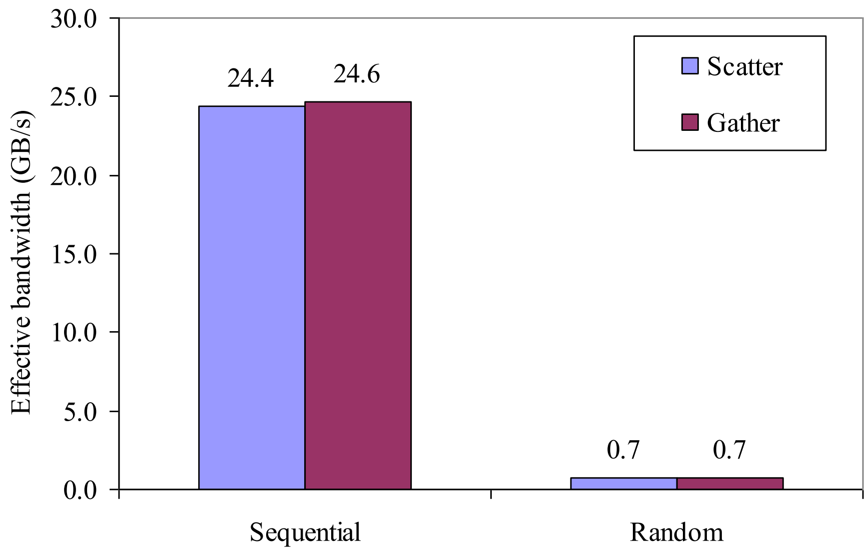
(a) Scatter



(b) Gather

Scatter and Gather on the GPU

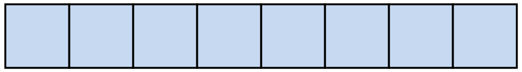
- Access pattern makes a 30X difference in performance [Supercomputing 2007].



Example: Single-pass Scatter

R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
L	0	12	4	8	13	5	1	9	2	14	6	10	15	3	7	11

4 mem. blocks to write
4 concurrent threads
2 cache lines



Cache

Example: Single-pass Scatter

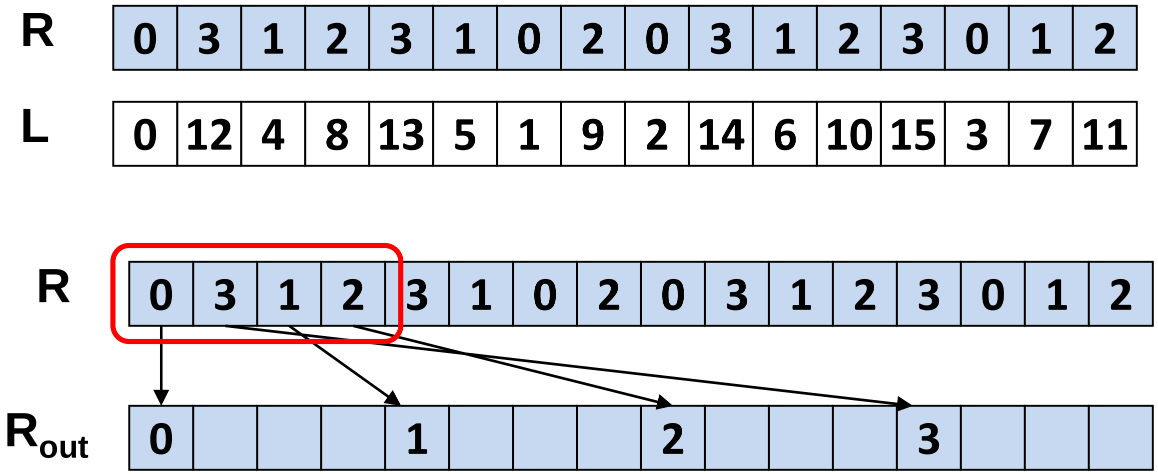
R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
L	0	12	4	8	13	5	1	9	2	14	6	10	15	3	7	11

4 mem. blocks to write
4 concurrent threads
2 cache lines

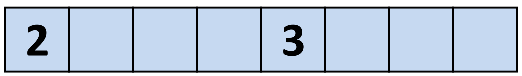
R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
R_{out}																

Cache

Example: Single-pass Scatter



4 mem. blocks to write
4 concurrent threads
2 cache lines



Cache

Cache Misses = 4
Cache Hits = 0

Example: Single-pass Scatter

R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
L	0	12	4	8	13	5	1	9	2	14	6	10	15	3	7	11

4 mem. blocks to write
4 concurrent threads
2 cache lines

R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
R_{out}	0				1				2				3			

2				3			

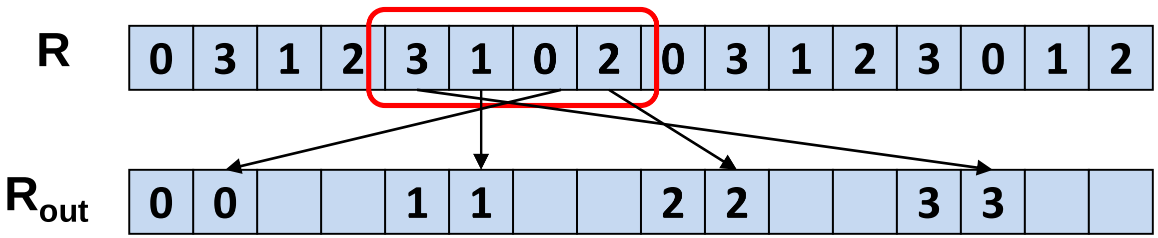
Cache

Cache Misses = 4
Cache Hits = 0

Example: Single-pass Scatter

R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
L	0	12	4	8	13	5	1	9	2	14	6	10	15	3	7	11

4 mem. blocks to write
4 concurrent threads
2 cache lines



0	0			1	1		

Cache

Cache Misses = 6
Cache Hits = 2

Example: Single-pass Scatter

R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
L	0	12	4	8	13	5	1	9	2	14	6	10	15	3	7	11

4 mem. blocks to write
4 concurrent threads
2 cache lines

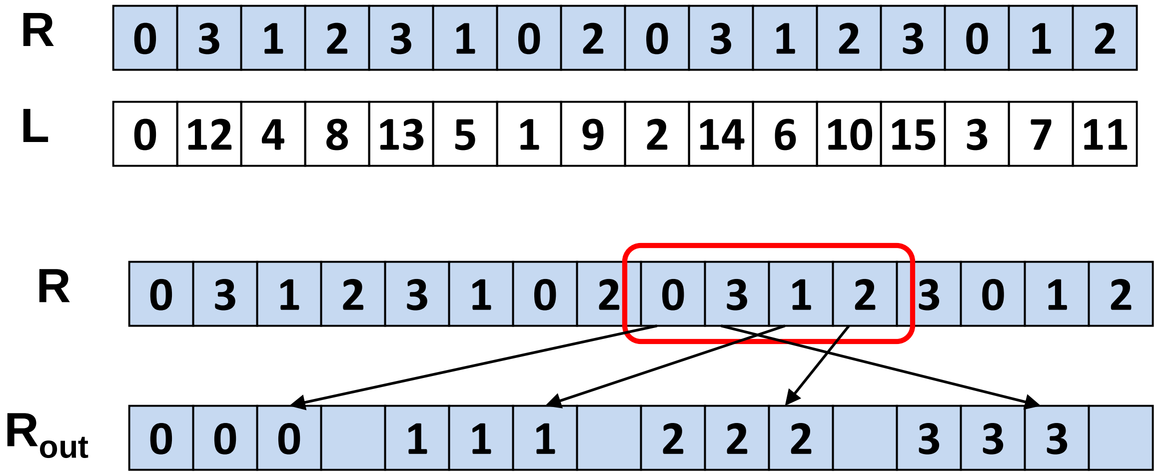
R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
R_{out}	0	0			1	1			2	2			3	3		

0	0			1	1		

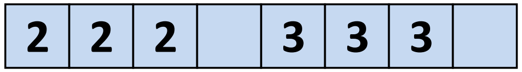
Cache

Cache Misses = 6
Cache Hits = 2

Example: Single-pass Scatter



4 mem. blocks to write
4 concurrent threads
2 cache lines



Cache

Cache Misses = 8
Cache Hits = 4

Example: Single-pass Scatter

R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
L	0	12	4	8	13	5	1	9	2	14	6	10	15	3	7	11

4 mem. blocks to write
4 concurrent threads
2 cache lines

R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
R_{out}	0	0	0		1	1	1		2	2	2		3	3	3	

2	2	2		3	3	3	

Cache

Cache Misses = 8
Cache Hits = 4

Example: Single-pass Scatter

4 mem. blocks to write
4 concurrent threads
2 cache lines

R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
L	0	12	4	8	13	5	1	9	2	14	6	10	15	3	7	11

R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
R_{out}	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3

0	0	0	0	1	1	1	1

Cache

Cache Misses = 10
Cache Hits = 6

Cache miss rate = 62.5%
Effective write bandwidth = |R|/Transfer Time = 4/10* B_{seq}= 0.4 B_{seq}

Multi-pass Scheme

- The entire scatter is performed in multiple passes.
- Each pass writes to a small chunk

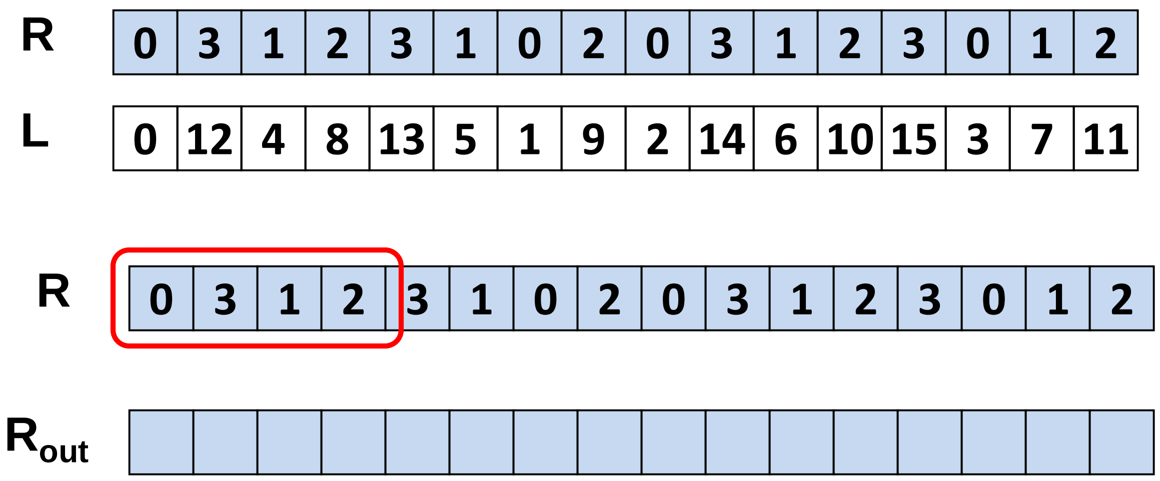
Two-pass Scatter

R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
L	0	12	4	8	13	5	1	9	2	14	6	10	15	3	7	11

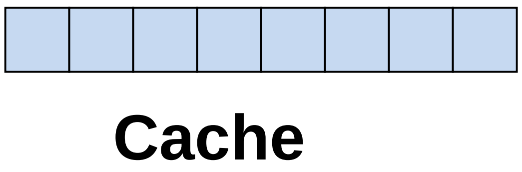
4 mem. blocks to write
4 concurrent threads
2 cache lines

Cache

Two-pass Scatter



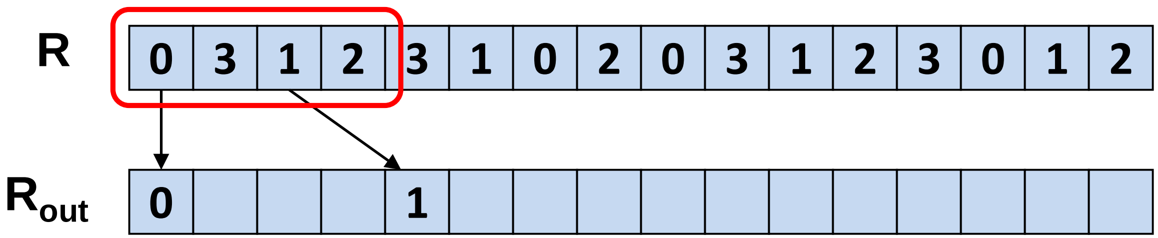
4 mem. blocks to write
4 concurrent threads
2 cache lines

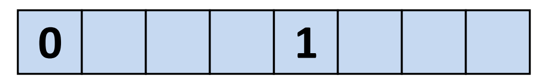


Two-pass Scatter

R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
L	0	12	4	8	13	5	1	9	2	14	6	10	15	3	7	11

4 mem. blocks to write
4 concurrent threads
2 cache lines





Cache

Cache Misses = 2
Cache Hits = 0

Two-pass Scatter

R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
L	0	12	4	8	13	5	1	9	2	14	6	10	15	3	7	11

4 mem. blocks to write
4 concurrent threads
2 cache lines

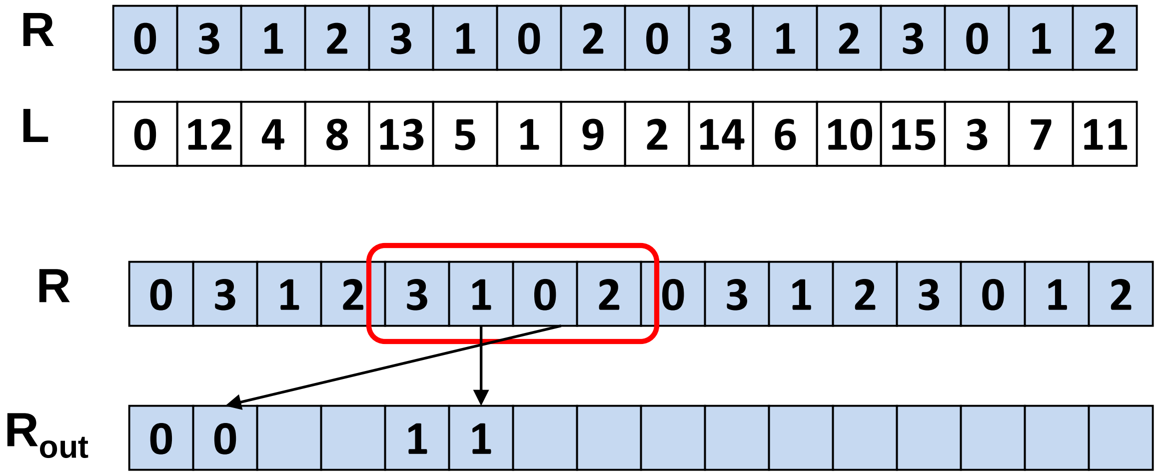
R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
R_{out}	0				1											

0				1			

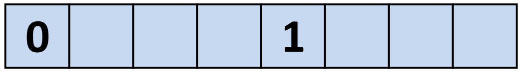
Cache

Cache Misses = 2
Cache Hits = 0

Two-pass Scatter



4 mem. blocks to write
4 concurrent threads
2 cache lines



Cache

Cache Misses = 2
Cache Hits = 2

Two-pass Scatter

R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
L	0	12	4	8	13	5	1	9	2	14	6	10	15	3	7	11

4 mem. blocks to write
4 concurrent threads
2 cache lines

R	0	3	1	2	3	1	0	2	$\boxed{0}$	$\boxed{3}$	$\boxed{1}$	$\boxed{2}$	3	0	1	2
R_{out}	0	0			1	1										

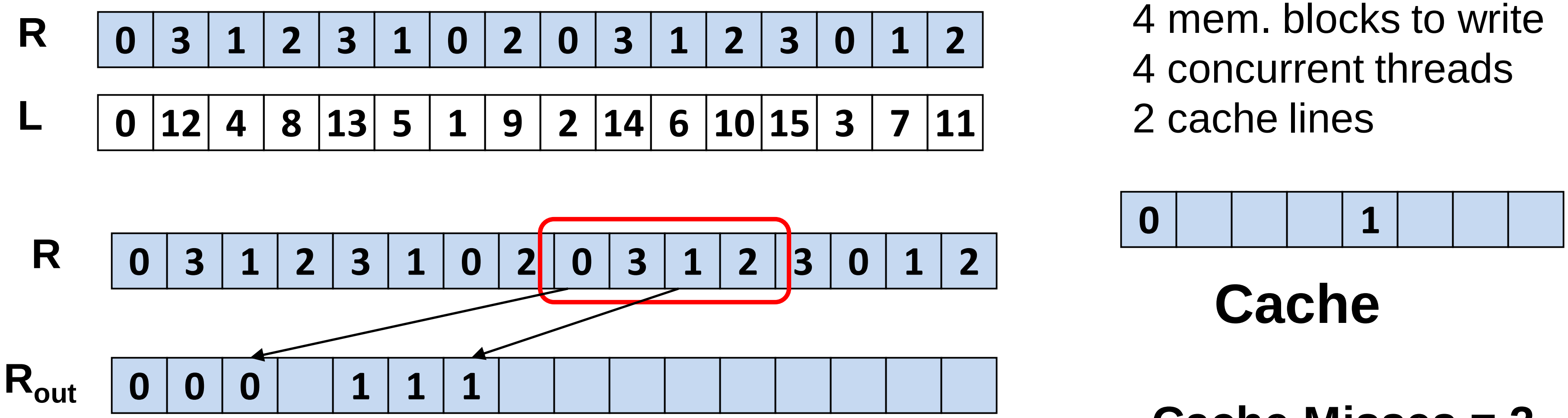
0				1			

Cache

Cache Misses = 2
Cache Hits = 2

Two-pass Scatter

4 mem. blocks to write
4 concurrent threads
2 cache lines



Cache Misses = 2
Cache Hits = 4

Two-pass Scatter

R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
L	0	12	4	8	13	5	1	9	2	14	6	10	15	3	7	11

4 mem. blocks to write
4 concurrent threads
2 cache lines

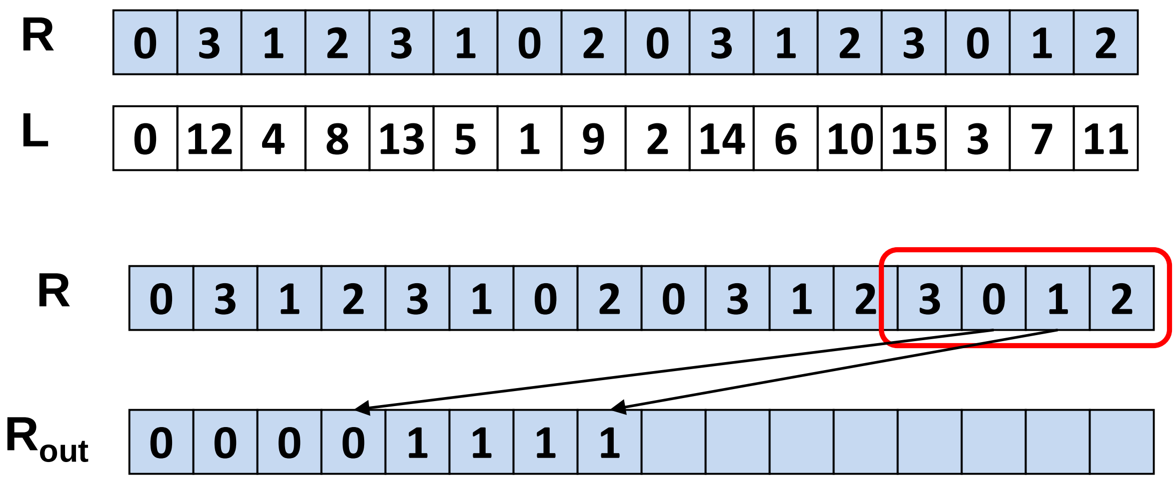
R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
R_{out}	0	0	0		1	1	1									

0				1			

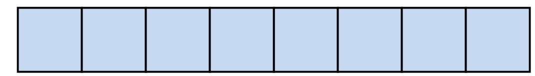
Cache

Cache Misses = 2
Cache Hits = 4

Two-pass Scatter



4 mem. blocks to write
4 concurrent threads
2 cache lines



Cache

Cache Misses = 2
Cache Hits = 6

Two-pass Scatter

R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
L	0	12	4	8	13	5	1	9	2	14	6	10	15	3	7	11

4 mem. blocks to write
4 concurrent threads
2 cache lines

R	0	3	1	2	3	1	0	2	0	3	1	2	3	0	1	2
R_{out}	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3

Cache

Cache Misses = 4
Cache Hits = 12

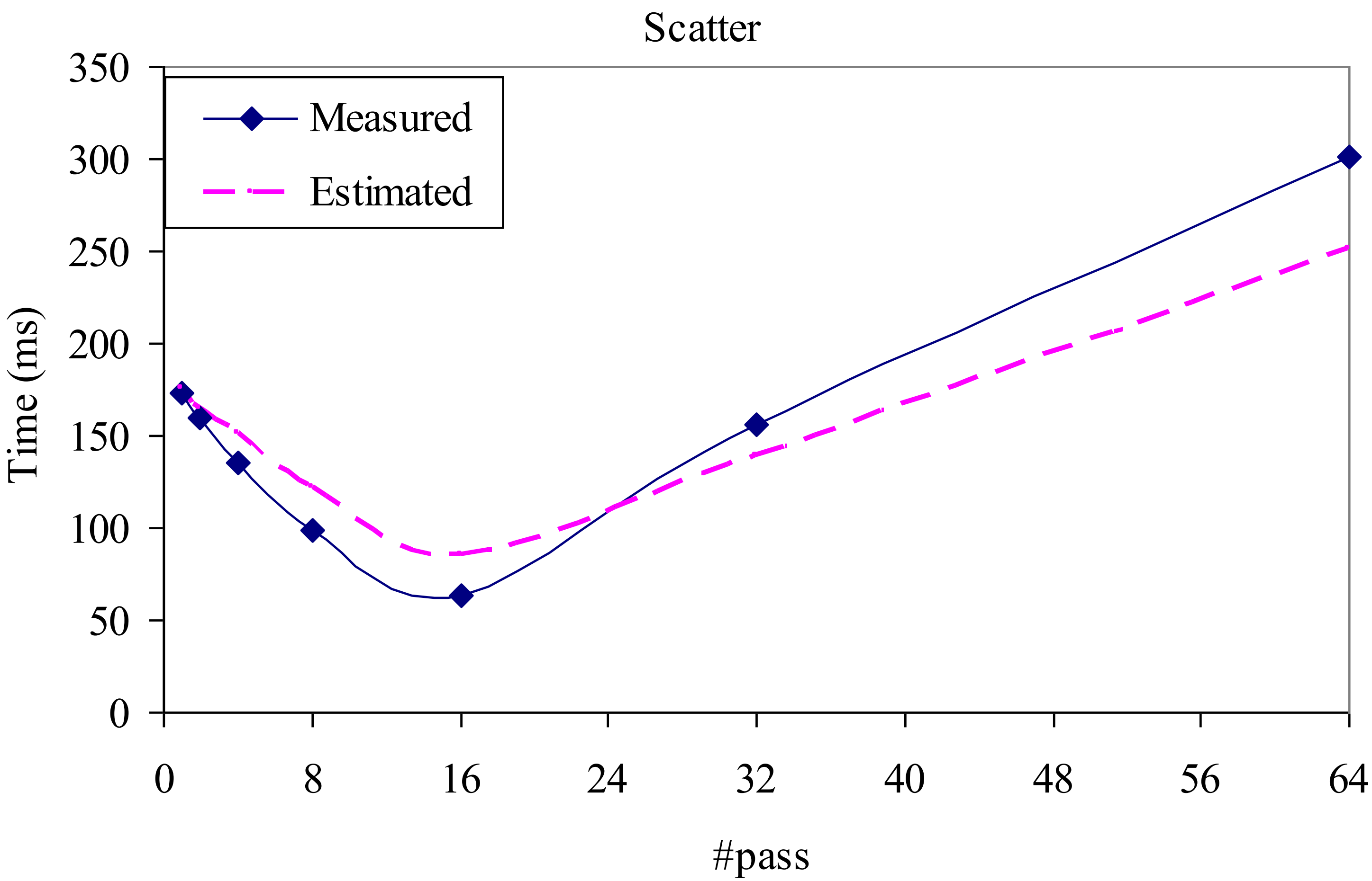
Cache miss rate = 25%
Effective write bandwidth = B_{seq}

Cost Model

- Estimate the performance of different access patterns
  - Sequential bandwidth
  - Random bandwidths of different degrees
- Estimate the total cost of sequential access and random access in the multi-pass scheme.

  $T_{scatter} = (|R| + |L|) * npasses/B_{seq} + |R|/B_{rand}$
- Determine the optimal number of passes.

Performance Results -- Multi-pass Scatter

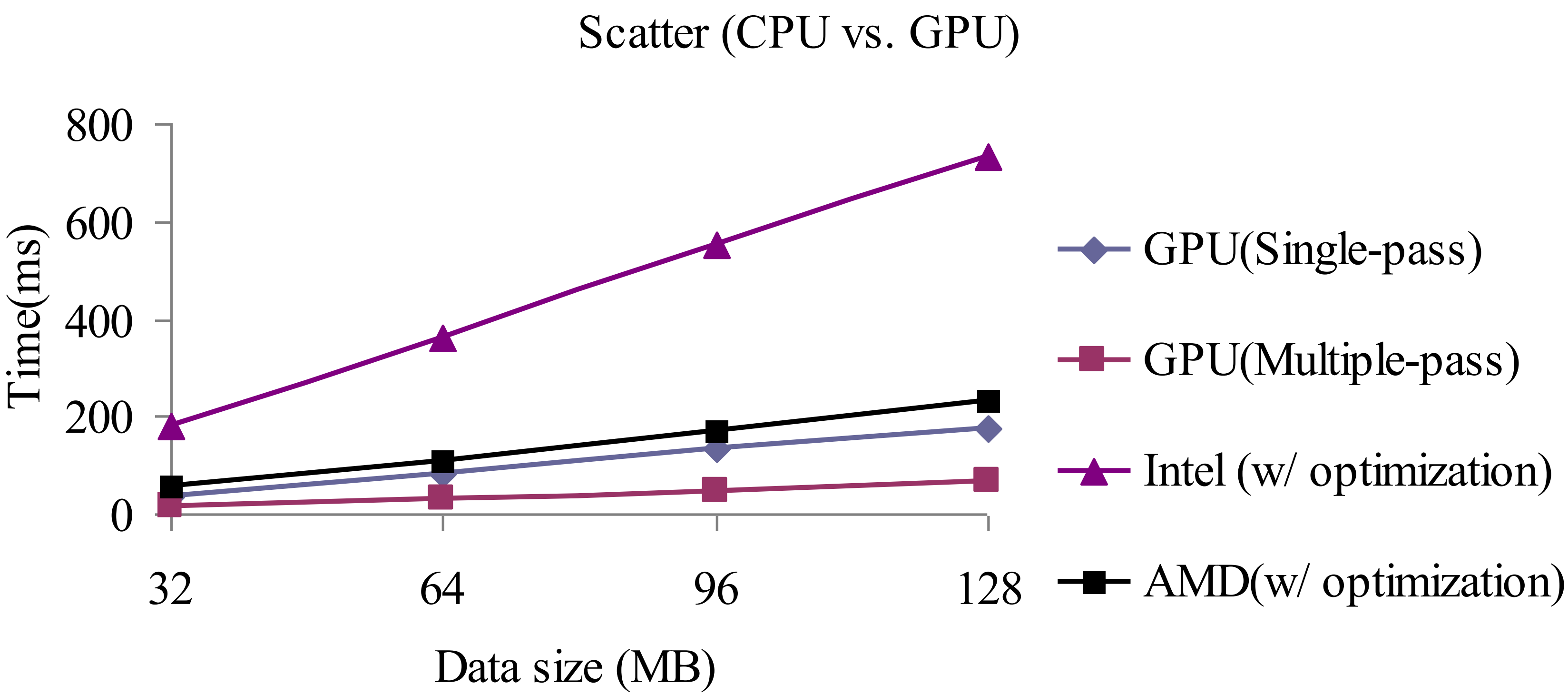


The optimal number of passes is 16.

Applications and Analysis

- Applications
  - Radix sort, hash search, and sparse-matrix vector multiplication
- Platforms
  - CPUs: Intel Quad, or two AMD dual-core processors.
  - GPU: Nvidia 8800 GTX.
- Overall results
  - The cost model has an accuracy of over 85%.
  - The multipass scheme improves the application 10%~50%.
  - The GPU-based algorithm outperforms the CPU-based algorithm by 2-7X.

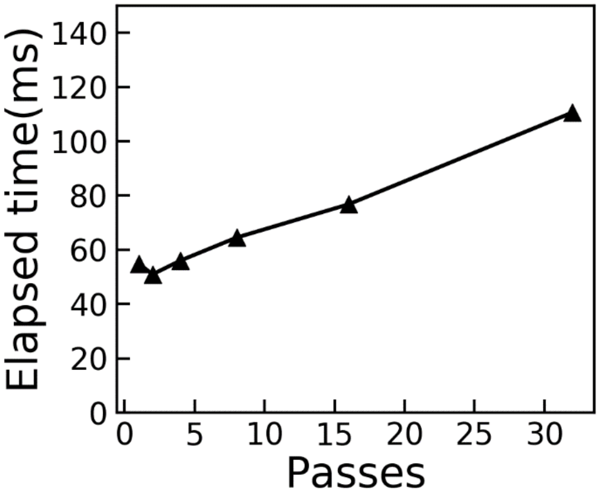
Performance Impact of Multi-Pass Scatter



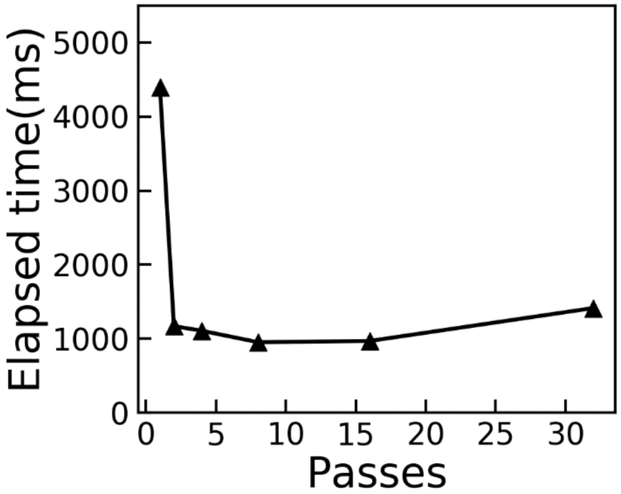
(1) The speedup is 7-13X and 2-4X on Intel and AMD, respectively.
(2) The multi-pass scheme improves the GPU-based scatter by 2-4X.

Newer Results (2018)

- Multi-pass scatter & gather on modern GPUs
  - Nvidia Tesla K40 GPU (LLC: 1536 KB)
  - 256MB & 3072MB 4-byte tuples
  - Random data distribution



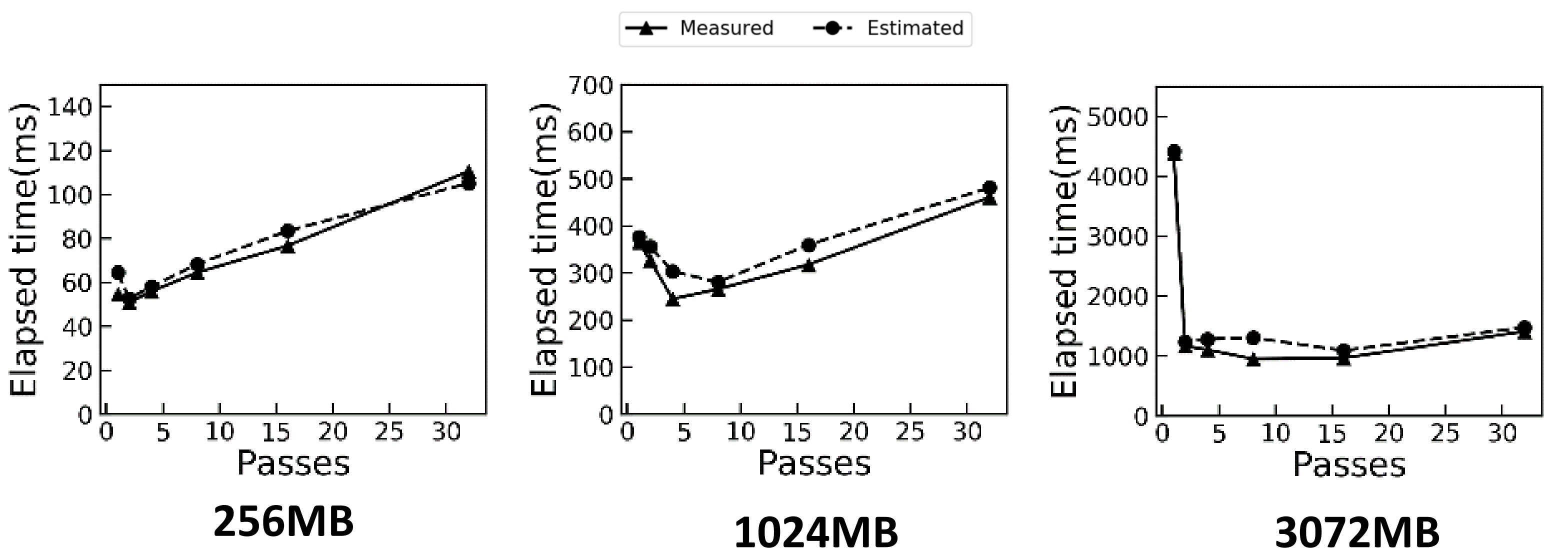
256MB



3072MB

New Performance Model (2018)

- Considering not only data caching but also Translation Lookaside Buffer (TLB) caching



Scatter, K40

Summary

- Data-parallel primitives are an effective way of utilizing GPU's parallelism.
- Scatter and gather are memory-bound and can be optimized through multi-pass schemes.

References:
Bingsheng He, Naga K. Govindaraju, Qiong Luo, and Burton Smith. Efficient Gather and Scatter Operations on Graphics Processors. ACM/IEEE SuperComputing (SC), Nov 2007.

Zhuohang Lai, Qiong Luo, Xiaoying Jia:
Revisiting Multi-pass Scatter and Gather on GPUs. ICPP 2018: 25:1-25:11