CS 839: Design the Next-Generation Database

Lecture 7: GPU Database

Xiangyao Yu

2/11/2020

Announcements

[Optional] 5-min presentation of your project idea

- Find teammates
- Receive feedback
- Email me if you are interested

Discussion Highlights

Necessary to know read/write set?

- No. But not knowing the sets severely degrades performance. (any solutions?)

Optimizations if know read/write sets?

- No need to broadcast reads to all active participants
- Use better deterministic ordering to improve performance
- Enforce no conflicts within a batch -> no need to lock
- Blind write optimization

Batch to amortize 2PC?

- Run 2PC in batches
- Epoch-based concurrency control like Silo

Today's Paper

A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics

Anil Shanbhag
MIT
anil@csail.mit.edu

Samuel Madden
MIT
madden@csail.mit.edu

Xiangyao Yu
University of Wisconsin-Madison
yxy@cs.wisc.edu

ABSTRACT

There has been significant amount of excitement and recent work on GPU-based database systems. Previous work has claimed that these systems can perform orders of magnitude better than CPU-based database systems on analytical workloads such as those found in decision support and

KEYWORDS

in-memory analytics, heterogenous systems, GPU data analytics, GPU query performance

ACM Reference Format:
Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A Study

SIGMOD 2020

Today’s Agenda

GPU background

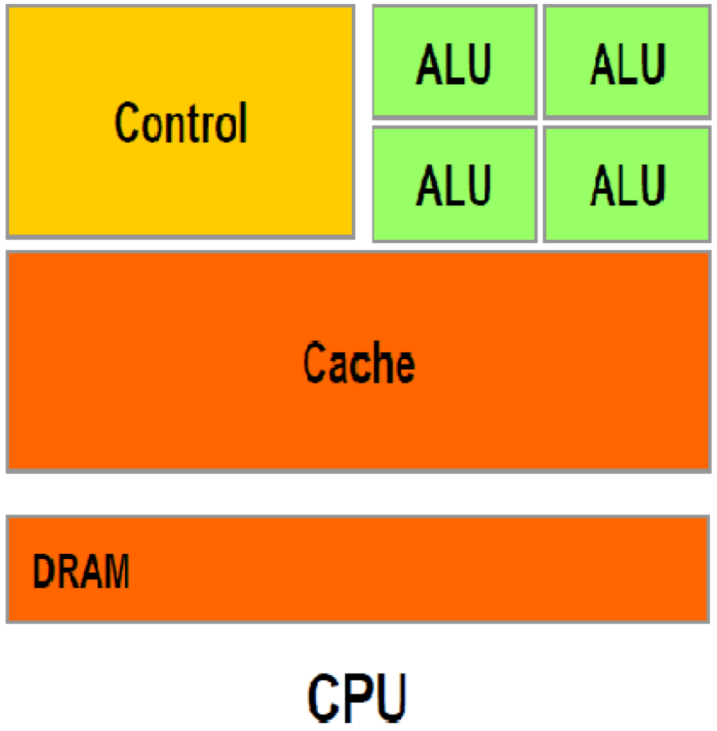
Data analytics on CPU vs. GPU

Crystal library

GPU Background

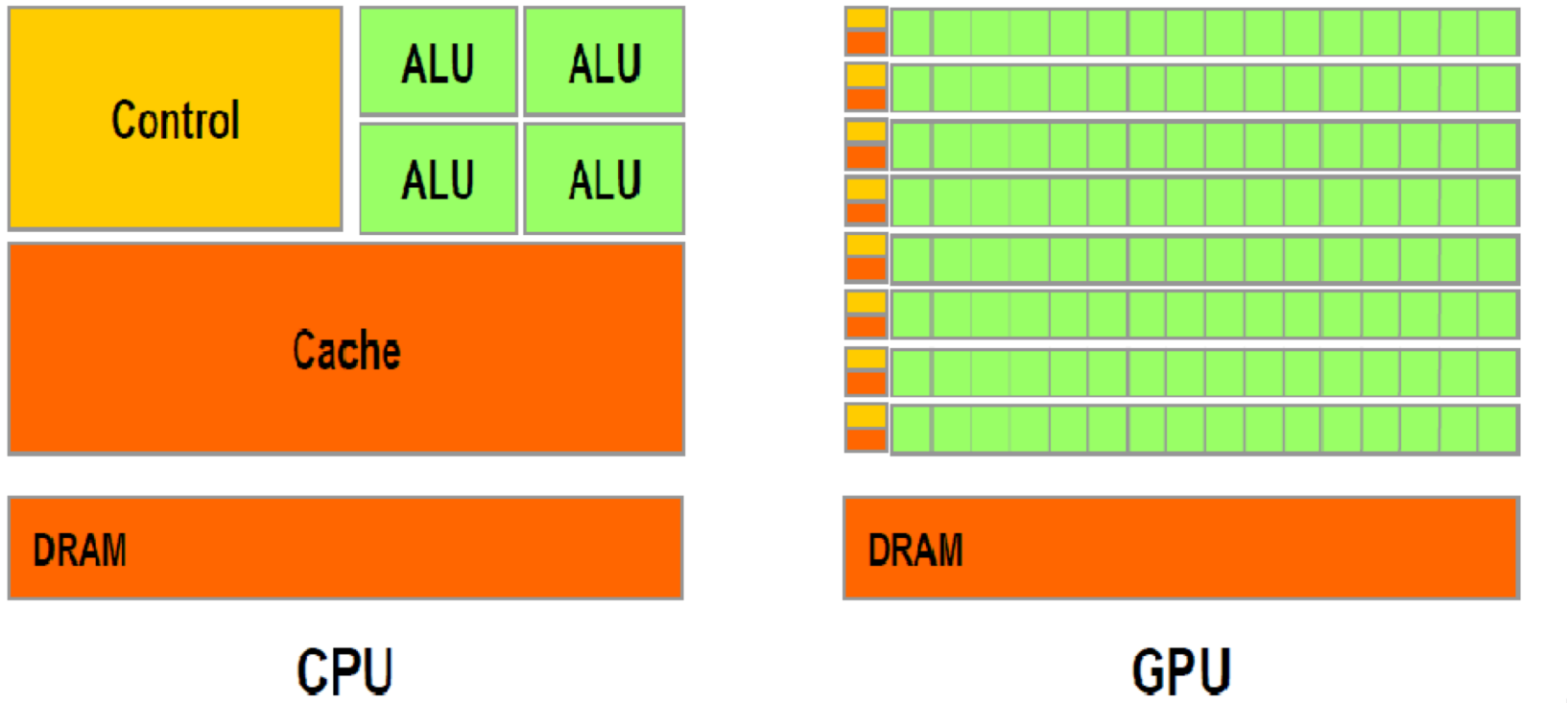
- Graphics processing unit (GPU)
- Accelerators for graphics computation
- Dedicated accelerators with simple, massively parallel computation
- More and more used for general-purpose computing

CPU vs. GPU



CPU: A few powerful cores with large caches. Optimized for sequential computation

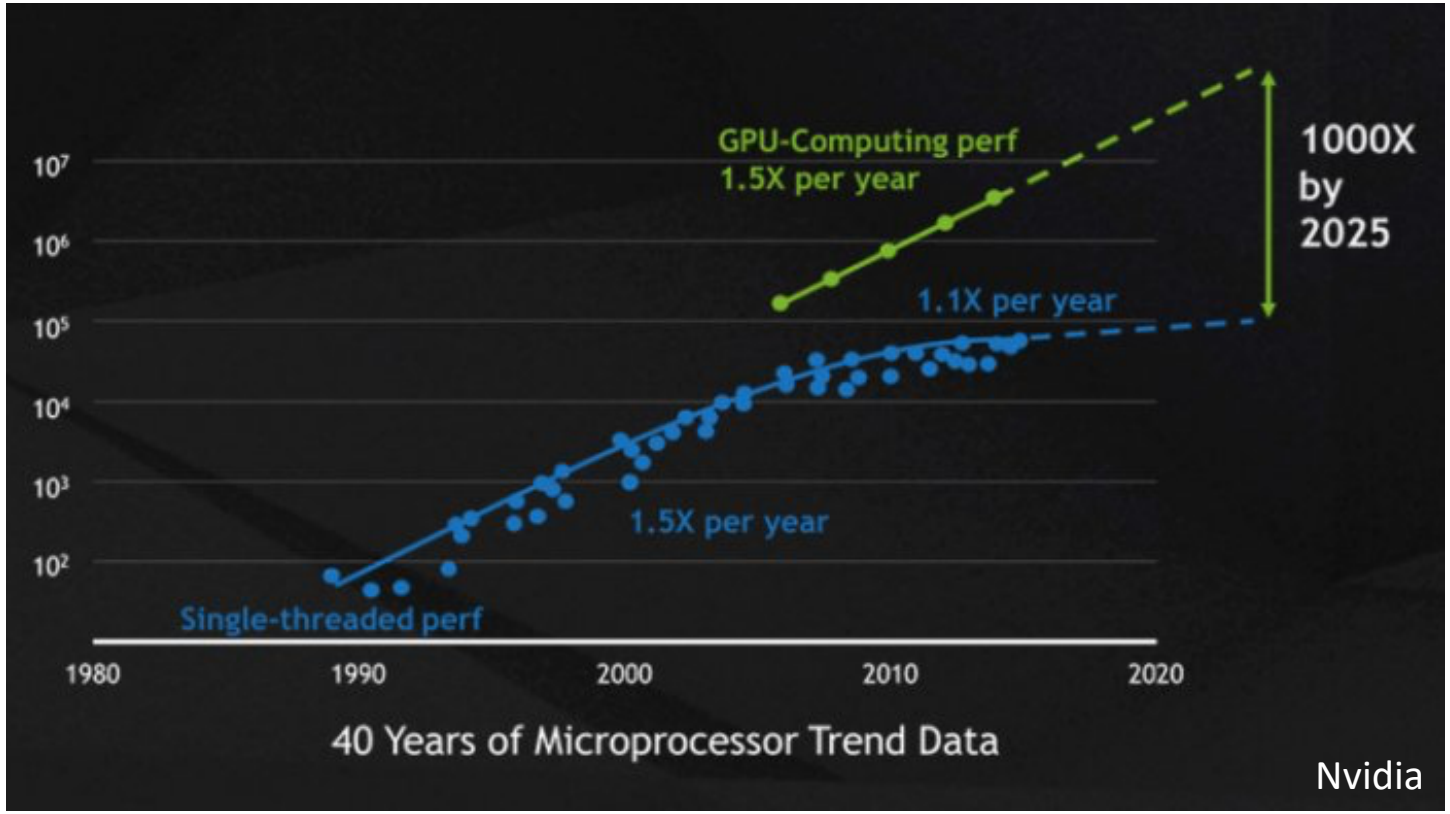
CPU vs. GPU



CPU: A few powerful cores with large caches. Optimized for sequential computation

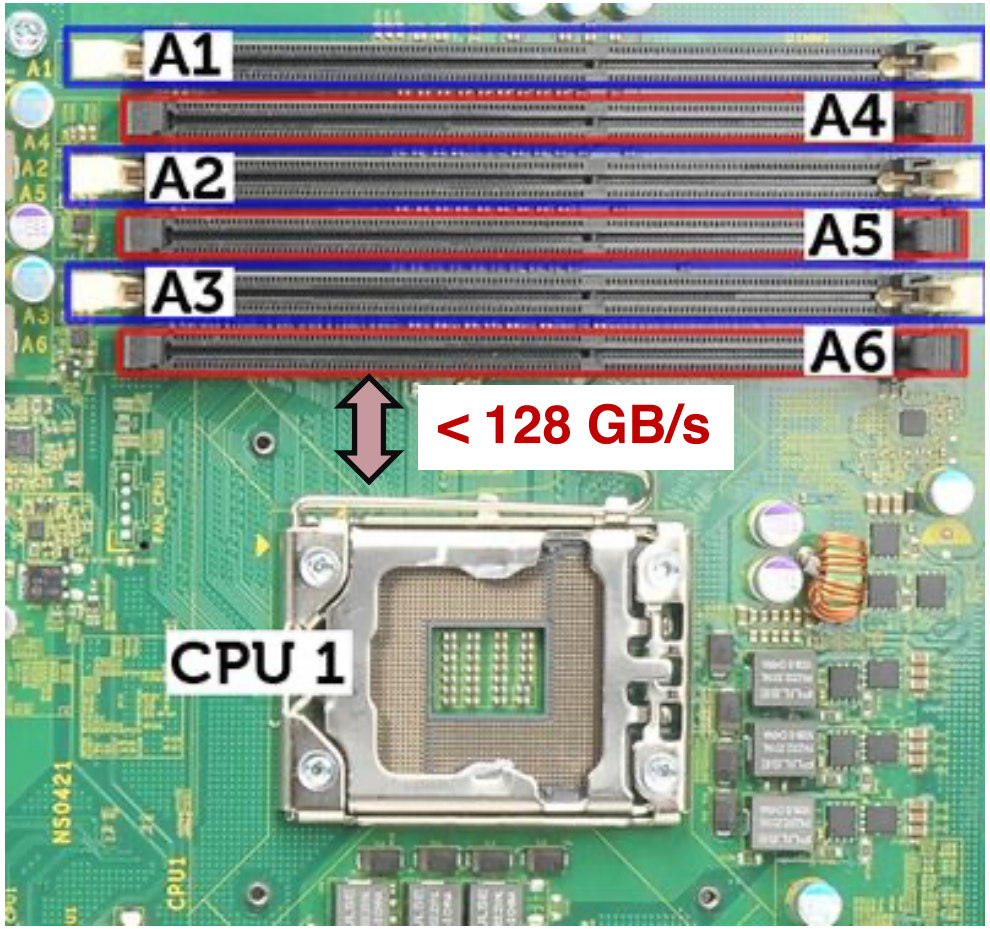
GPU: Many small cores. Optimized for parallel computation

CPU vs. GPU – Processing Units



	Throughput	Power	Throughput/Power
Intel Skylake	128 GFLOPS/4 Cores	100+ Watts	~1 GFLOPS/Watt
NVIDIA V100	15 TFLOPS	200+ Watts	~75 GFLOPS/Watt

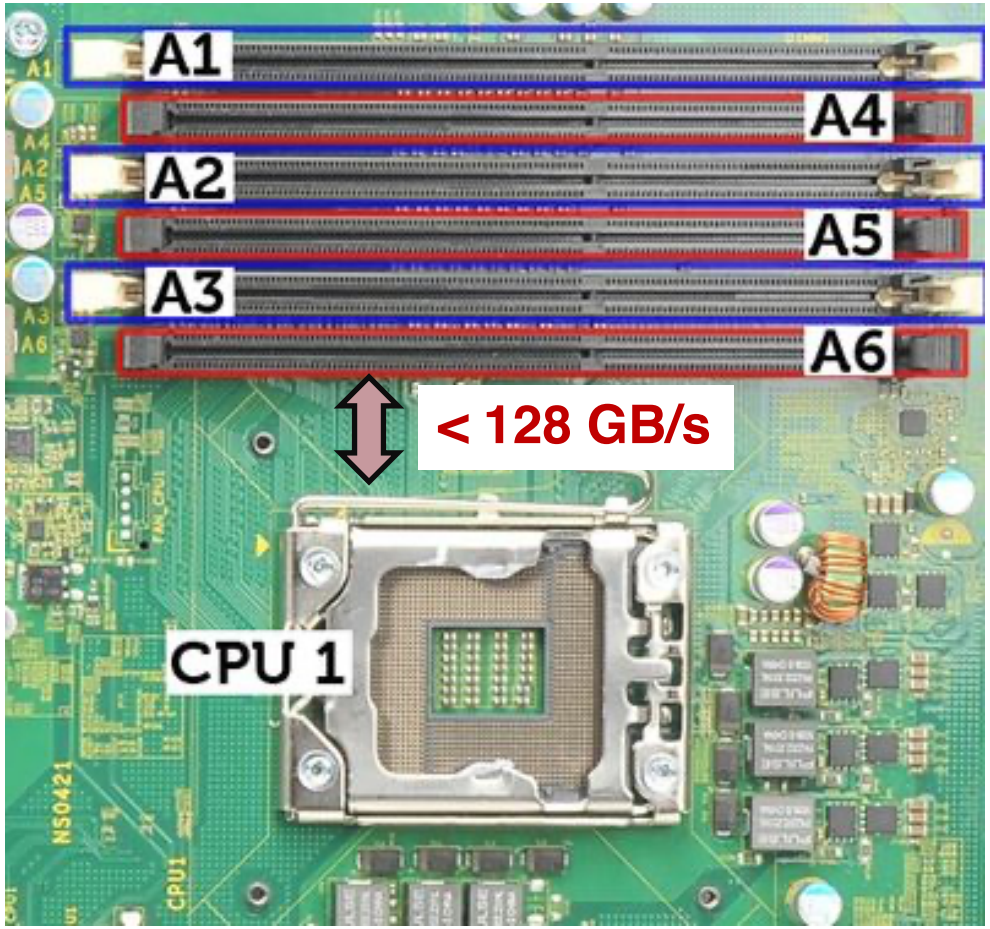
CPU vs. GPU – Memory System



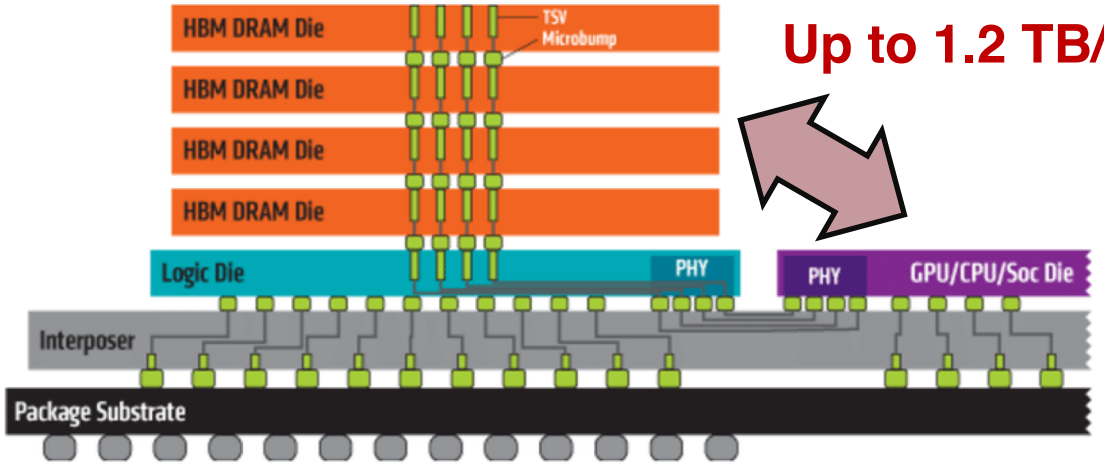
DRAM DIMMs

CPU: Large memory (up to Terabytes) with limited bandwidth (up to 100GB/s)

CPU vs. GPU – Memory System



DRAM DIMMs

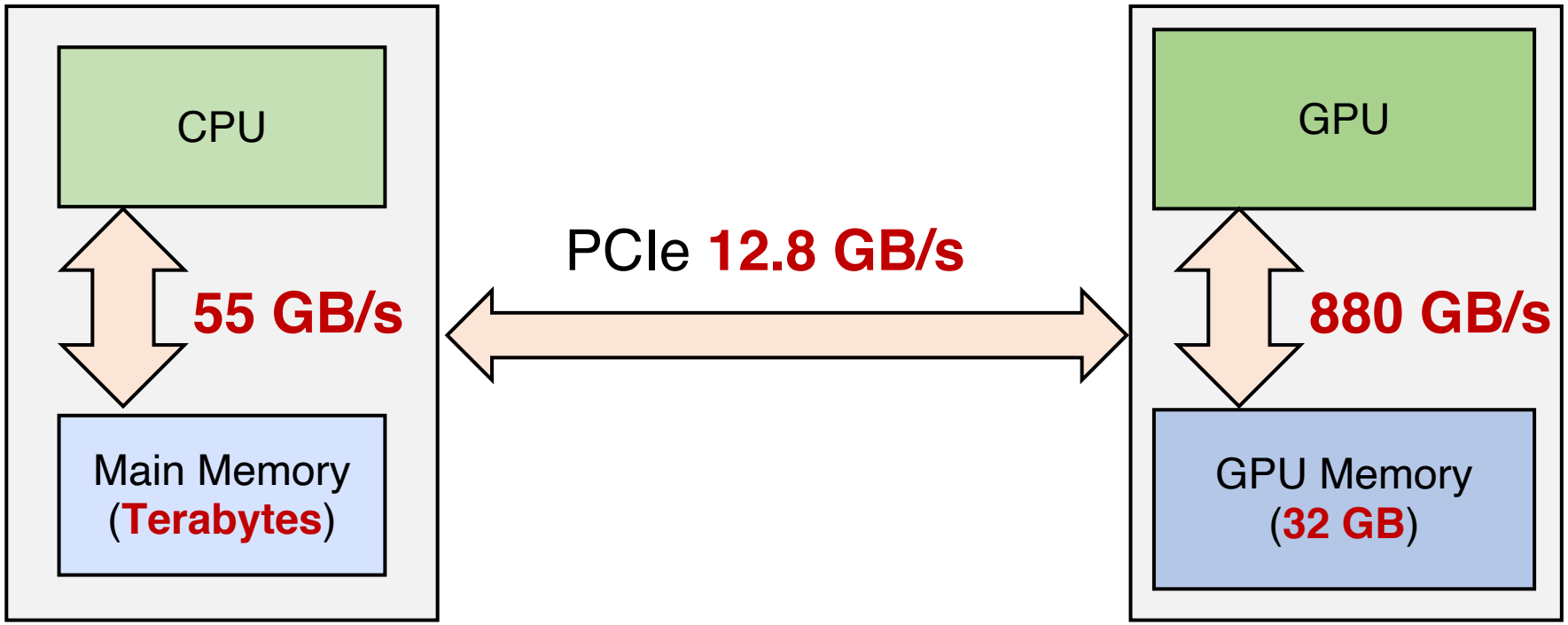


Up to 1.2 TB/s

CPU: Large memory (up to Terabytes) with limited bandwidth (up to 100GB/s)

GPU: Small memory (up to 32 GB) with high bandwidth (up to 1.2 TB/s)

CPU vs. GPU – Overall Architecture



GPU has immense computational power

GPU memory has high bandwidth

GPU memory has small capacity

Loading data from main memory is slow

GPU Database Operation Mode

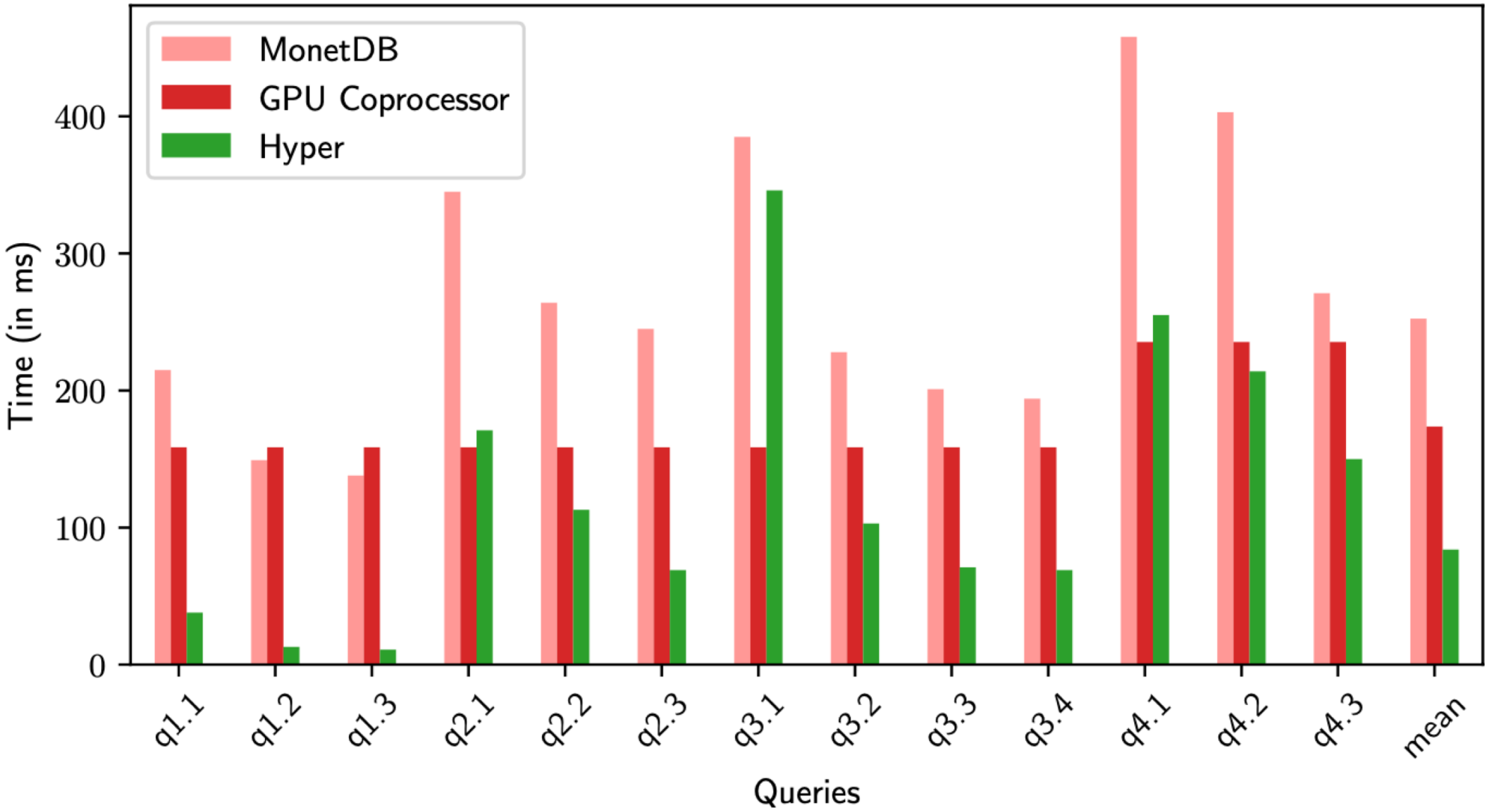
Coprocessor mode: Every query loads data from CPU memory to GPU

GPU-only mode: Store working set in GPU memory and run the entire query on GPU

Key observation: With efficient implementations that can saturate memory bandwidth

GPU-only > CPU-only > coprocessor

CPU-only vs. Coprocessor



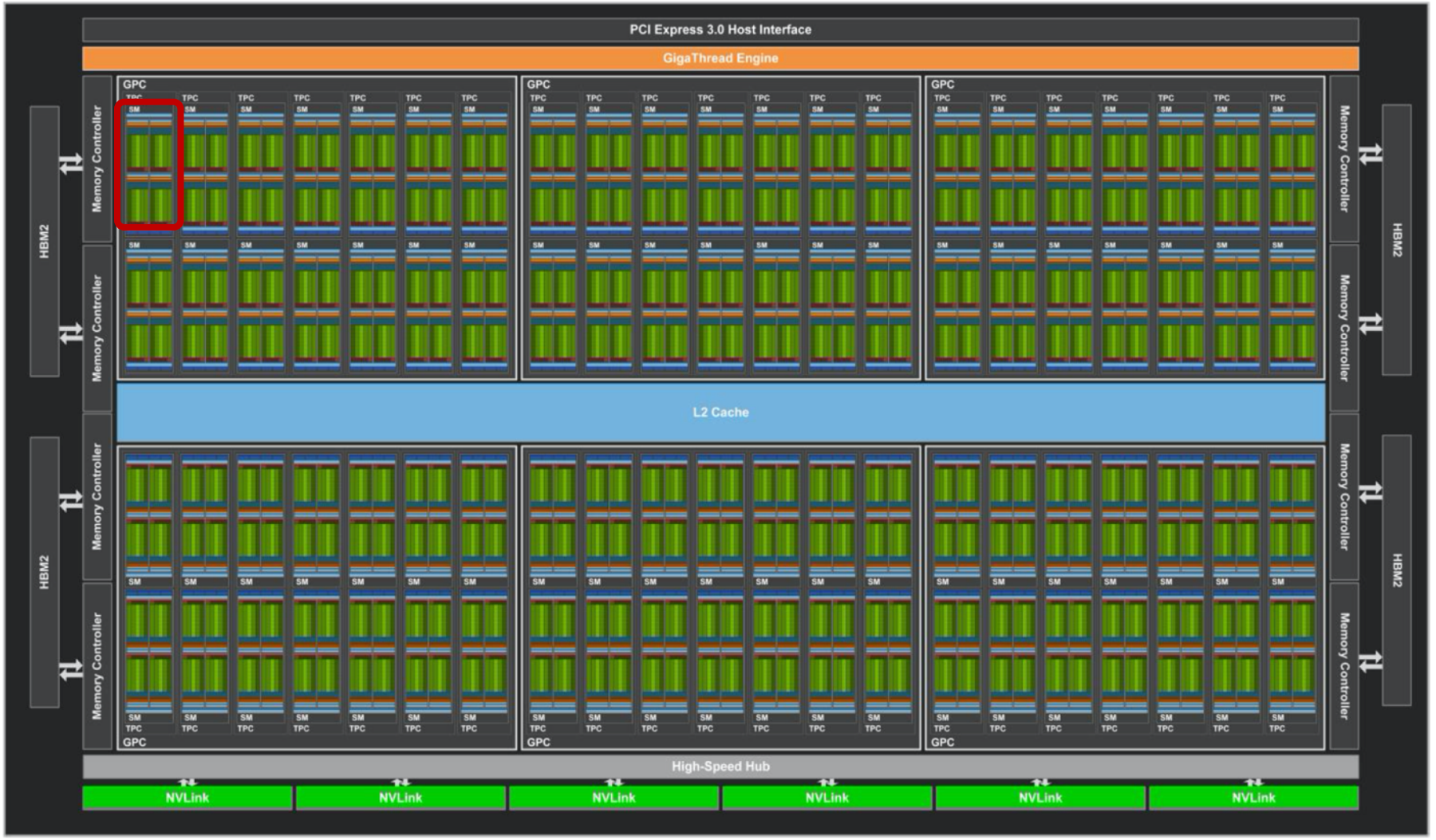
Efficient Query Execution on GPUs

Tile-based Execution Model

Crystal Library

GPU Architecture

SM



84 streaming Multiprocessors (SM)

GPU Architecture – Streaming Multiprocessor

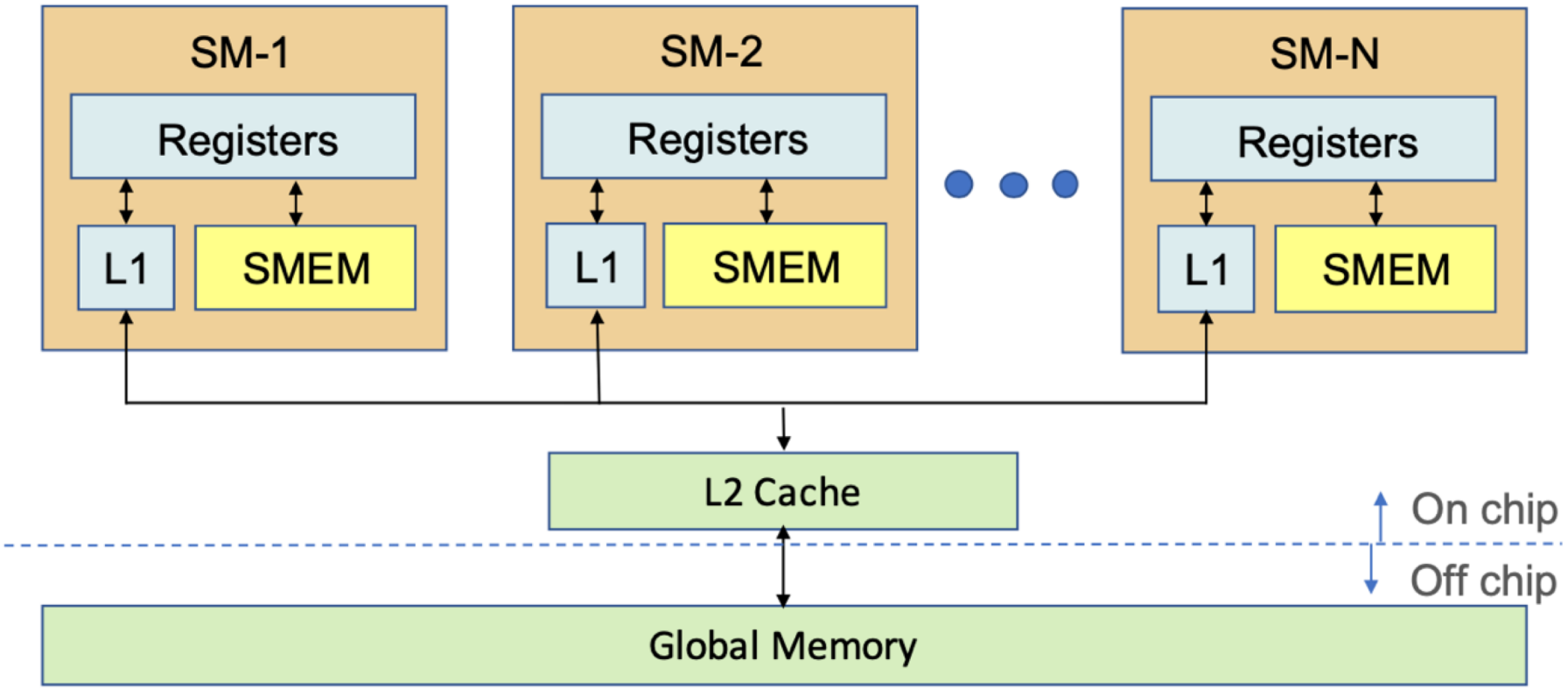
Each SM has 4 warps

Each warp contains 32 threads

Each warp executes in a single instruction multiple threads (SIMT) model

[1] V100 GPU Hardware Architecture In-Depth,
https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

GPU Architecture – Memory System



Data from global memory cached in L2/L1

Shared memory: a scratchpad controlled by the programmer

Sequential vs. Parallel

```
Q0: SELECT y FROM R WHERE y > v;
```

Goal: write the results in parallel into a contiguous output array

Sequential

```
cnt = 0
for i in R.size():
  if R[i] > v
    output[cnt++] = R[i]
```

Sequential vs. Parallel

```
Q0: SELECT y FROM R WHERE y > v;
```

Goal: write the results in parallel into a contiguous output array

Sequential

```
cnt = 0
for i in R.size():
  if R[i] > v
    output[cnt++] = R[i]
```

Parallel

```
for start in partitions[thread_id]
  cnt = 0
  for (i=start; i<start+1000; i++)
    if R[i] > v
      cnt ++
  out_offset = atom_add(&out_pos, cnt)
  for (i=start; i<start+1000; i++)
    if R[i] > v
      output[out_offset ++] = R[i]
```

Sequential vs. Parallel

```
Q0: SELECT y FROM R WHERE y > v;
```

Goal: write the results in parallel into a contiguous output array

Sequential

```
cnt = 0
for i in R.size():
  if R[i] > v
    output[cnt++] = R[i]
```

Parallel

```
for start in partitions[thread_id]
  cnt = 0
  for (i=start; i<start+1000; i++)
    if R[i] > v
      cnt ++
  out_offset = atom_add(&out_pos, cnt)
  for (i=start; i<start+1000; i++)
    if R[i] > v
      output[out_offset ++] = R[i]
```

Vector-based execution model

Parallel on CPU vs. GPU

```
Q0: SELECT y FROM R WHERE y > v;
```

Goal: write the results in parallel into a contiguous output array

Parallel

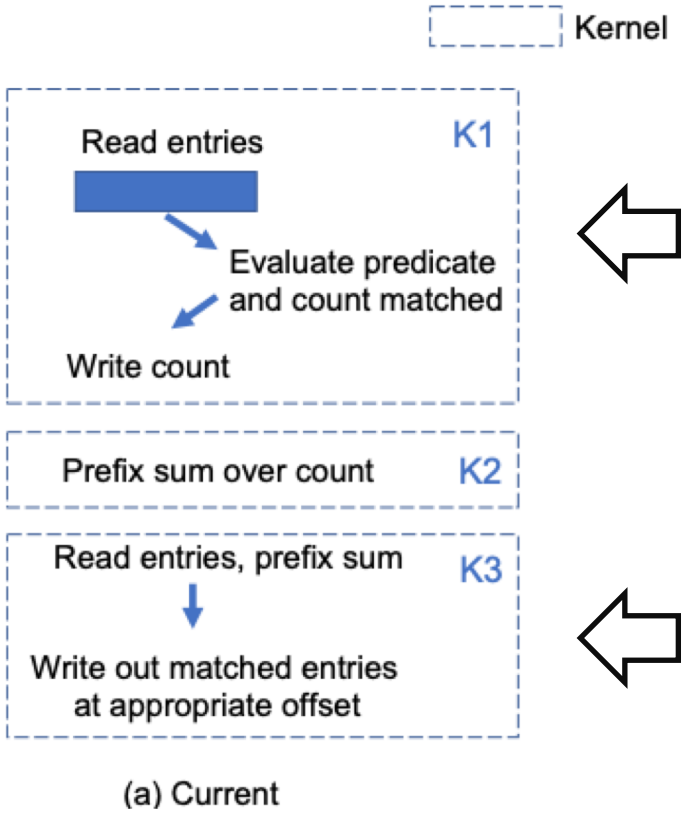
```
for p_start in partitions[thread_id]
  cnt = 0
  for (i=p_start; i<p_start+1000; i++)
    if R[i] > v
      cnt ++
  out_offset = atom_add(&out_pos, cnt)
  for (i=start; i<start+1000; i++)
    if R[i] > v
      output[out_offset ++] = R[i]
```

In CPU, **10s** of threads call atom_add()

In GPU, **1000s** of threads call atom_add()
--> performance bottleneck

Current GPU Parallel Implementation

```
Q0: SELECT y FROM R WHERE y > v;
```



(a) Current

K1: Load from global memory

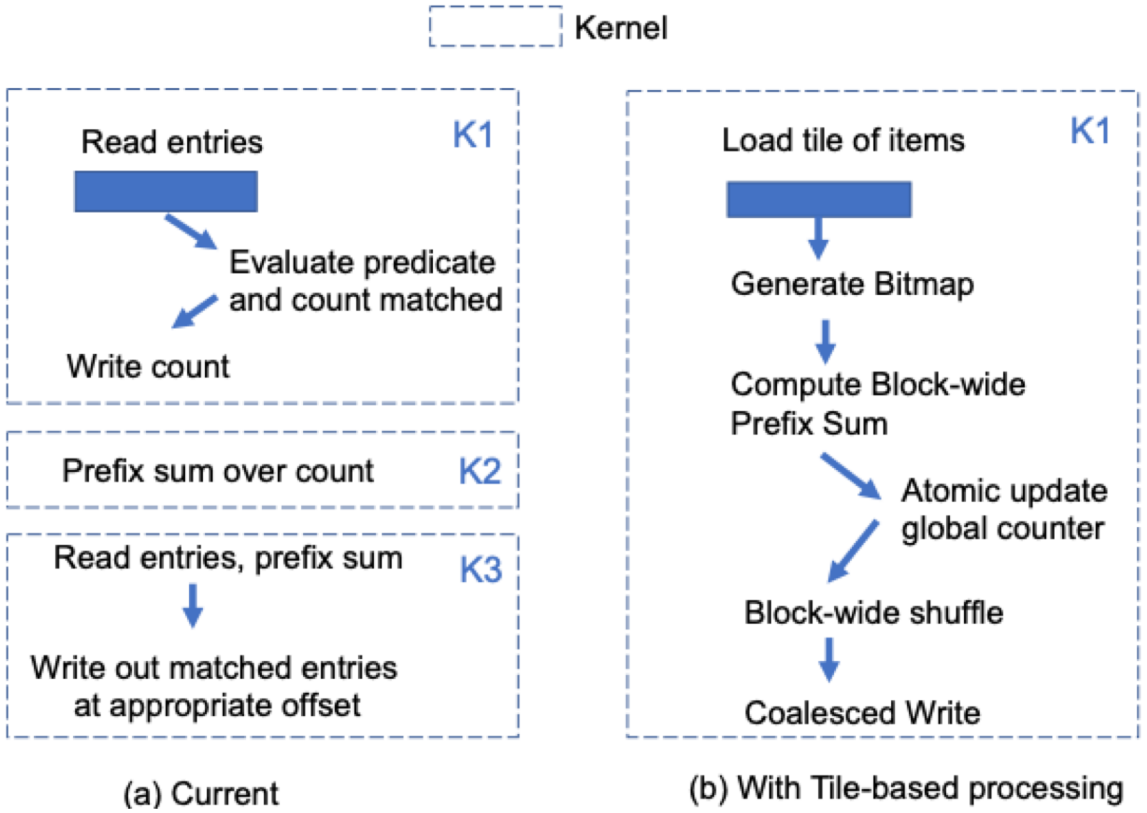
K3: Load from global memory

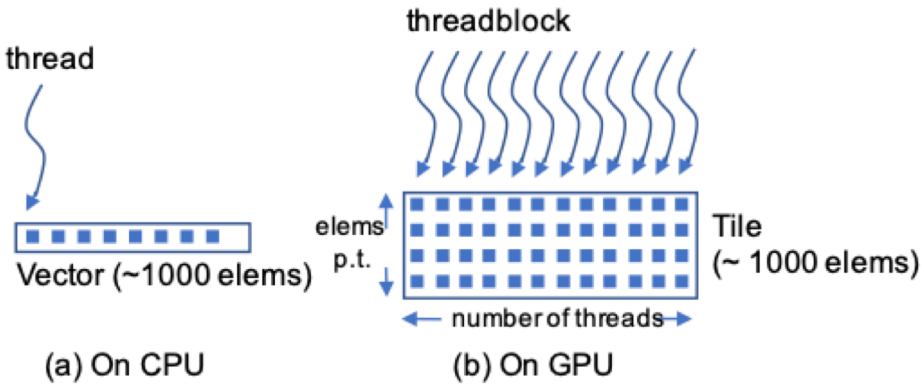
Issue 1: Input array read from global memory twice

Issue 2: each thread writes to a different location in output array

Tile-Based Execution Model

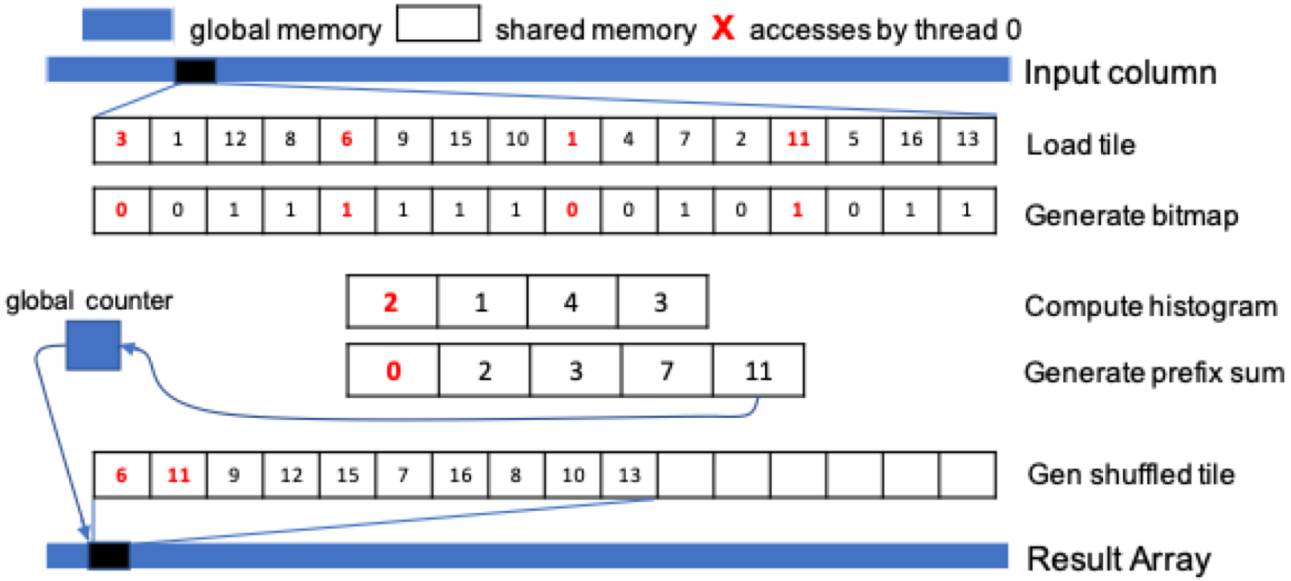
```
Q0: SELECT y FROM R WHERE y > v;
```

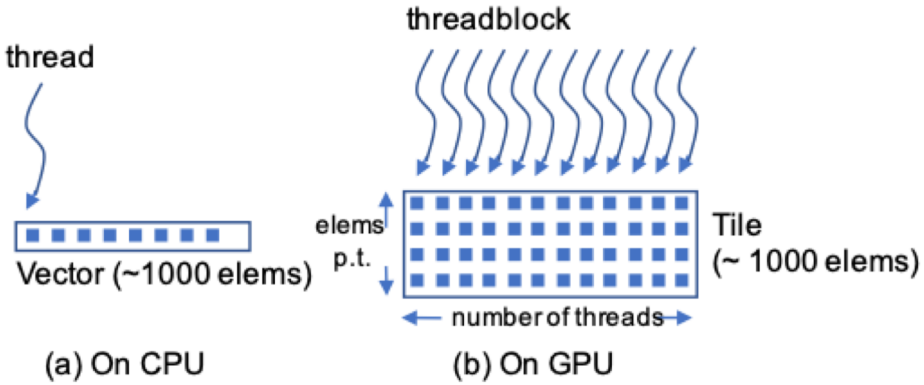




Tile-Based Execution Model – Example

```
Q0: SELECT y FROM R WHERE y > 5;
```





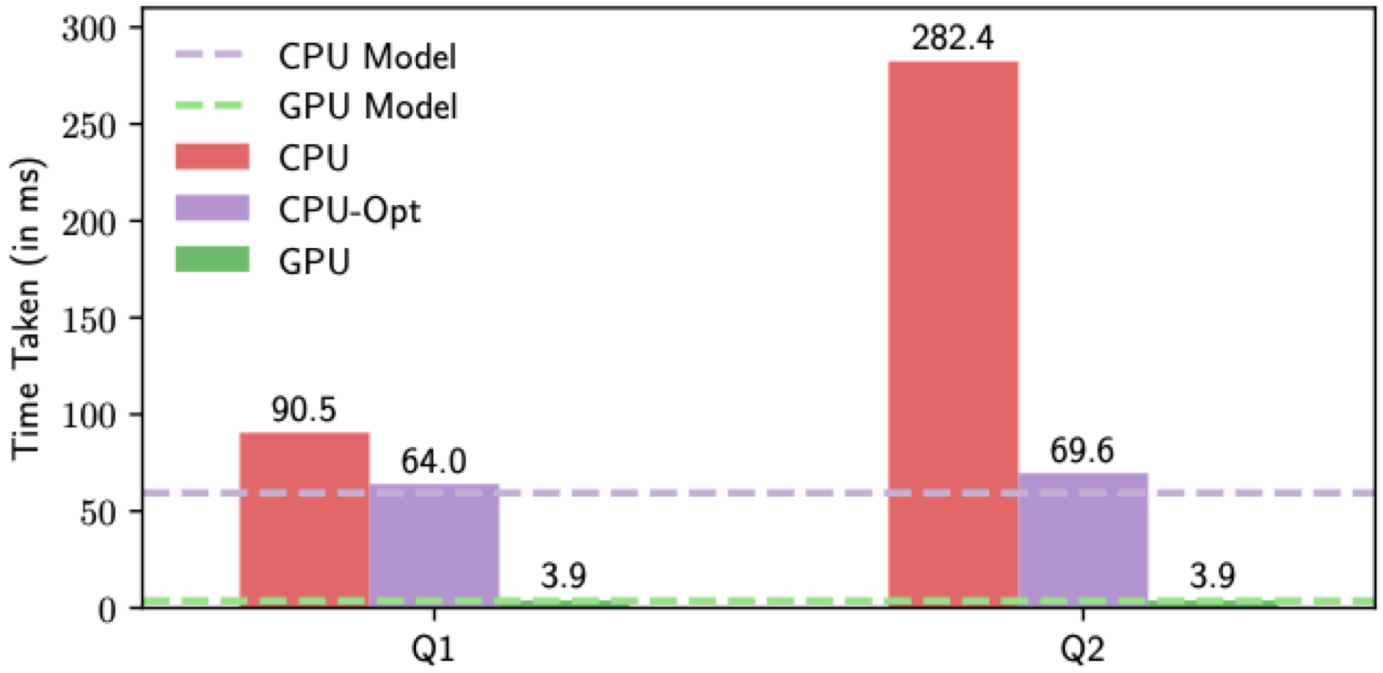
Crystal Library

Block-wide function: takes in a set of tiles as input, performs a specific task, and outputs a set of tiles

Primitive	Description
BlockLoad	Copies a tile of items from global memory to shared memory. Uses vector instructions to load full tiles.
BlockLoadSel	Selectively load a tile of items from global memory to shared memory based on a bitmap.
BlockStore	Copies a tile of items in shared memory to device memory.
BlockPred	Applies a predicate to a tile of items and stores the result in a bitmap array.
BlockScan	Co-operatively computes prefix sum across the block. Also returns sum of all entries.
BlockShuffle	Uses the thread offsets along with a bitmap to locally rearrange a tile to create a contiguous
BlockLookup	array of matched entries. Returns matching entries from a hash table for a tile of keys.
BlockAggregate	Uses hierarchical reduction to compute local aggregate for a tile of items.

Operators – Project

```
Q1: SELECT ax1 + bx2 FROM R;
Q2: SELECT σ(ax1 + bx2) FROM R;
```



CPU-Opt:

- Non-temporal writes
- SIMD

Efficient CPU/GPU implementations can saturate DRAM bandwidth

Operators – Select

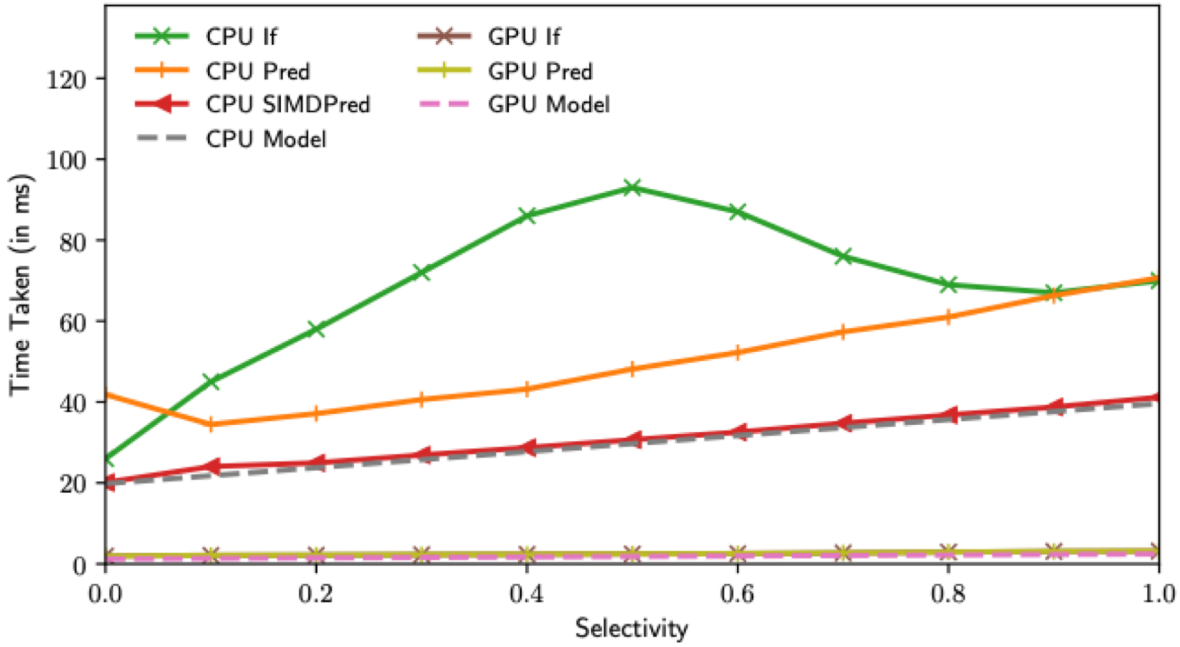
```
Q3: SELECT y FROM R WHERE y < v;
```

```
for y in R:
  if y < v
    output[cnt++] = v
```

(a) With branching

```
for y in R:
  output[i] = y
  cnt += (y>v)
```

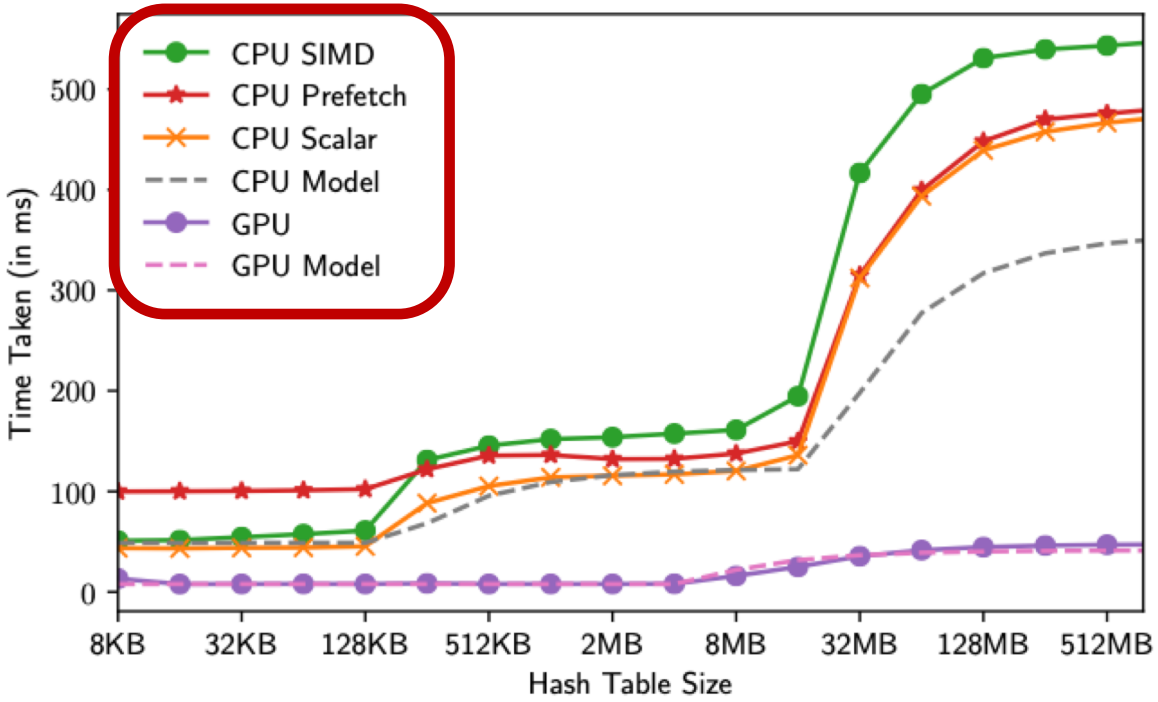
(a) With predication



Operators – Hash Join

```
Q4: SELECT SUM(A.v + B.v) AS checksum
    FROM A,B WHERE A.k = B.k
```

Latency-bound



Build phase: populate the hash table using tuples in one relation (typically the smaller relation)

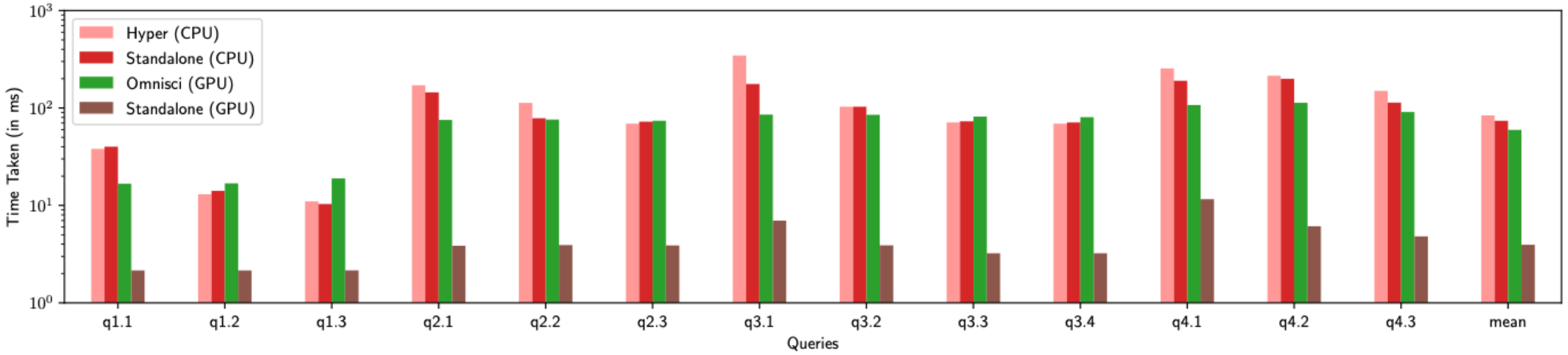
Probe phase: use tuples in the other relation to probe the hash table

Star-Schema Benchmark

Platform	CPU	GPU
Model	Intel i7-6900	Nvidia V100
Cores	8 (16 with SMT)	5000
Memory Capacity	64 GB	32 GB
L1 Size	32KB/Core	16KB/SM
L2 Size	256KB/Core	6MB (Total)
L3 Size	20MB (Total)	-
Read Bandwidth	53GBps	880GBps
Write Bandwidth	55GBps	880GBps
L1 Bandwidth	-	10.7TBps
L2 Bandwidth	-	2.2TBps
L3 Bandwidth	157GBps	-

Crystal-based implementations always saturate GPU memory bandwidth

GPU is on average **25X** faster than CPU



Cost Analysis

	Purchase Cost	Renting Cost (AWS)
CPU	$2-5K	$0.504 per hour
GPU	$CPU + 8.5K	$3.06 per hour

GPU is 25X faster than CPU

GPU is 6X more expensive than CPU

GPU is 4X more cost effective than CPU

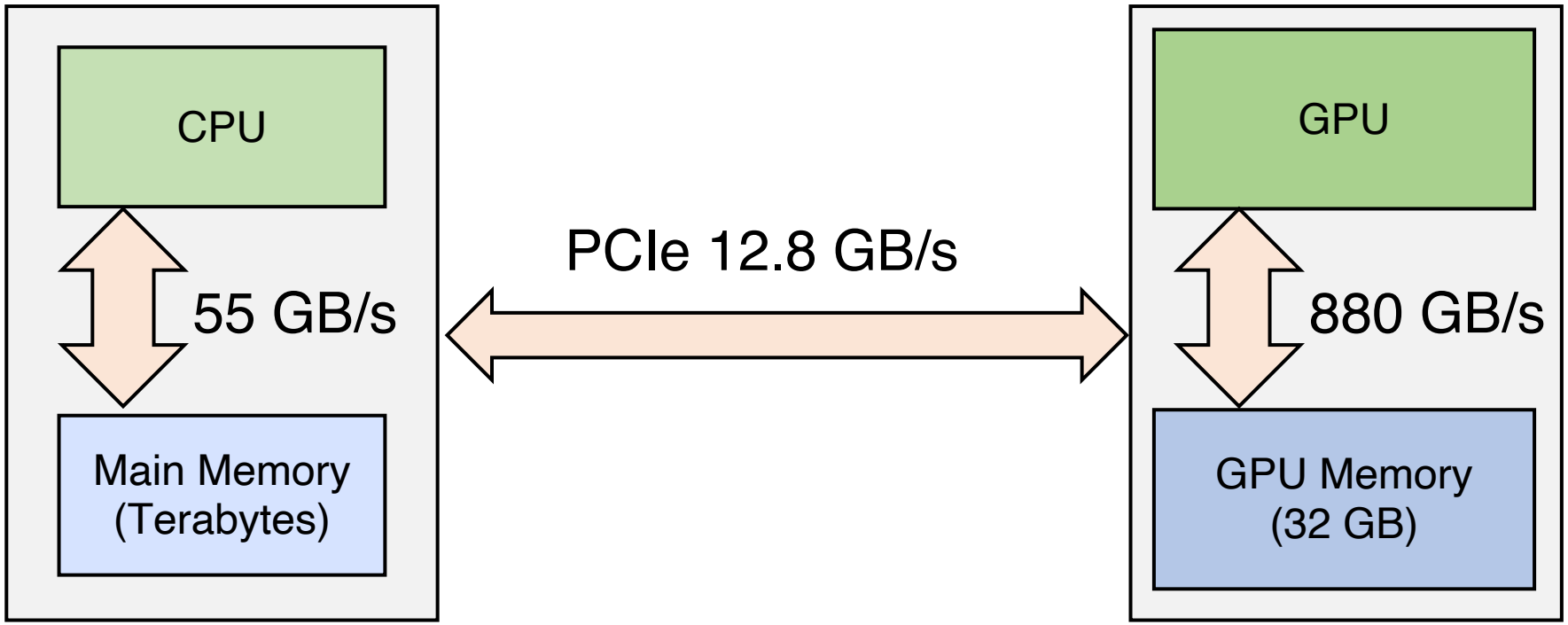
Future Work

Distributed GPUs + hybrid GPU/CPU

Data compression

Supporting string and array data type in GPU

Summary



Performance: GPU-only > CPU-only > coprocessor

Crystal: Tile-based execution model

GPUs are 25X faster and 4X more cost effective

GPU Database – Q/A

Does NVLink solve the PCIe bottleneck?

Will open-source the code soon

Overhead of loading data to GPU and transferring results back to CPU

What about updates/transactions?

Group Discussion

What is the advantages and disadvantages of executing transactions on GPUs?

Can you think of any solutions (either software or hardware) to overcome the problems of (1) limited PCIe bandwidth between CPU and GPU and (2) limited GPU memory capacity?

What are the main opportunities and challenges of deploying a database on heterogeneous hardware?

Before Next Lecture

Submit discussion summary to https://wisc-cs839-ngdb20.hotcrp.com

- **Deadline: Wednesday 11:59pm**

Submit review for

- Q100: The Architecture and Design of a Database Processing Unit