南方科技大学

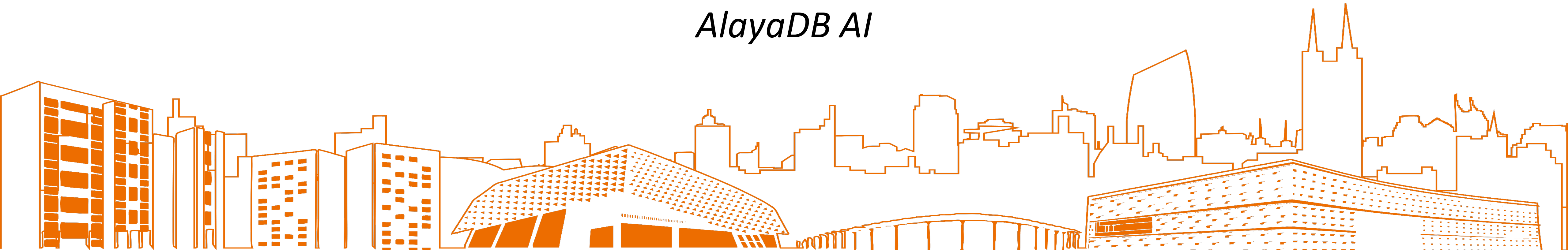
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

How Does Software Prefetching Work on GPU Query Processing?

Yangshen Deng, Shiwen Chen, Zhaoyang Hong, Bo Tang

Southern University of Science and Technology

AlayaDB AI



Existing GPU DBs (GPU as primary processor)

- What is the performance bottleneck?
  - *Bandwidth* of global memory
  - Achieving a higher bandwidth requires increasing the memory-level parallelism
- How do they achieve memory-level parallelism?
  - Simply rely on the *implicit hardware scheduling*

Question 1

Does

implicit hardware scheduling

efficiently utilize

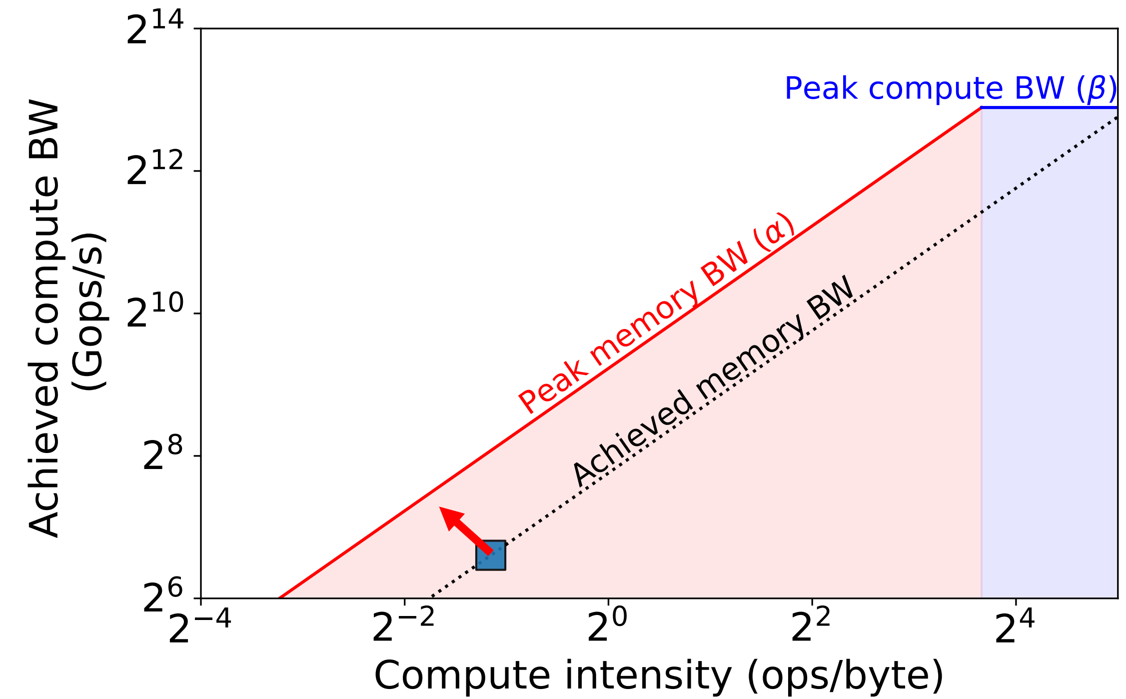
memory bandwidth?

Performance Analysis of BTree Search

Searching a GPU BTree with ***implicit hardware scheduling***

Results

- Bounded by memory bandwidth
- **LOW** bandwidth utilization



💡 **Implicit hardware scheduling is NOT good enough**

An Alternative: Explicit Software Prefetching

- Manually overlap compute operations and memory requests
- Could improve memory-level parallelism and achieve higher bandwidth
- Success in CPU databases but has not been studied in GPU databases

Question 2

How does *software prefetching* work on GPU query processing?

This Work

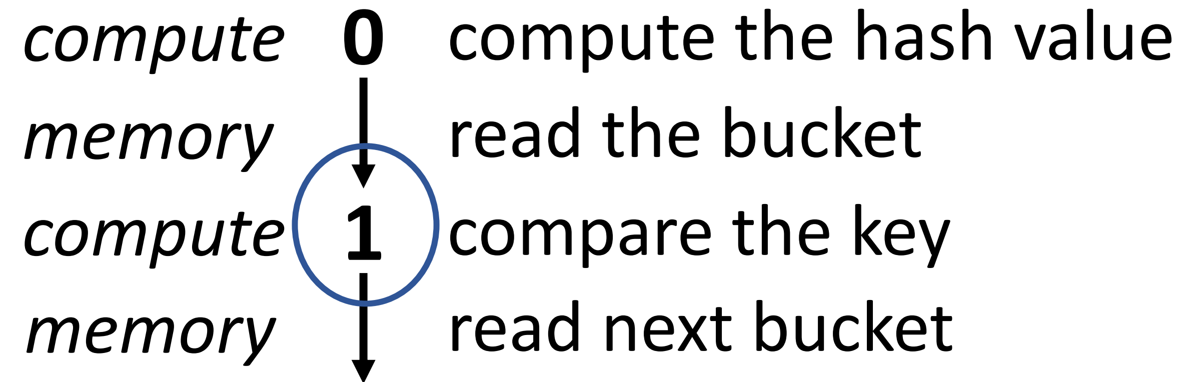
- Implements 4 existing prefetching algorithms on GPU
- Analyzes their performance with **hash join probe** and **BTree search**
- Proposes several optimizations
- Gives a list of guidelines

Prefetching Algorithms in CPU Databases

The processing of each tuple can be modeled as

a **code path** with dependent **memory accesses** and **compute operations**

E.g., hash join probe



Prefetching Algorithms in CPU Databases

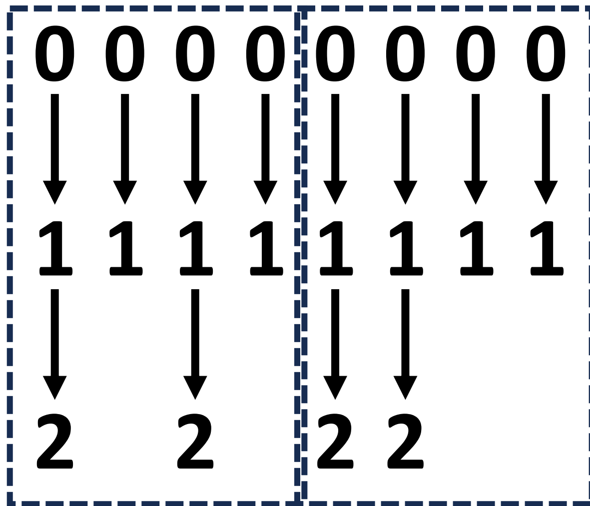
We study 4 prefetching algorithms previously proposed for CPU DB

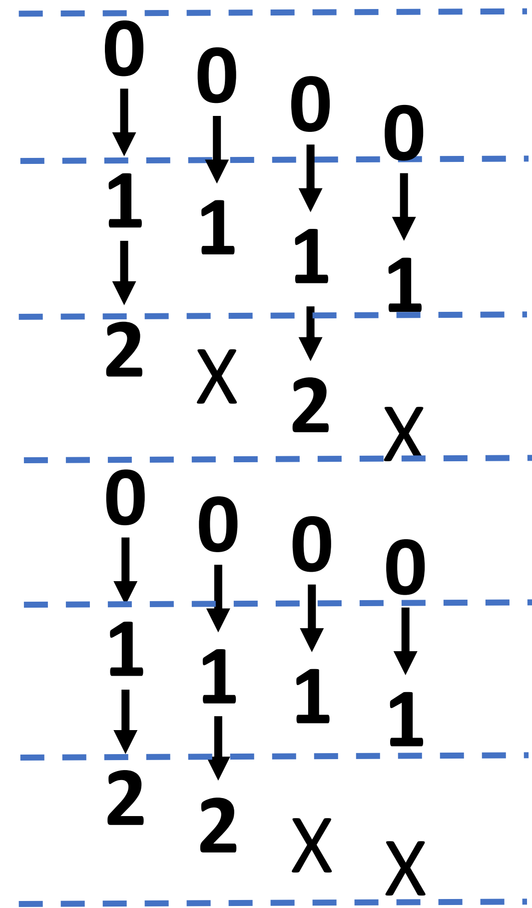
- Group Prefetch (GP) [1]
- Software Pipeline Prefetch (SPP) [1]
- Asynchronous Memory Access Chaining (AMAC) [2]
- Interleaved Multi-Vectorizing (IMV) [3]

Group Prefetch (GP)

An example of processing 8 tuples

An iteration processes 1 *code stage* of 4 tuples



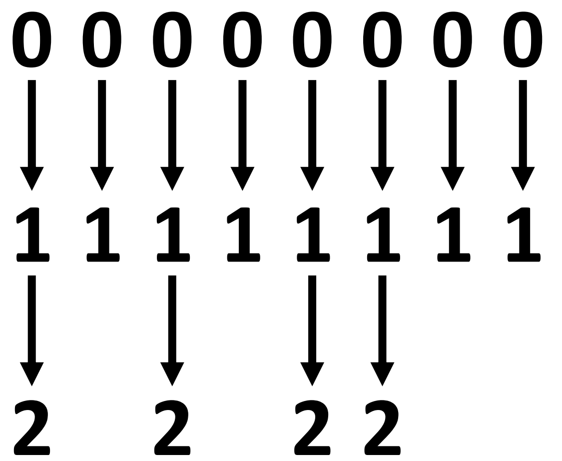


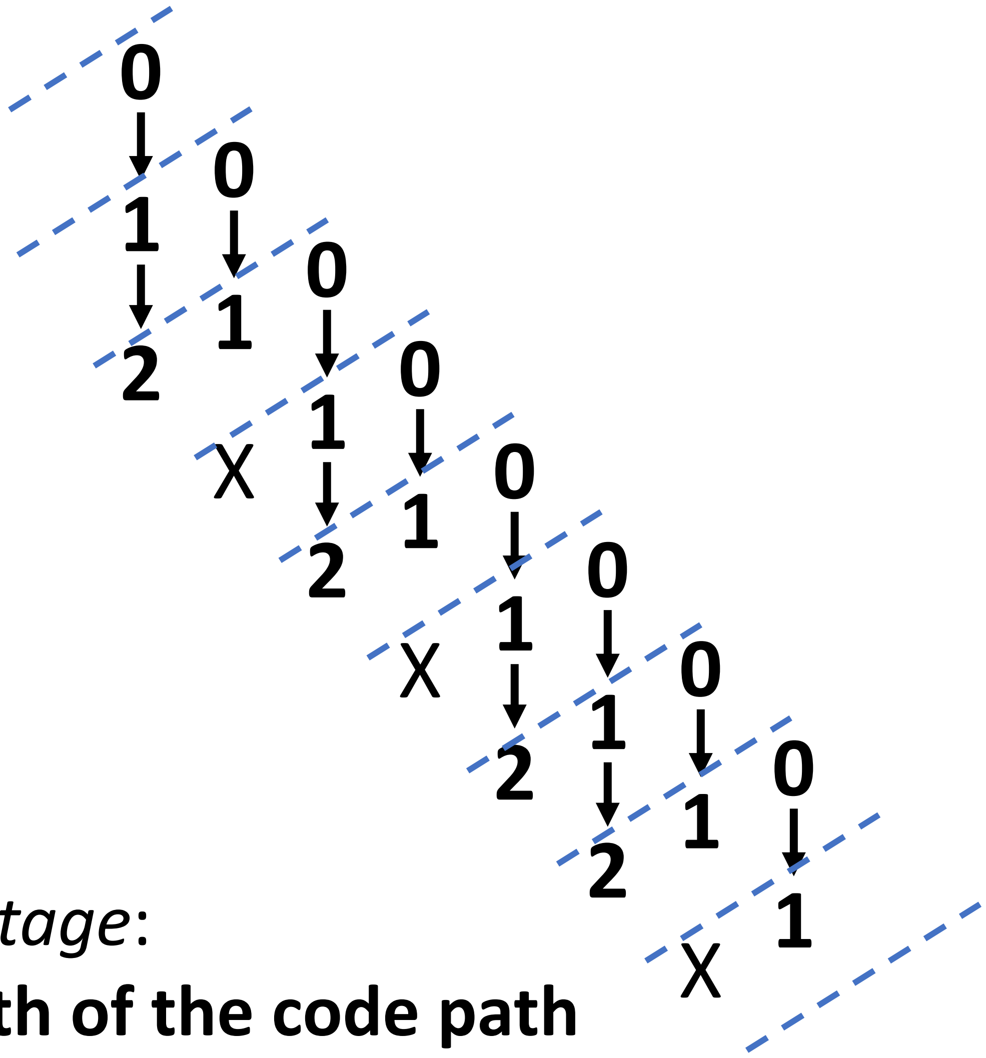
Needs to buffer the state of each *tuple* in an iteration

Software Pipelined Prefetch (SPP)

An example of processing 8 tuples

Each iteration processes all *code stages*





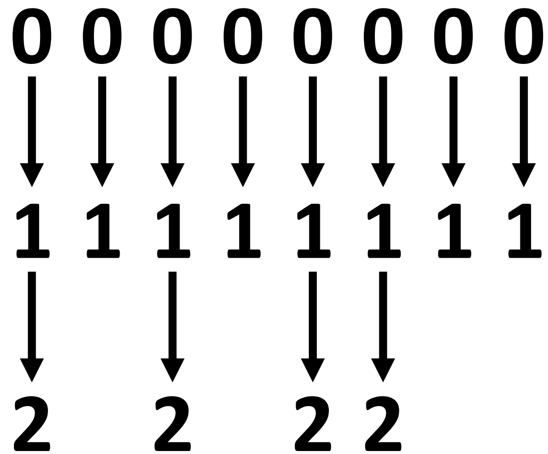
Needs to buffer the state of each *code stage*:

Buffer size grows linearly with the length of the code path

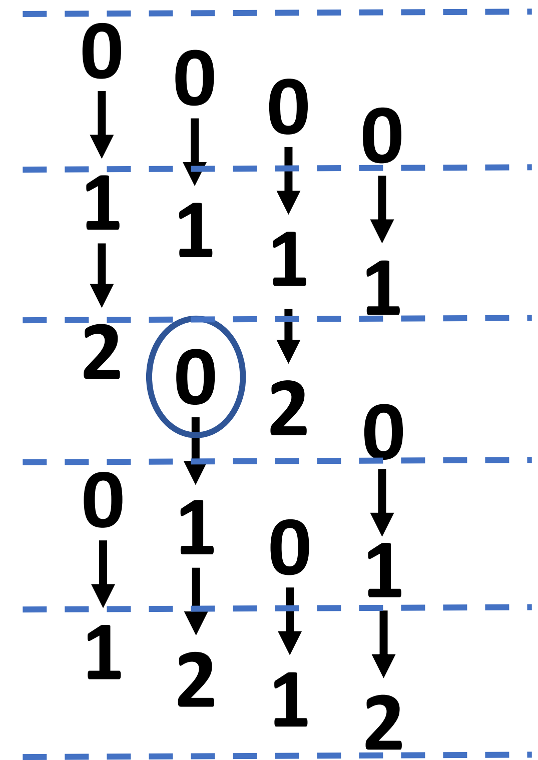
Asynchronous Memory Access Chaining (AMAC)

An example of processing 8 tuples

An iteration processes 1 *code stage* of 4 tuples



*Handle **divergent code paths** by dynamically filling the empty stage with a new stage*

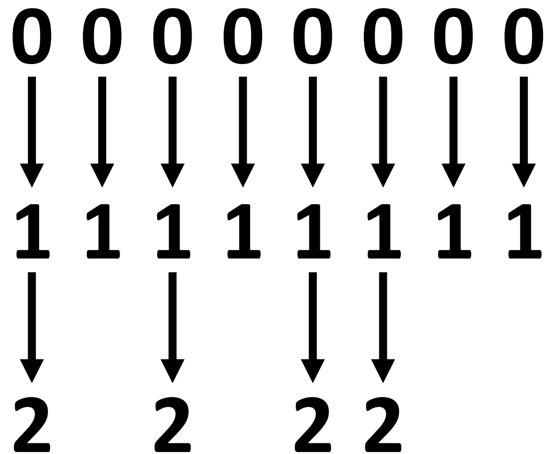


Needs to buffer the state of each *tuple* in an iteration

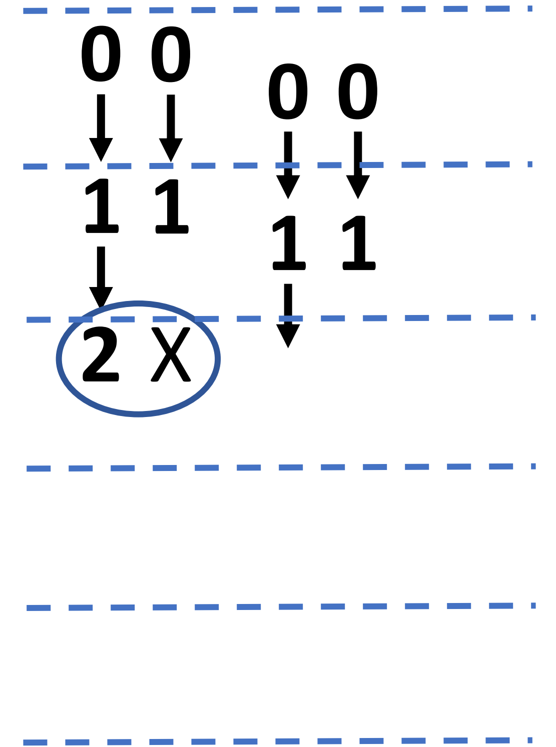
Interleaved Multi-Vectorizing (IMV)

An example of processing 8 tuples

An iteration processes 1 *code stage* of 2 vectors, each with 2 tuples



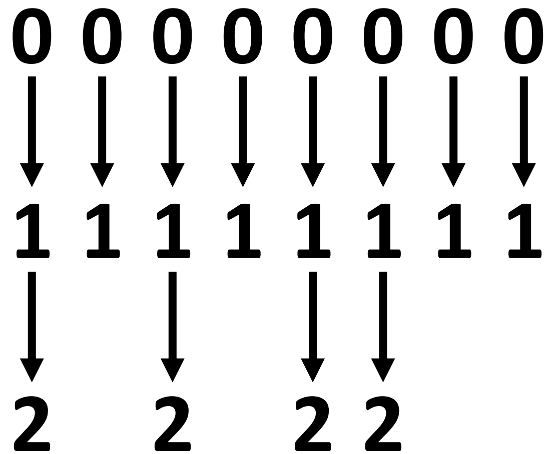
divergent states



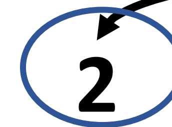
Interleaved Multi-Vectorizing (IMV)

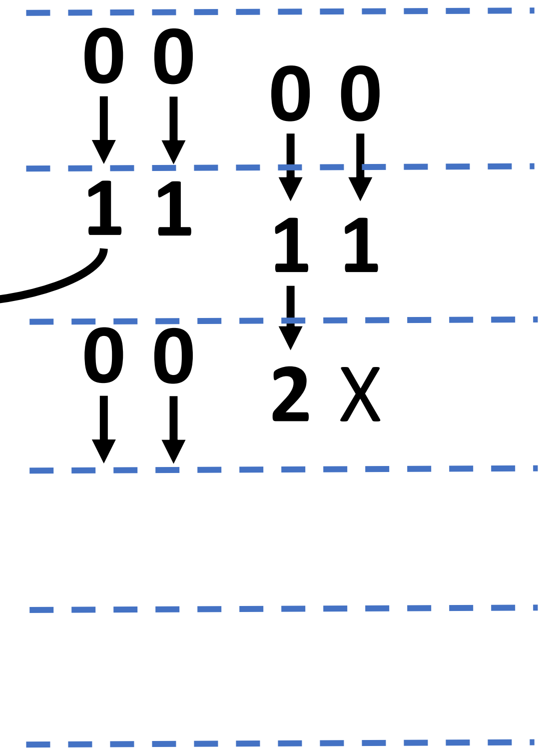
An example of processing 8 tuples

An iteration processes 1 *code stage* of 2 vectors, each with 2 tuples



*Buffer the **divergent states***

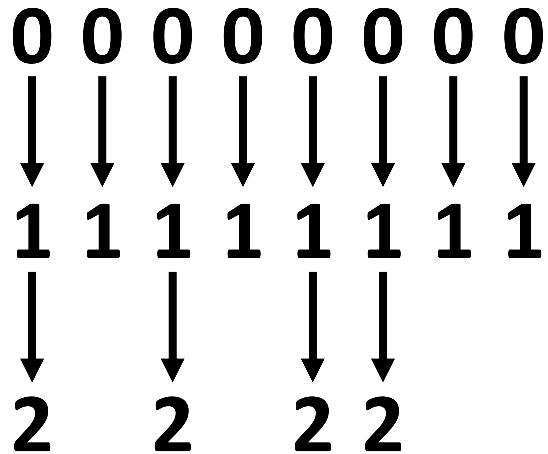




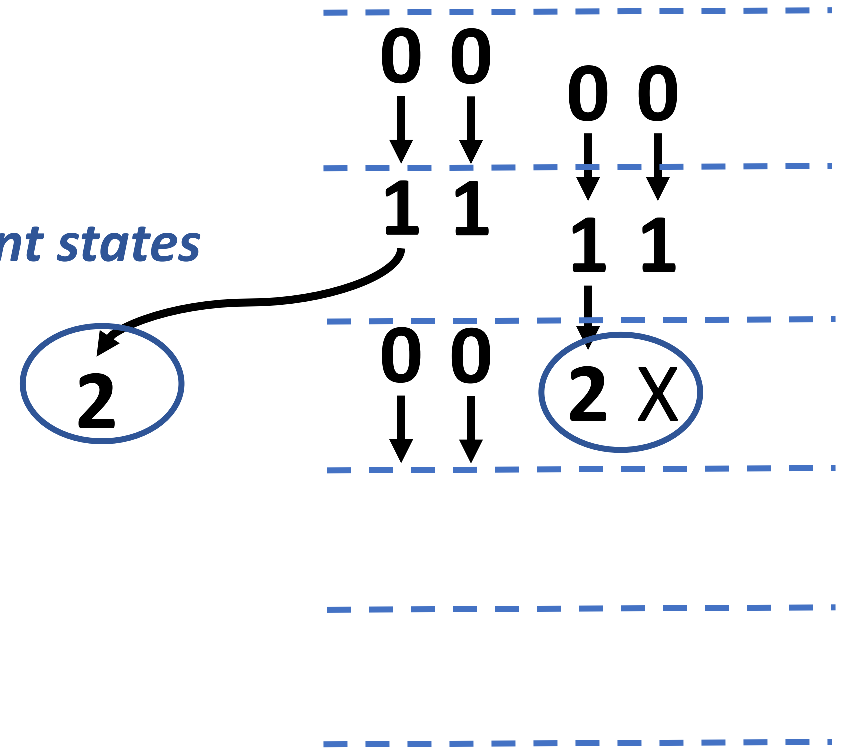
Interleaved Multi-Vectorizing (IMV)

An example of processing 8 tuples

An iteration processes 1 *code stage* of 2 vectors, each with 2 tuples



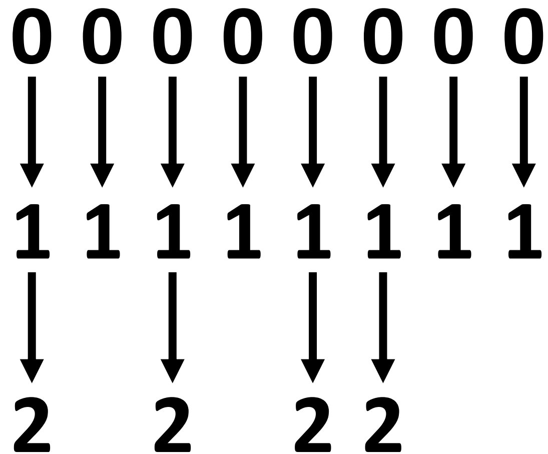
*Fill the **divergent states***



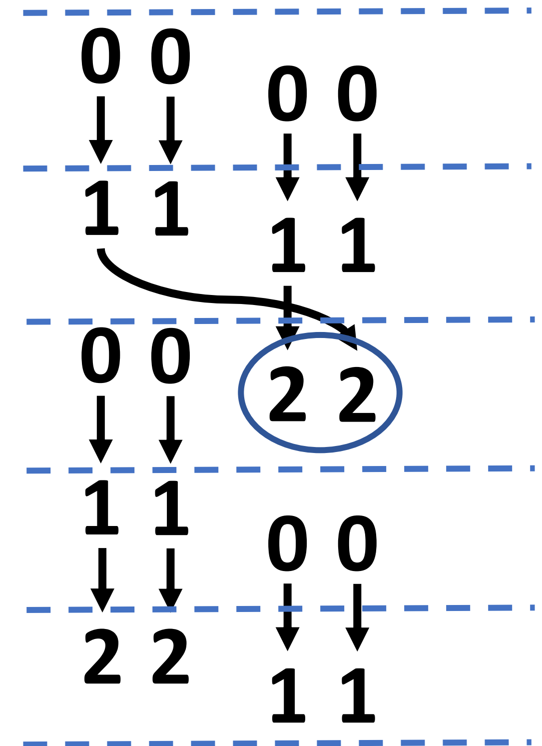
Interleaved Multi-Vectorizing (IMV)

An example of processing 8 tuples

An iteration processes 1 *code stage* of 2 vectors, each with 2 tuples



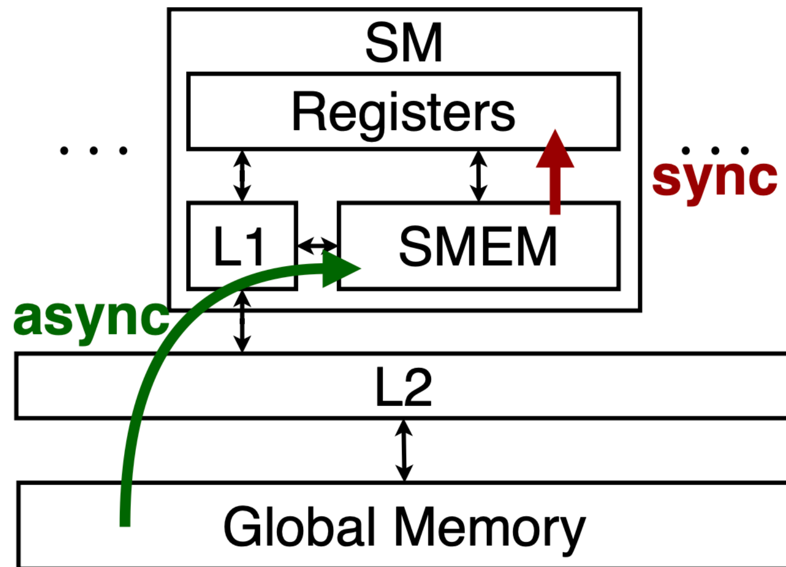
Fill the ***divergent states***



Prefetching APIs in GPU (CUDA)

cuda::memcpy_async

- Explicitly manage the cache (shared memory)
- Explicitly synchronize the prefetch request with an FIFO




```
P = cuda::make_pipeline()
cuda::memcpy_async()   D
P.producer_commit()    C
                       B
P.consumer_wait()      A
```

Implementation

For **GP**, **SPP**, and **AMAC**, just replace the prefetch APIs

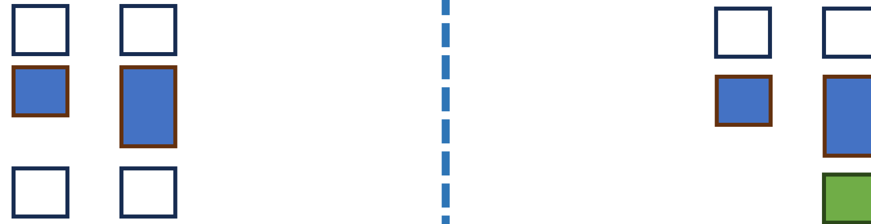
For **IMV**, need to re-implement the vector states reorganization

- It was proposed for SIMD execution on CPU

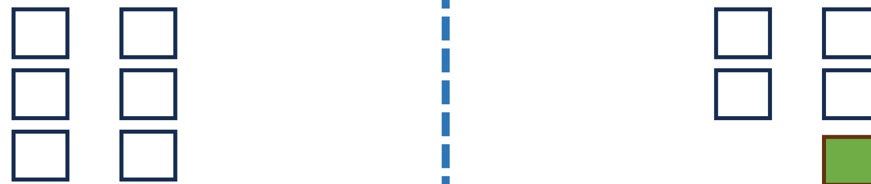
💡 We propose an efficient implementation for SIMT execution on GPU with ***warp primitives*** and ***shared memory***

Challenges 1: Divergence

w/ divergence inside a code stage



uniform code paths -- *divergent code paths*



w/o divergence inside a code stage

Challenges 1: Divergence

w/ divergence inside a code stage

BTree search
Searches inside a node finish with different steps

uniform code paths | *divergent code paths*

Hash Join Probe
When bucket chains have the same length

Hash Join Probe
When bucket chains have different lengths

w/o divergence inside a code stage

Solutions for Divergence

Divergent code paths

- ***AMAC*** fills empty code stage with new stages
- ***IMV*** reorganizes the divergent vector states

Divergence inside a code stage

- Existing algorithms do not solve this divergence

💡 Our optimization:

Enabling ***fewer threads in a warp*** to reduce divergence

Challenges 2: Cache Miss

Incurred by the frequent accesses to **state buffer**

Our optimization:

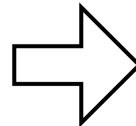
Explicitly caching states in ***shared memory*** with a ***coalesced layout***

```
struct state_t {
  Tuple s_tuple;
  Entry *next;
  id_t id;
};

state_t state[GROUP_SIZE];
While (...) {
  read(states[i].id);
  write(states[i].id);
  i++;
}
```

Implicitly stored in global memory

Cache miss



```
struct state_t {
  Tuple s_tuple[BLOCK_SIZE];
  Entry *next[BLOCK_SIZE];
  id_t id[BLOCK_SIZE];
};

__shared__ state_t state[];
While (...) {
  read(states[i].id[THREAD_ID]);
  write(states[i].id[THREAD_ID]);
  i++;
}
```

Explicitly stored in shared memory

No bank conflict & no cache miss

Experiments

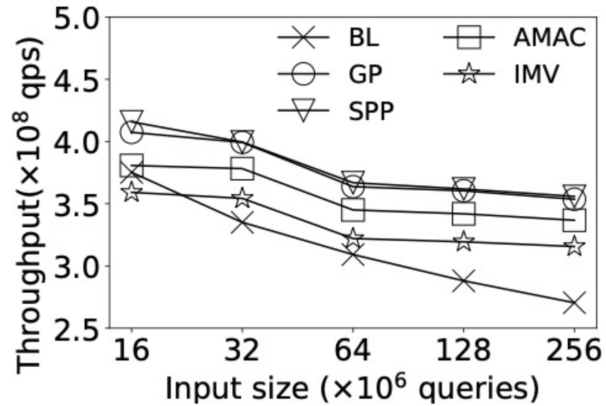
Hardware: NVIDIA A10 GPU, 72 SMs, 128 cores per SM.

Data: 4-byte key and 4-byte value

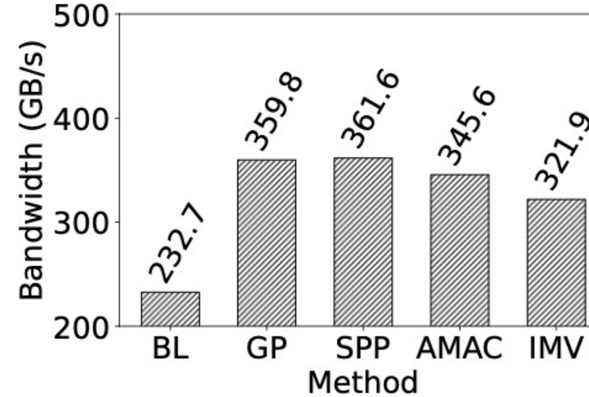
Tasks: hash join probe & BTree search

Baseline(BL): w/o prefetching, only hardware scheduling

Uniform Code Paths



(a) Throughput



(b) Achieved memory bandwidth

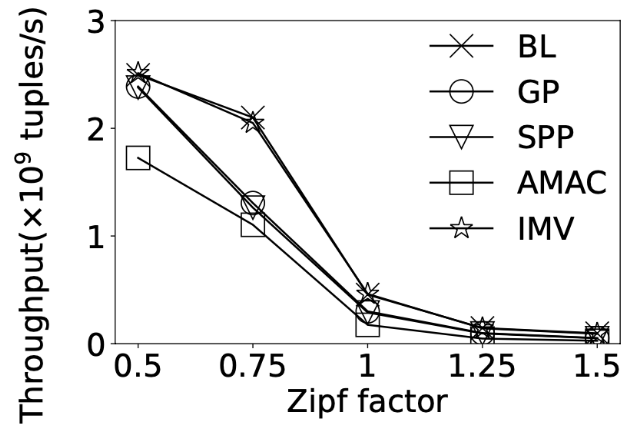
Figure 12: Performance evaluation of BTree search

- Software prefetching achieves higher bandwidth than hardware scheduling
- AMAC and IMV incur extra overhead in handling divergence

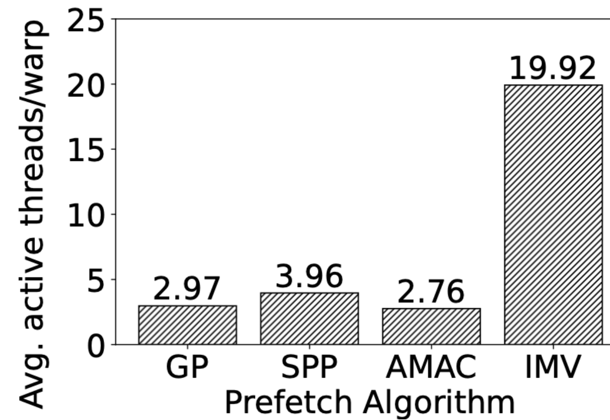
💡 **GP=SPP > AMAC > IMV > Baseline**

Divergent Code Paths

Hash join probe on skewed keys: bucket chains have different lengths



(a) Throughput



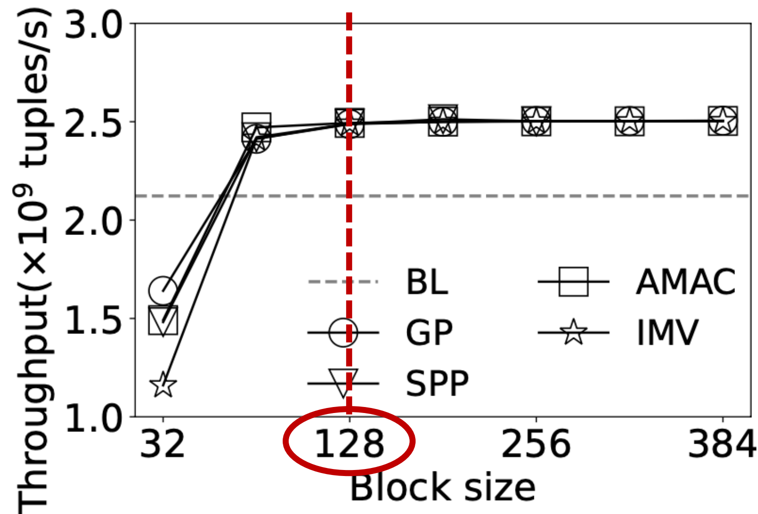
(b) Divergence in SIMT

- Both IMV and hardware scheduling can handle the divergence
- The way that AMAC handles divergence instead exacerbated the divergence in GPU

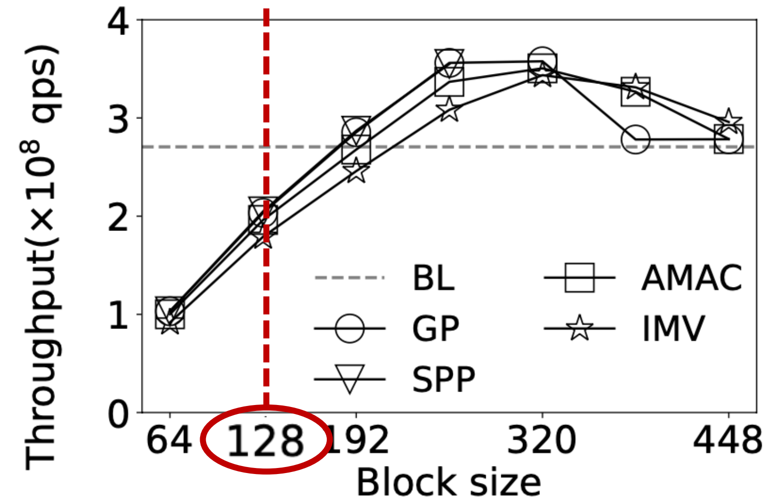
IMV=Baseline > GP=SPP > AMAC

Software Prefetching or Hardware Scheduling?

Each SM has 128 hardware cores (4 warp schedulers)
It benefits from hardware scheduling only when threads per block > 128



(a) Hash join probe



(b) BTree search

Figure 16: Impact of concurrent threads (72 blocks)

💡 **It is not black or white!**

Combining them together may get the best performance

Takeaways

- Software Prefetching WORKS in GPU query processing.
- For workloads with uniform code paths, use GP.
- For workloads with divergent code paths, use IMV.
- For workloads with divergence inside a code stage, enable a proper number of threads per warp.
- Make sure the states are cached in shared memory or registers.
- Combine software prefetching and hardware scheduling to get the best performance.

Thank you

Code: https://github.com/DBGroup-SUSTech/GPUDB-Prefetch

Reference

[1] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. 2004. Improving hash join performance through prefetching. ICDE.

[2] Onur Kocberber, Babak Falsafi, and Boris Grot. 2015. Asynchronous memory access chaining. VLDB.

[3] Zhuhe Fang, Beilei Zheng, and Chuliang Weng. 2019. Interleaved multi-vectorizing. VLDB.