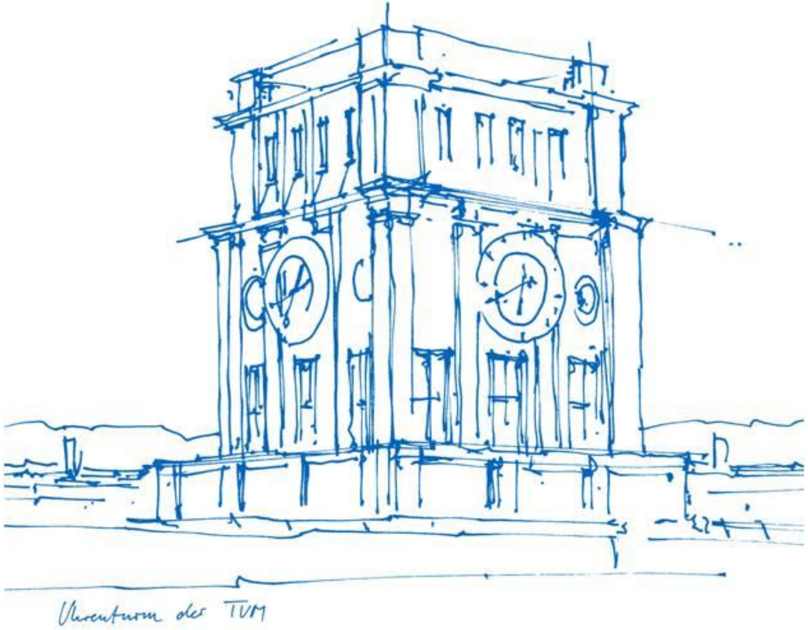
Parallel Query Processing on GPUs using Sub-operators

Master thesis

Artem Kroviakov

Lehrstuhl für Datenbanken

12. November 2024



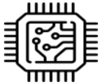
Motivation

- State-of-the-art CPU databases often hit the bandwidth boundary.
- GPUs offer much higher bandwidth.
- GPUs are ubiquitous in consumer and data center environments.
- AI boom suggests that GPUs are here to stay and will be actively developed, can we ride the GPU wave in databases?
- For analytics, why not sacrifice latency for bandwidth?

	CPU	**GPU**
RAM latency (cycles)	45	470
RAM bandwidth	460 GB/s	3.3 TB/s
ALUs (total)	~400	~15000
Load/Store Units (total)	~300	~3500

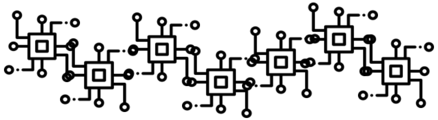
GPUs offer higher memory bandwidth and more compute

Hardware intuition behind GPUs



Goal: minimize instruction latency

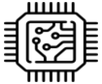
Goal: maximize instruction throughput



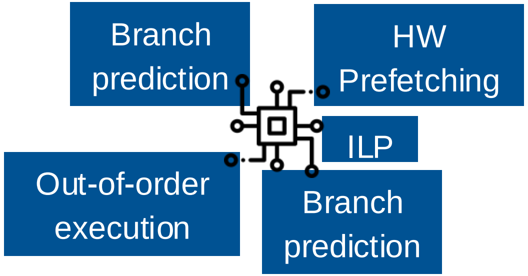
Memory coalescing

Multiple Schedulers

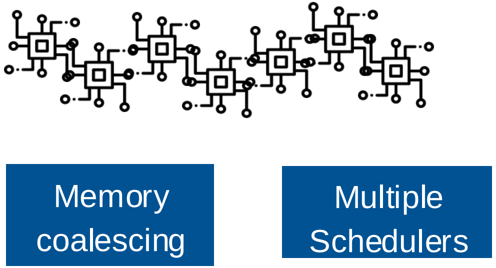
Hardware intuition behind GPUs



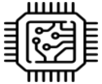
Goal: minimize instruction latency



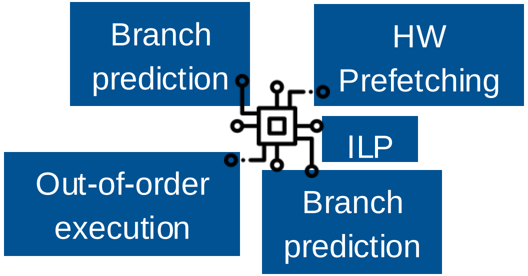
Goal: maximize instruction throughput



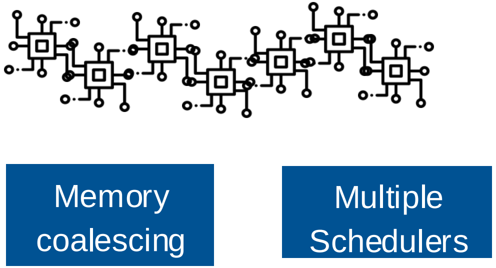
Hardware intuition behind GPUs



Goal: minimize instruction latency

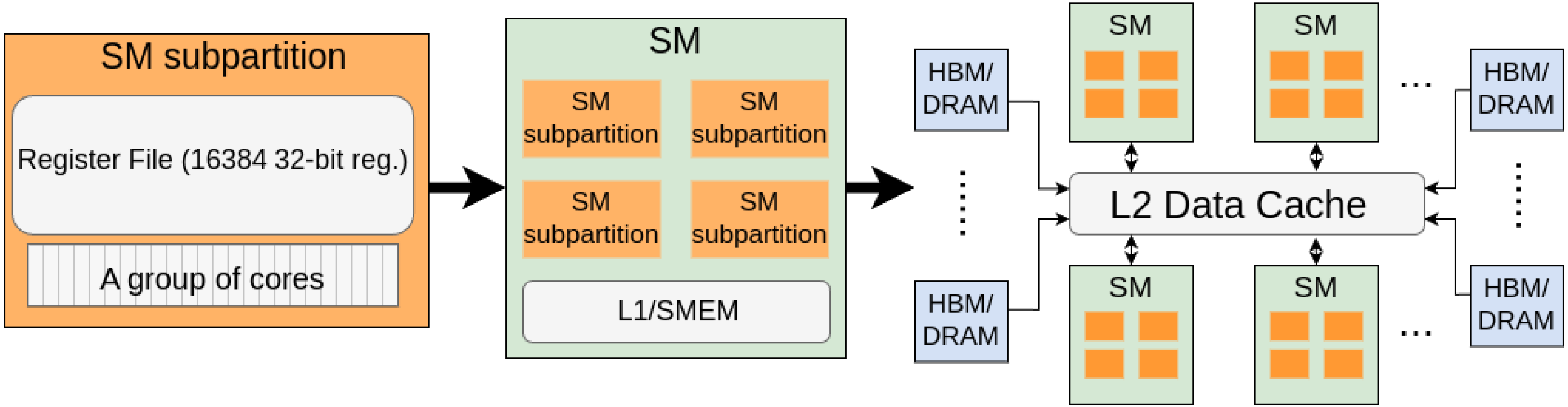


Goal: maximize instruction throughput

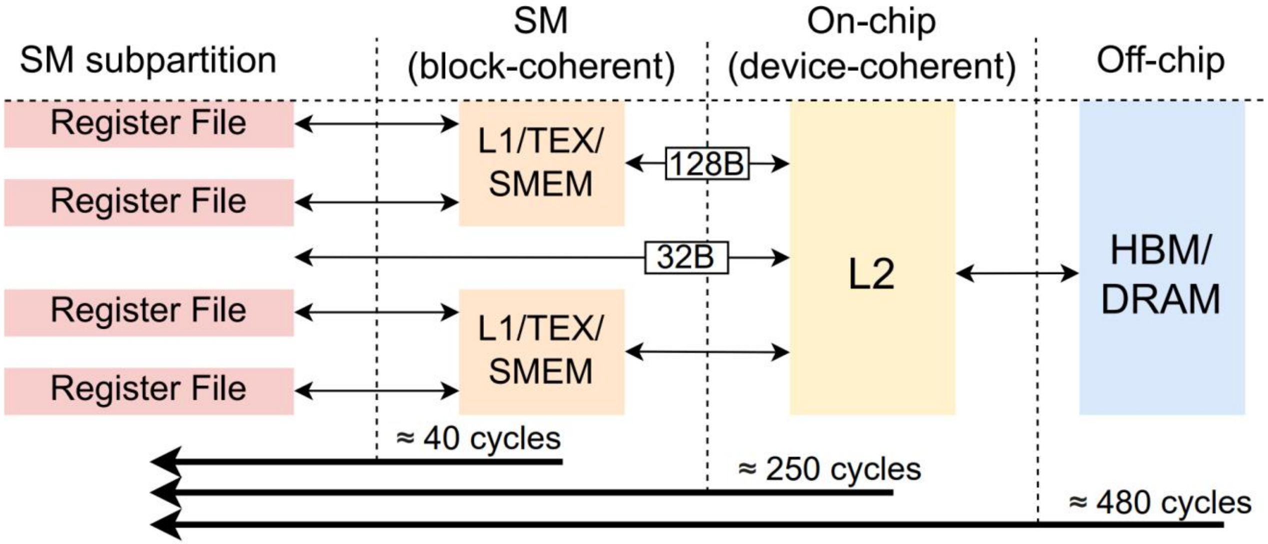


Last resort ← Optimization via SMT → First step

Quick GPU HW overview



Quick GPU HW overview



SMEM – a fast user-managed memory region (same HW unit as L1)

GPUs in databases

MapD, Omnisci, HeavyDB (mid 2010s):

- LLVM JIT compilation
- Huge code base
- Many physical operators
- Multi-GPU support

TQP (2022):

- Leverage PyTorch tensor runtime
- Low effort
- Ok-ish performance

Crystal, Crystal-opt (2022):

- Hand-written SSB queries
- Vectorization
- Fastest SSB runner on GPUs

GPUs in databases

MapD, Omnisci, HeavyDB (mid 2010s):
- LLVM JIT compilation
- Huge code base
- Many physical operators
- Multi-GPU support

TQP (2022):
- Leverage PyTorch tensor runtime
- Low effort
- Ok-ish performance

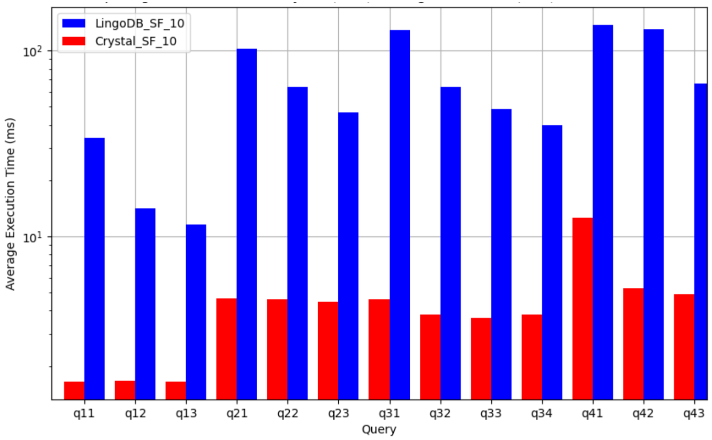
Crystal, Crystal-opt (2022):
- Hand-written SSB queries
- Vectorization
- Fastest SSB runner on GPUs

Idea

Approximate Crystal's hand-written performance while preserving HeavyDB's generality by leveraging the existing state-of-the-art query engine.

Step 1: Gain insights from Crystal – CPU vs. GPU

Good bandwidth utilization means that the query time is proportional to the bandwidth limits of devices. For CPUs and GPUs it is an order of magnitude.



Step 1: Crystal - Overview

Fastest GPU runner for simplified (numeric types only) SSB:

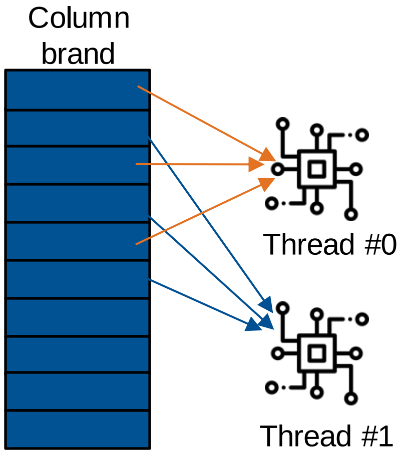
- Hand-crafted queries in CUDA C++

- Vectorized execution

```
int brand[ITEMS_PER_THREAD];
BlockLoad<..., ITEMS_PER_THREAD>(...
BlockProbeAndPHT_2<..., ITEMS_PER_THREAD>(...
```

- Collision free hash tables, tricky hash functions

```
int hash = (brand[ITEM] * 7 + (year[ITEM] - 1992)) % ((1998-1992+1) * (5*5*40));
```



Parallelism – vector per thread
Vector – 4 elements (in registers)

Step 1: Gain insights from Crystal – Compiled vs. Vectorized

Could the GPU's SMT-oriented model be enough to hide latencies like vectorization, even when running "compiled" queries?

Step 1: Gain insights from Crystal – Compiled vs. Vectorized

Could the GPU's SMT-oriented model be enough to hide latencies like vectorization, even when running "compiled" queries?

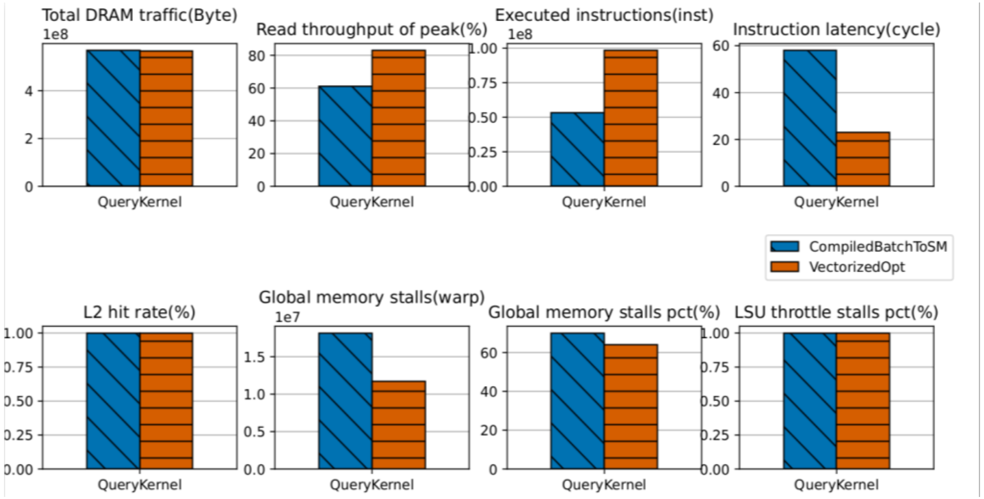
Not really.

Some queries are up to 30% slower.

Step 1: Gain insights from Crystal - Compiled vs. Vectorized

Compiled vs vectorized:

- Less instructions, but higher instruction latency.
- GPU's SMT is not enough to fully cover latency.
- Higher selectivity negates vectorization benefit.



SSB Q1.1
Scan+Filter+Sum Reduction.
Low selectivity.

Step 1: Gain insights from Crystal – Register usage

Table 4.1: Register usage for selected kernels/"pipelines".

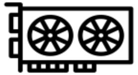
	Vectorized	Compiled
Q11	26	26
Q21 (probe)	28	19
Q21 (build date)	26	19
Q21 (build partkey)	22	16
Q31 (probe)	32	21

Register usage – why is it so important?

GPU performance comes from the massive SMT, but how do we achieve it?

Q31 (probe)	32	21
Q31 (build customer)	22	16

There is a limit on simultaneously active threads for a kernel.
The limit depends on 3 parameters:



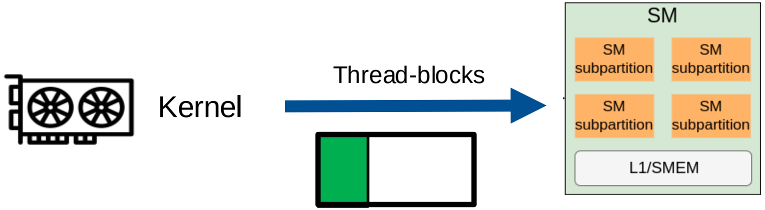
Kernel

Register usage – why is it so important?

Q31 (probe)	32	21
Q31 (build customer)	22	16

GPU performance comes from the massive SMT, but how do we achieve it?

There is a limit on simultaneously active threads for a kernel.
The limit depends on 3 parameters:



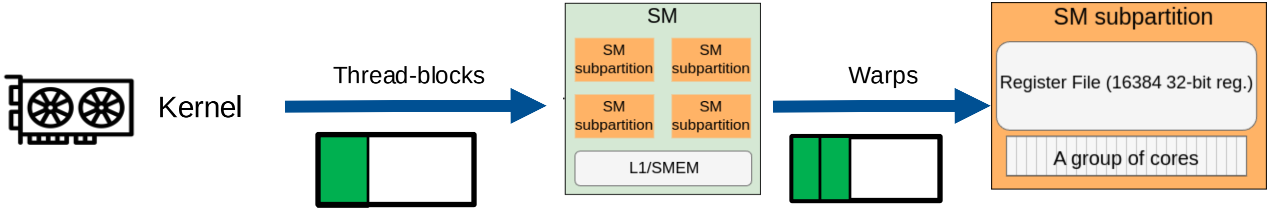
SMEM usage
A thread-block requires N bytes of SMEM,
how many N's can SMEM hold at once?

Register usage – why is it so important?

Q31 (probe)	32	21
Q31 (build customer)	22	16

GPU performance comes from the massive SMT, but how do we achieve it?

There is a limit on simultaneously active threads for a kernel.
The limit depends on 3 parameters:



SMEM usage
A thread-block requires N bytes of SMEM, how many N's can SMEM hold at once?

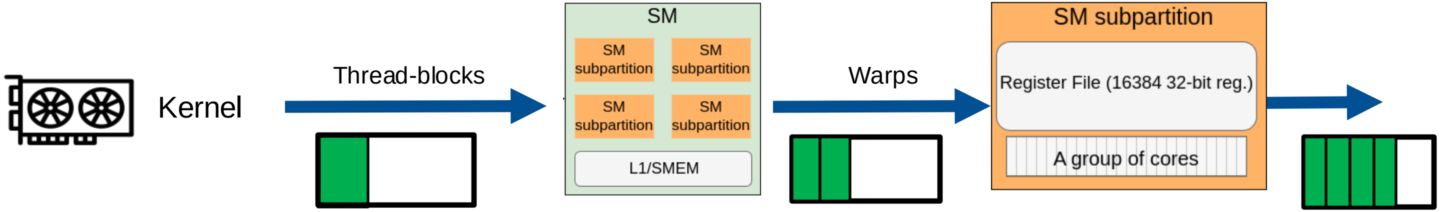
SM Scheduler slots
One SM can sustain at most 64 warps

Register usage – why is it so important?

Q31 (probe)	32	21
Q31 (build customer)	22	16

GPU performance comes from the massive SMT, but how do we achieve it?

There is a limit on simultaneously active threads for a kernel.
The limit depends on 3 parameters:



SMEM usage
A thread-block requires N bytes of SMEM, how many N's can SMEM hold at once?

SM Scheduler slots
One SM can sustain at most 64 warps

Register demand of a thread

Good scenario:
32 registers
No SMEM usage
=
Max. 2048 active threads per SM

Bad scenario:
150 registers
No SMEM usage
=
Max. 416 active threads per SM

How to map a batch to GPU?

HeavyDB – each batch is processed by the entire GPU.

- Prefers huge batches, default size is 32M rows.
- Low pressure on the bookkeeping infrastructure.

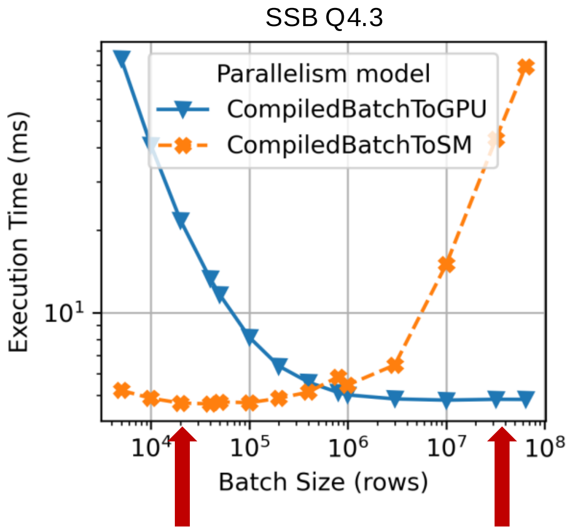
Crystal – each "batch" is processed by a thread-block.

- Prefers small batches (512 rows in the original Crystal).
- A column is linearly stored in memory, threads "slice" into it.

Approach	Thread Blocks : Batches	Intra-batch iteration range
HeavyDB	1 : **N**	[0, #Total threads)
Crystal	1 : 1	[0, #**TB** threads)

Too large batches

Too small batches



CPU morsel-driven

HeavyDB

How to map a batch to GPU?

HeavyDB – each batch is processed by the entire GPU.

- Prefers huge batches, default size is 32M rows.
- Low pressure on the bookkeeping infrastructure.

Crystal – each "batch" is processed by a thread-block.

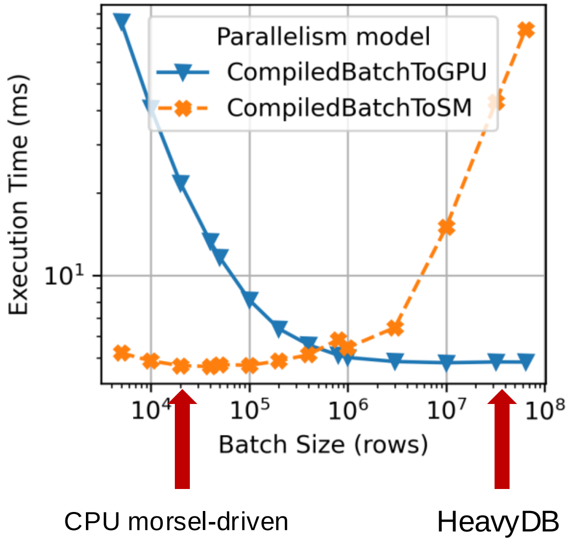
- Prefers small batches (512 rows).
- A column is linearly stored in memory, threads "slice" into it.

Approach	Thread Blocks : Batches	Intra-batch iteration range
HeavyDB	1 : **N**	[0, #Total threads)
Crystal	1 : 1	[0, #**TB** threads)
Mix	1 : **N**	[0, #**TB** threads)

➡ Too large batches

➡ Too small batches

SSB Q4.3



Step 2: More general query processing

Crystal has some simplifications:

- Known cardinalities
- Query-specific collision-free hash functions
- Primitive types for reductions

➡

Almost an unrealistic scenario in databases

+

Technical limitations of GPUs

(e.g., a thread stack variable is inaccessible to other threads, except for shuffles which are up to 8B)

To achieve a more general query processing, we need:

- General dynamic data structures
- Suitable results representation for generic merge logic

Step 2: More general query processing

Crystal has some simplifications:

- Known cardinalities
- Query-specific collision-free hash functions
- Primitive types for reductions

➡

Almost an unrealistic scenario in databases

+

Technical limitations of GPUs

(e.g., a thread stack variable is inaccessible to other threads, except for shuffles which are up to 8B)

To achieve a more general query processing, we need:

- General dynamic data structures ➡ Device heap
- Suitable results representation for generic merge logic ➡ Kernel-local concept

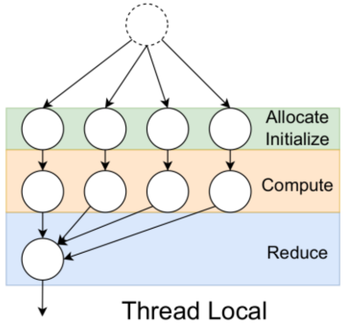
Step 2: More general queries – Kernel local

We must be able to easily access composite thread-local results.

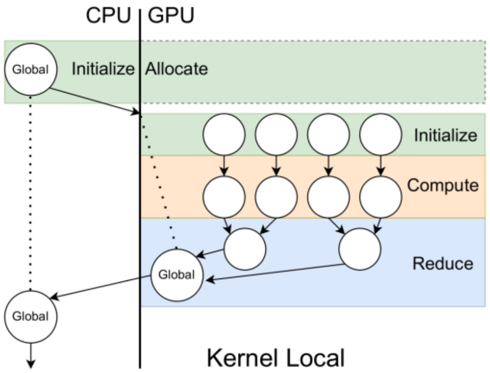
Kernel Local – let each kernel have a <u>global</u> memory allocation big enough to fit results at some locality level.

A different address space (not the same as thread's stack) makes the results accessible by a pointer from any thread.

Locality levels that reflect GPU hierarchy:
1. Thread
2. Warp (max. 32 Threads)
3. Thread Block (max. 32 Warps)



CPU morsel-driven



Step 2: More general queries – Device Heap

malloc() is regarded as a bad practice for GPUs.

- High latency
- Easily congested

Step 2: More general queries – Device Heap

malloc() is regarded as a bad practice for GPUs.

- High latency ➡ Sufficient parallelism
- Easily congested ➡ Proper locality level

Step 2: More general queries – Device Heap

malloc() is regarded as a bad practice for GPUs.

- High latency → Sufficient parallelism
- Easily congested → Proper locality level

How others deal with dynamic states?

HeavyDB: *HyperLogLog, Prefix sum*

TQP: *Prefix sum*

HyperLogLog:

1. One pass to *estimate* cardinality
2. Allocate
3. Second pass to fill allocation

Step 2: More general queries – Device Heap

malloc() is regarded as a bad practice for GPUs.

- High latency ➡ Sufficient parallelism
- Easily congested ➡ Proper locality level

How others deal with dynamic states?

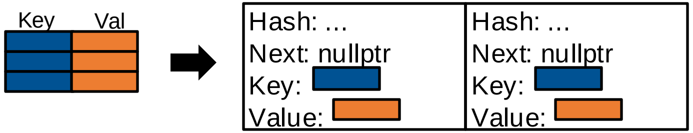
HeavyDB: *HyperLogLog, Prefix sum*

TQP: *Prefix sum*

HyperLogLog:

1. One pass to *estimate* cardinality
2. Allocate
3. Second pass to fill allocation

Workload: filter columns and fill a dynamic vector of entries



Step 2: More general queries – Device Heap

malloc() is regarded as a bad practice for GPUs.

- High latency ➡ Sufficient parallelism
- Easily congested ➡ Proper locality level

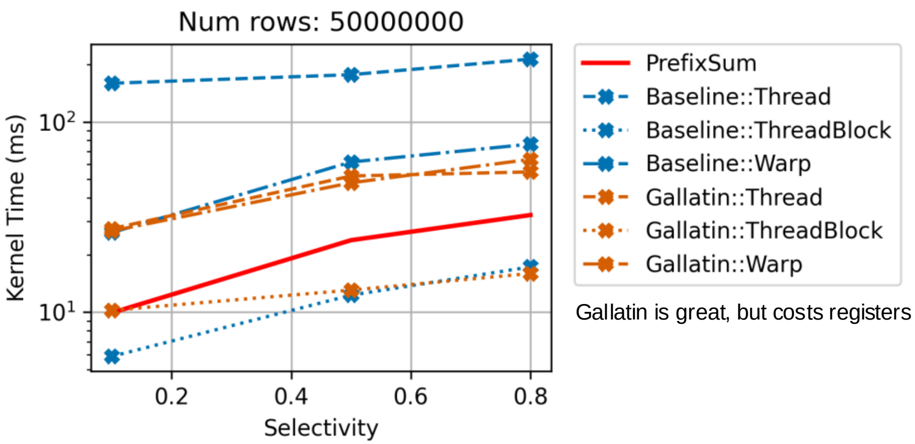
How others deal with dynamic states?

HeavyDB: *HyperLogLog, Prefix sum*

TQP: *Prefix sum*

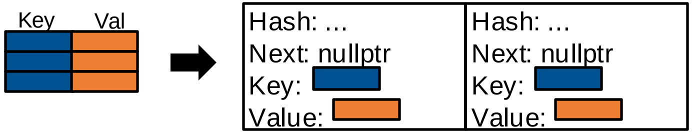
HyperLogLog:

1. One pass to *estimate* cardinality
2. Allocate
3. Second pass to fill allocation

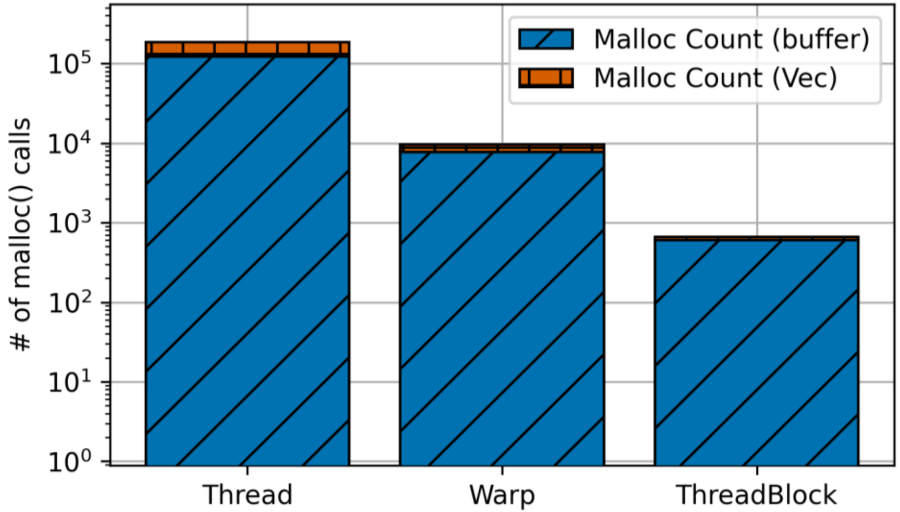


Gallatin is great, but costs registers

Workload: filter columns and fill a dynamic vector of entries



Step 2: More general queries – Device Heap



Going down by one level leads to an order of magnitude more allocations

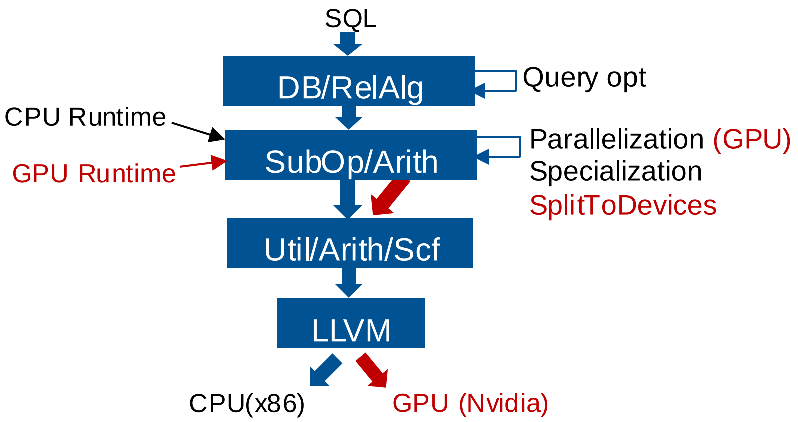
LingoDB (GPU extension)

- Prototype: replicate LingoDB's Q4.1 code in CUDA C++.
- Main problem: random global memory access during probing.

	Build	Probe-Aggregate
Crystal-opt	~1ms	~7ms
Prototype	~2ms	~20ms

LingoDB is a CPU state-of-the-art compiled analytical database that uses MLIR.

Multi-level IR – can pick suitable abstraction layer for modifications.

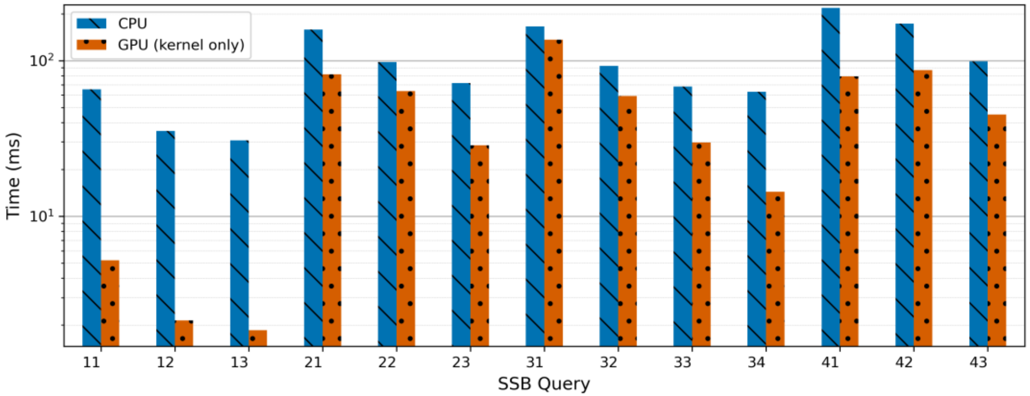


LingoDB (GPU extension) - Results

Simple queries (Q1.X) are straightforward to implement, but:

- High register usage: 72 vs. 26 in Crystal
- More logic: e.g., nullables

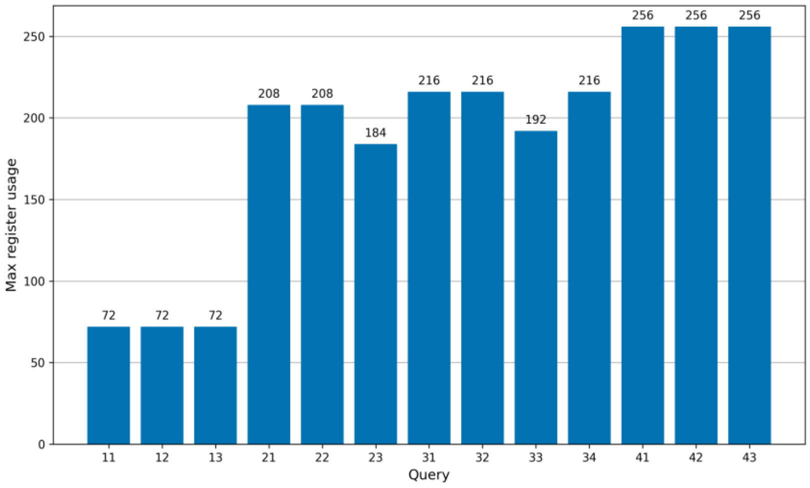
SF10



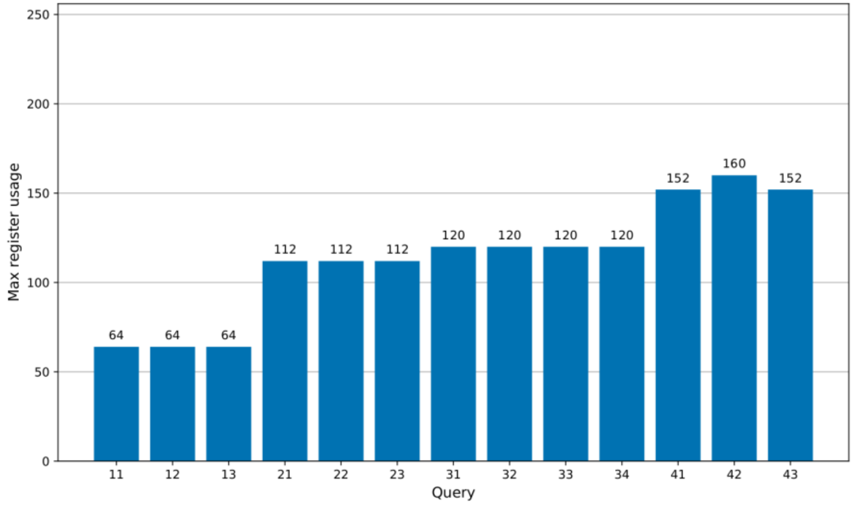
Complex queries have high register pressure:

- LingoDB likes to use i64 where i32 suffices.
- Runtime functions can have big stack and are not inlined (ABI calls cost registers).

LingoDB (GPU extension) - Hurdles



RTX 2060: sm_75



RTX A5000: sm_86

Higher register pressure = less threads active at once = less ability to hide latency = slower execution.

Conclusion

- Vectorization on GPU is faster when you have room for it.
- Even simple queries, produced by a real-world database engine can incur register pressure on GPUs.
- CPU and GPU can agree on the batch size without perf drawbacks on either side.
- Heap-based dynamic structures enable 2x performance benefit.
- Codegen infrastructure can be shared between two devices, no need for a whole new engine.

Future work

- There are multiple heap allocators for GPUs, evaluate them for database workloads. Do we need our own?
  We would like to have:
    1. low-register-cost coarse grained allocations
    2. fast free()
    3. latency is not so crucial as long as parallelism absorbs it.

- Is i64 needed in all of its uses? Can we inline runtime bitcode? Split complex pipelines?

- How to abort on heap overflow?

- How good chaining really is for HTs on GPUs? Compare LingoDB's chained hash table against HeavyDB's open addressing.

```
%5 = llvm.mlir.constant(10248 : index) : i64
%6 = llvm.mlir.constant(0 : index) : i64

%21 = nvvm.read.ptx.sreg.tid.x : i32
%22 = llvm.sext %21 : i32 to i64

%32 = nvvm.read.ptx.sreg.ntid.x : i32
%33 = llvm.sext %32 : i32 to i64

%34 = nvvm.read.ptx.sreg.nctaid.x : i32
%35 = llvm.sext %34 : i32 to i64
```

Index as i64: we have just wasted 5 registers

```
%116 = arith.extsi %55 : i32 to i128
```

i128 could be used more conservatively