Multi-threaded Queries

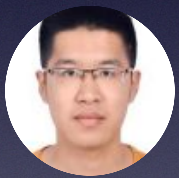
Intra-Query Parallelism in LLVM

Multithreaded Queries

Intra-Query Parallelism in LLVM

Yang Liu

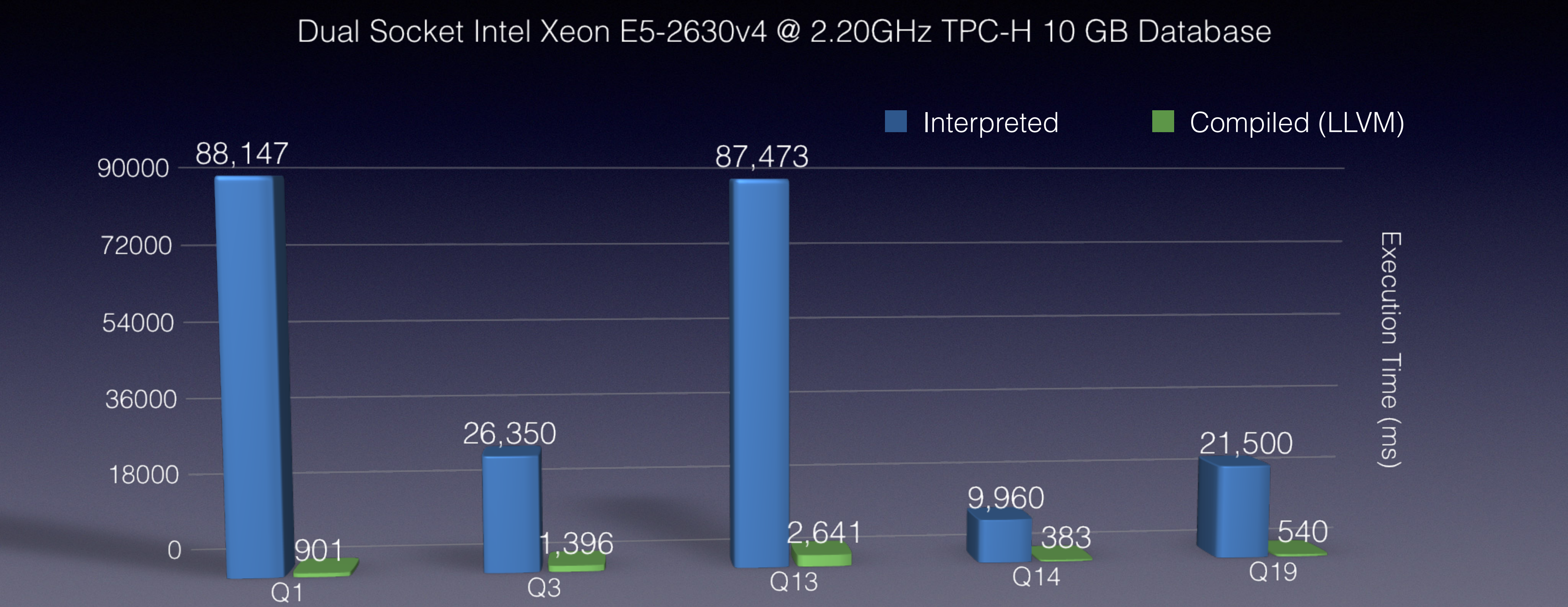
Tianqi Wu



Hao Li

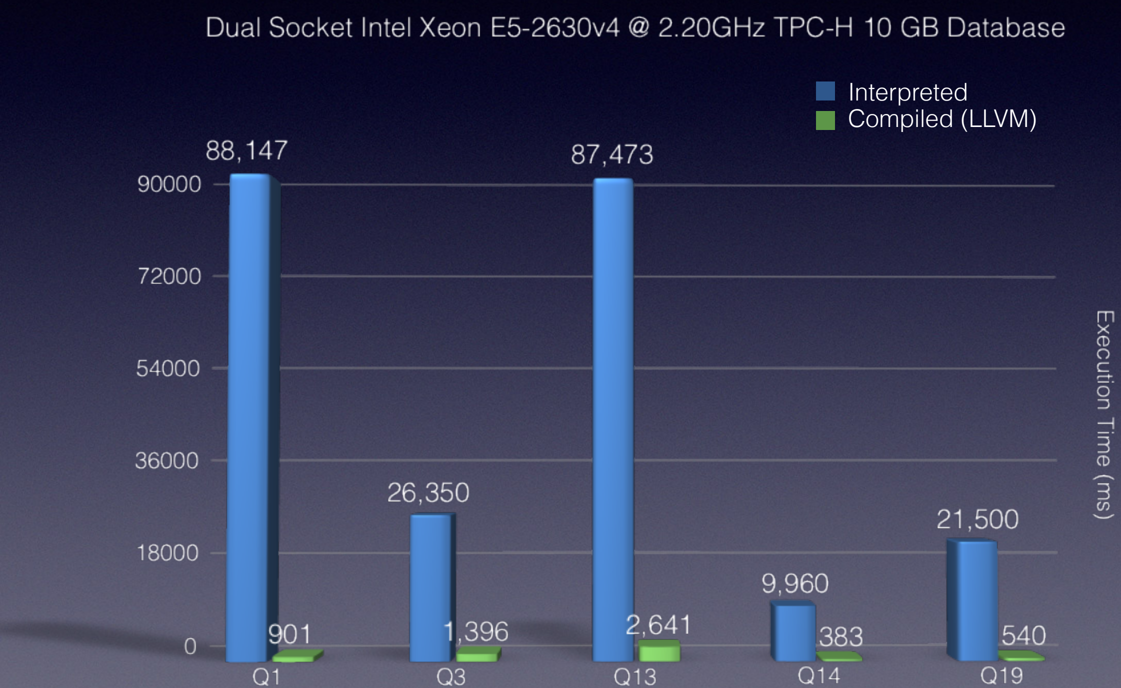
Interpreted vs Compiled (LLVM)

Interpreted vs Compiled (LLVM)



source: Prashanth Menon

Interpreted vs Compiled (LLVM)



Peloton currently only uses a single thread (worker) for each query.

Project: add support for intra-query parallelism that can work with the new LLVM execution engine to further improve performance.

source: Prashanth Menon

We wrote code in LLVM that ***generates*** IR code that runs in a multi-threaded manner.

Goals

✔ 75%: Implement multi-threaded LLVM versions of Sequential Scan with Filters

✔ 100%: Implement multi-threaded LLVM versions of Hash-join and Aggregations ⁉

🤷‍♂️ 125%: Make them NUMA-aware to further improve performance

Benchmark

Benchmark

- We didn’t use TPC-H because those queries contains aggregations that we currently don’t support.

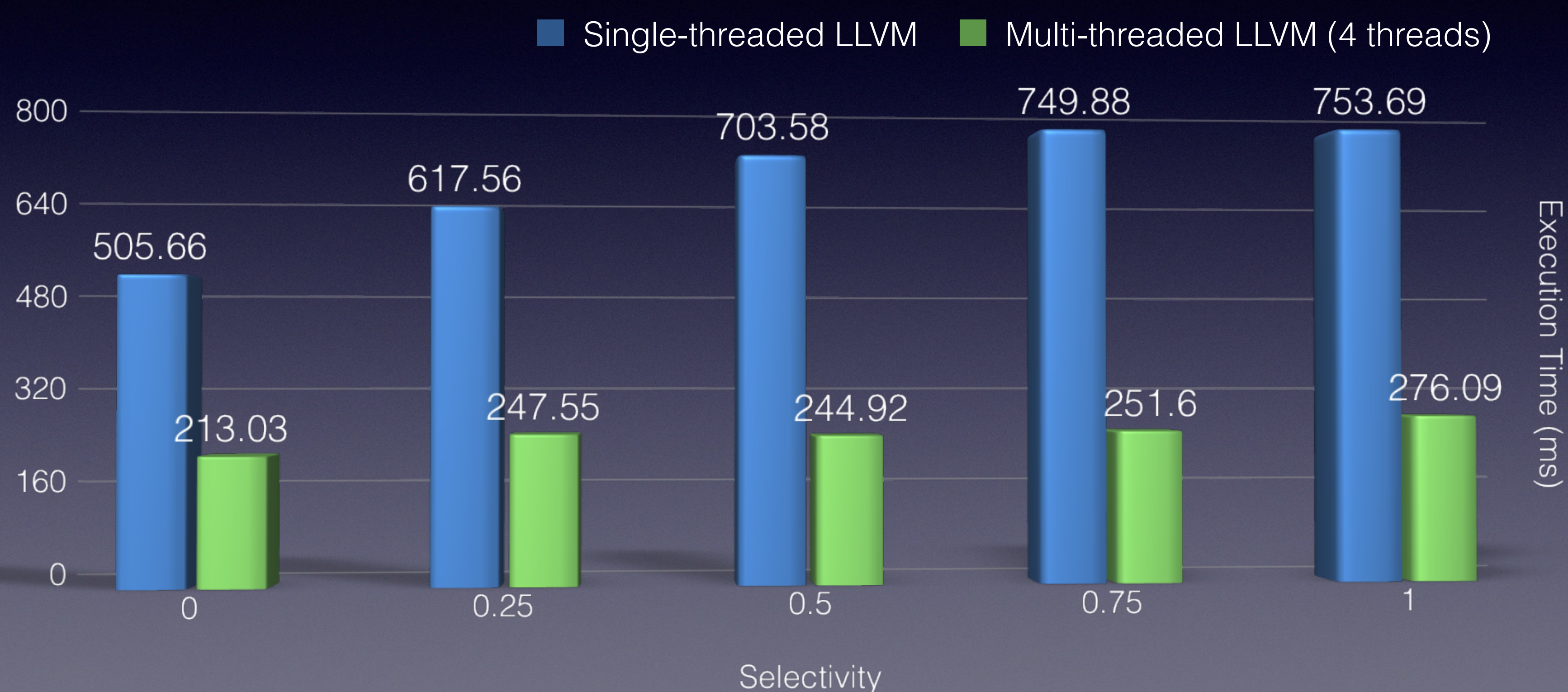
- Sequential Scan:
    - synthetic table with 1M tuples
    - predicates: a >= ? and b >= a

- Hash-join:
    - synthetic left table with 250K tuples
    - synthetic right table with 1M tuples
    - predicates: left_table.col == right_table.col

- Machine:
    - Dual Socket Intel Xeon E5-2620v3 @ 2.40GHz (6 cores / 12 threads)

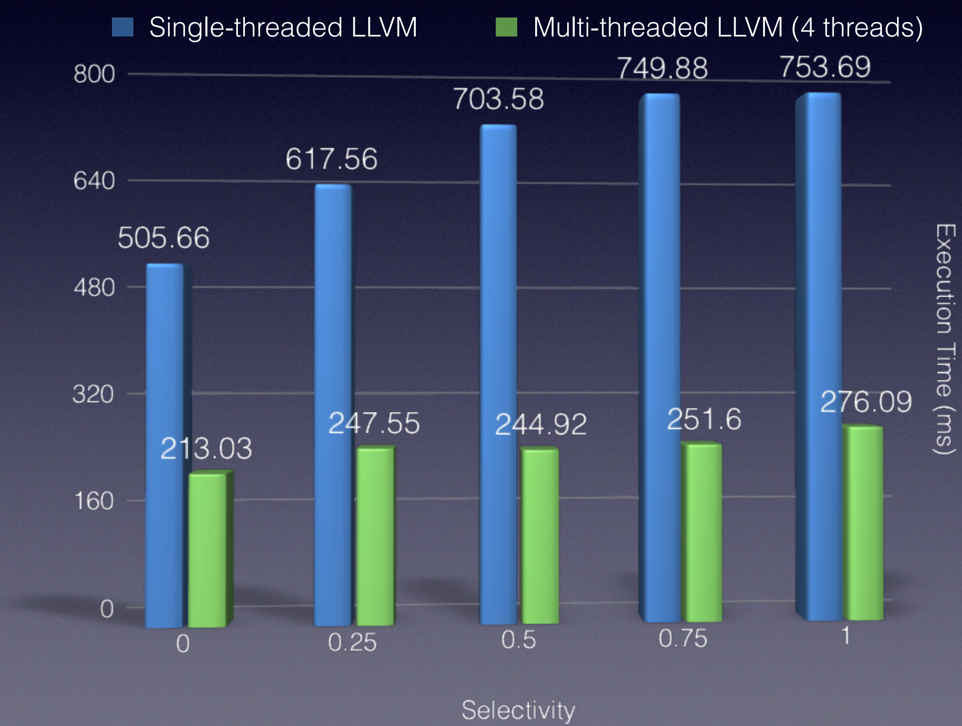
Sequential Scan: Execution Time

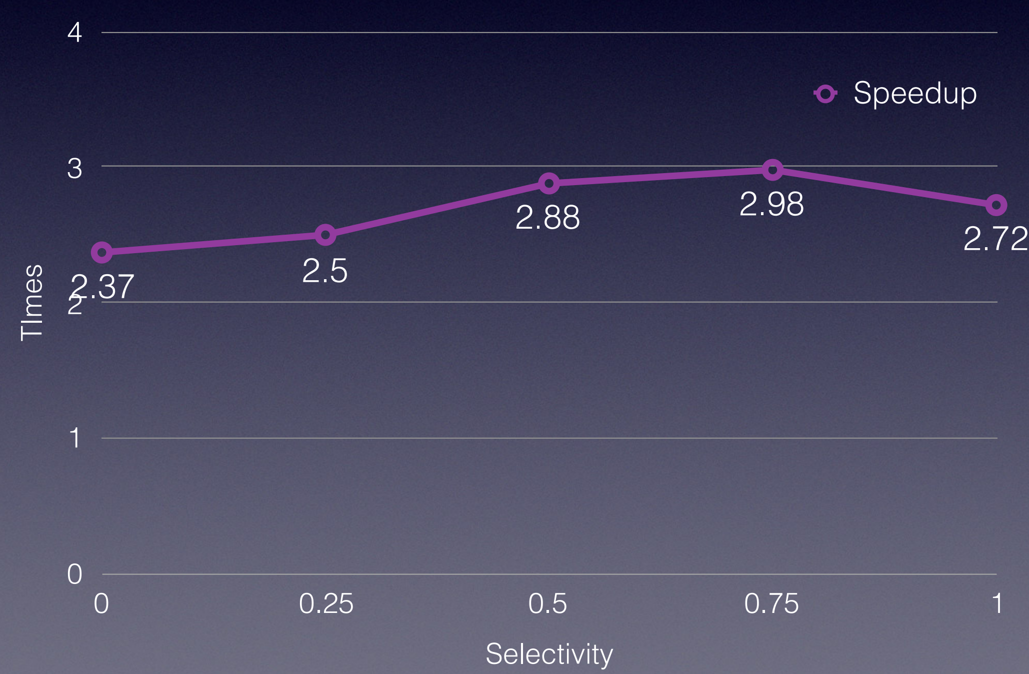
Table Size: 1M tuples
Predicates: a >= ? and b >= a
Dual Socket Intel Xeon E5-2620v3 @ 2.40GHz (6 cores / 12 threads)



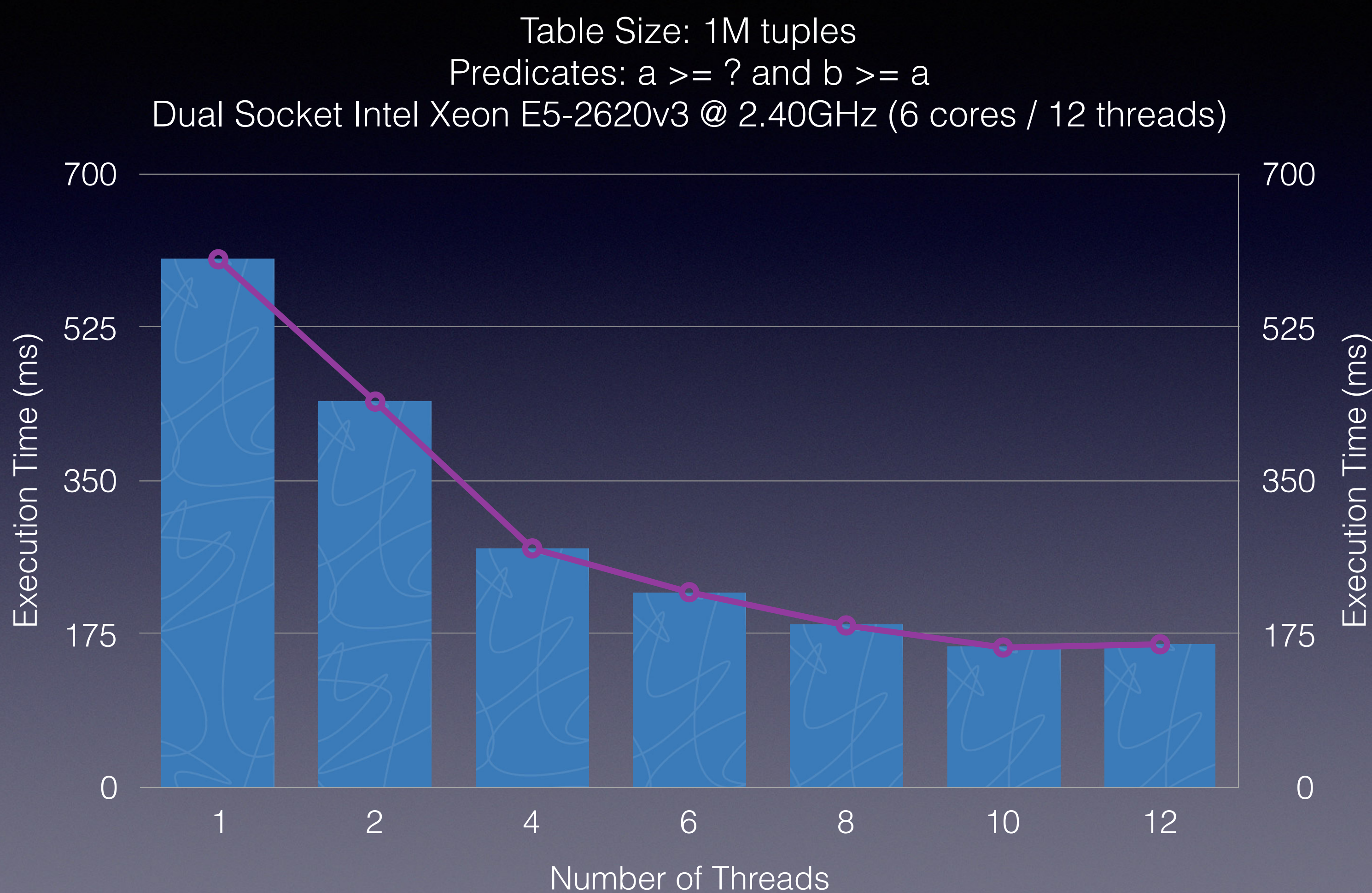
Sequential Scan: Execution Time

Table Size: 1M tuples
Predicates: a >= ? and b >= a
Dual Socket Intel Xeon E5-2620v3 @ 2.40GHz (6 cores / 12 threads)



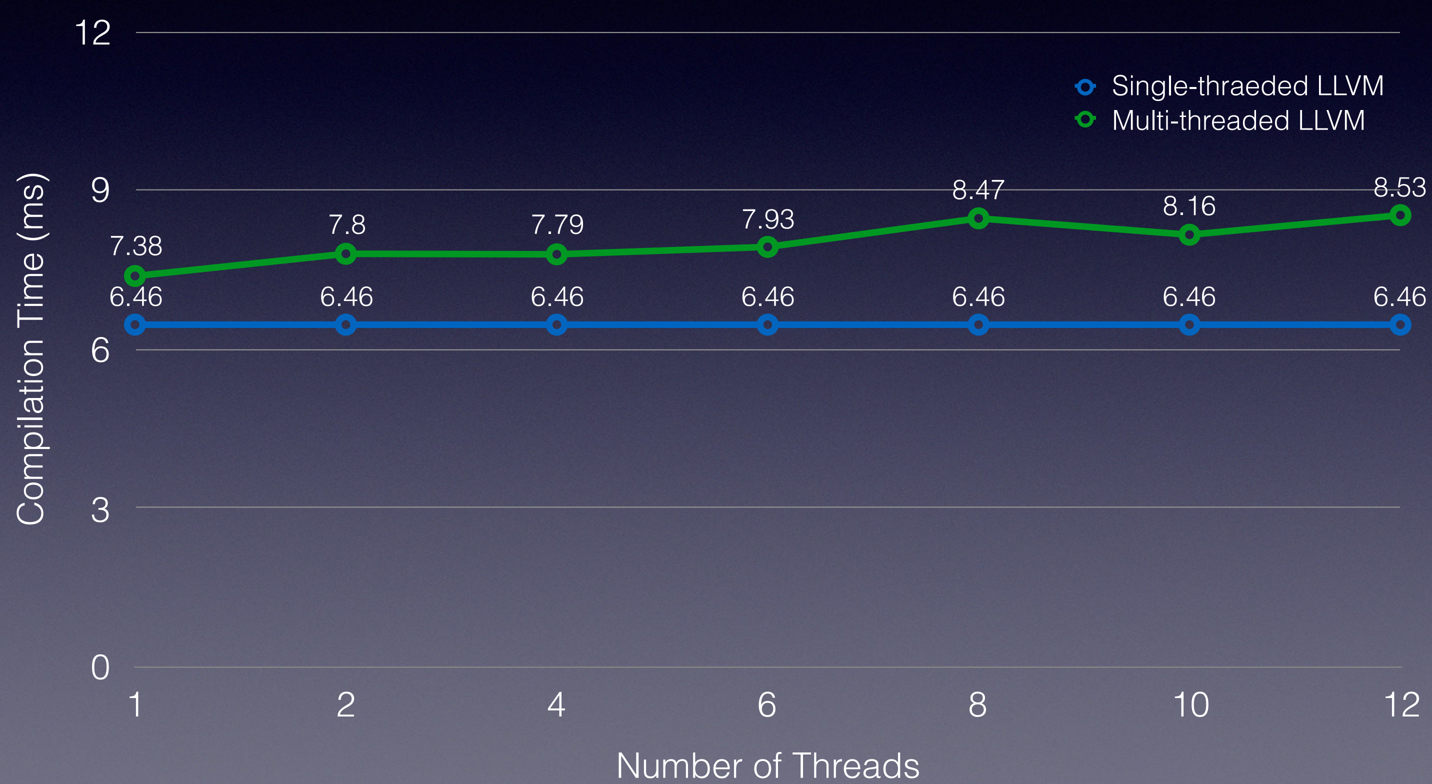


Sequential Scan: Execution Time



Sequential Scan: Compilation Time

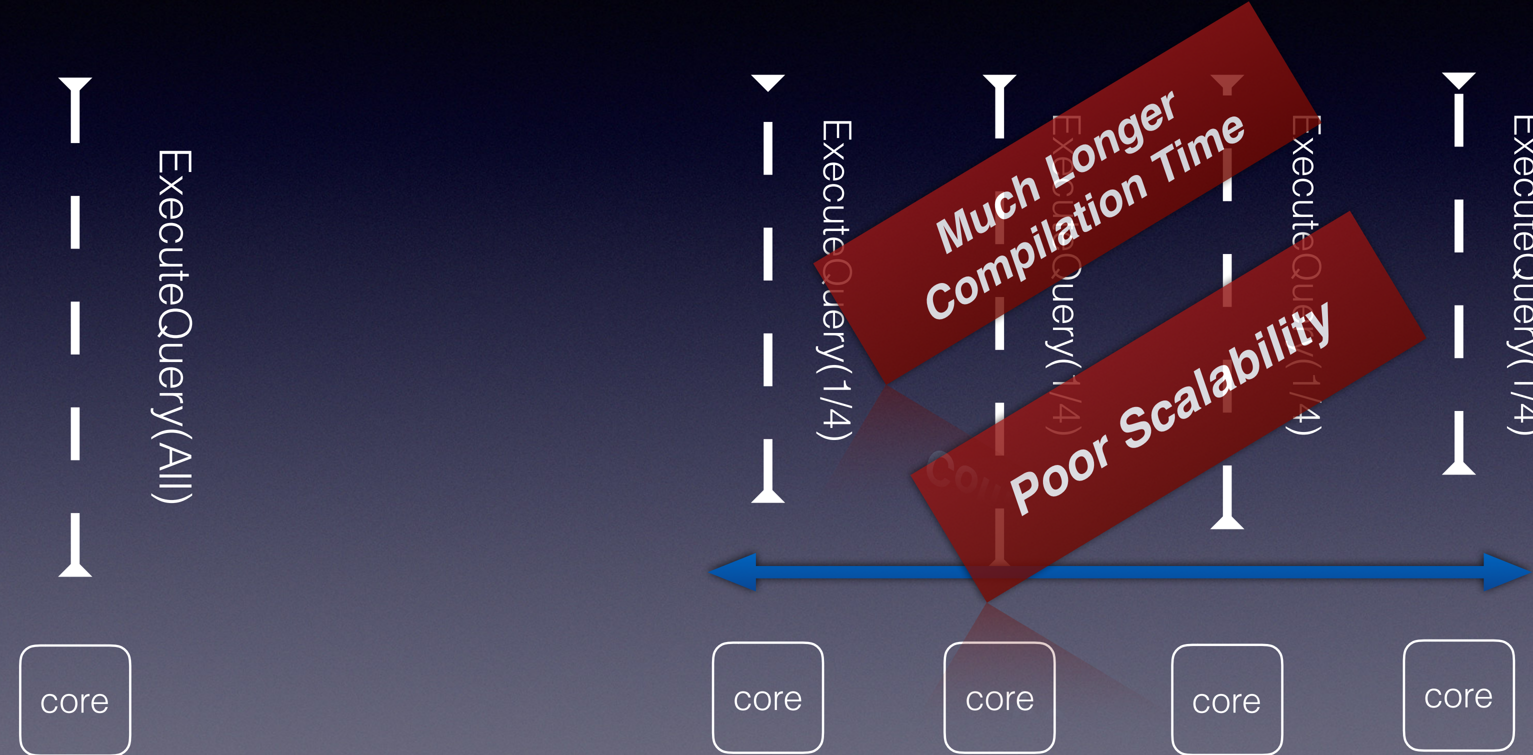
Table Size: 1M tuples
Predicates: a >= ? and b >= a
Dual Socket Intel Xeon E5-2620v3 @ 2.40GHz (6 cores / 12 threads)



Design Decision: Code Generation

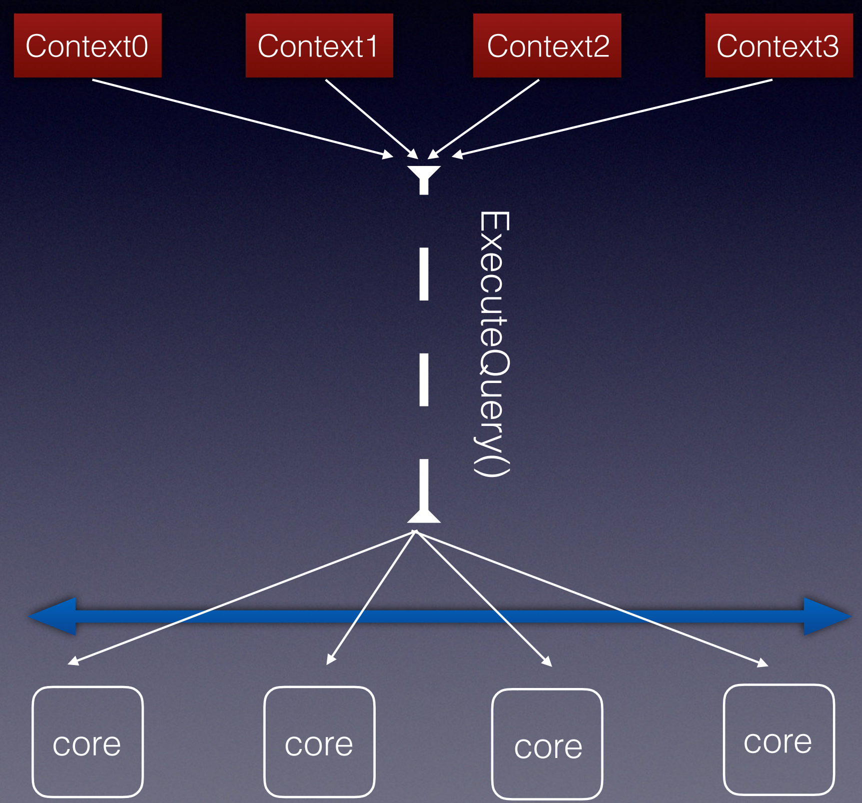
Single-threaded LLVM

Generate Code for Every Thread



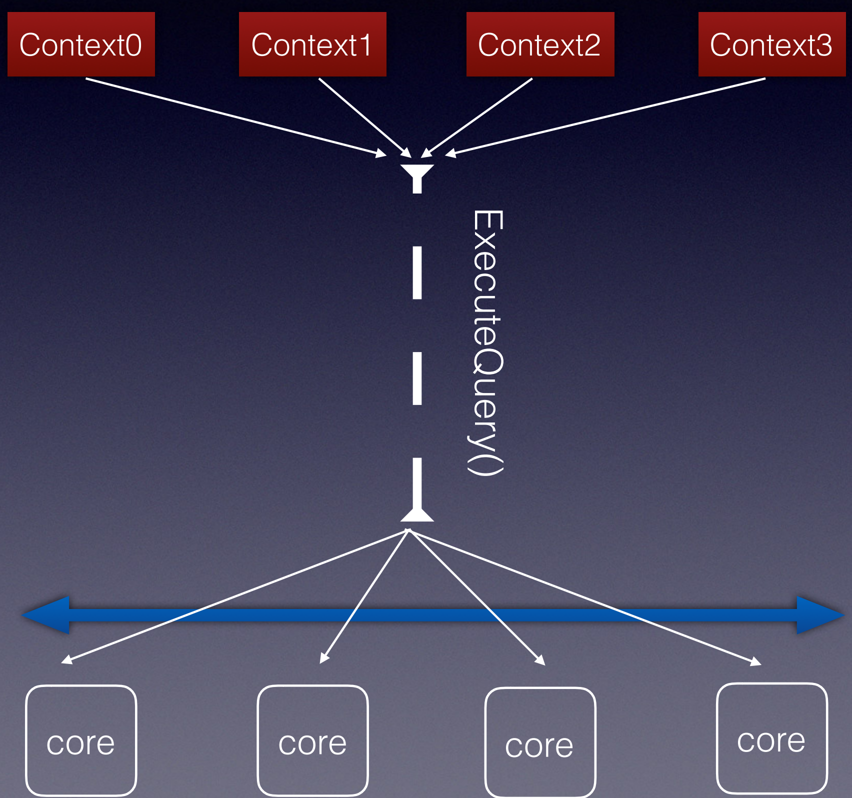
Design Decision: Code Generation

Shared Code Among Threads



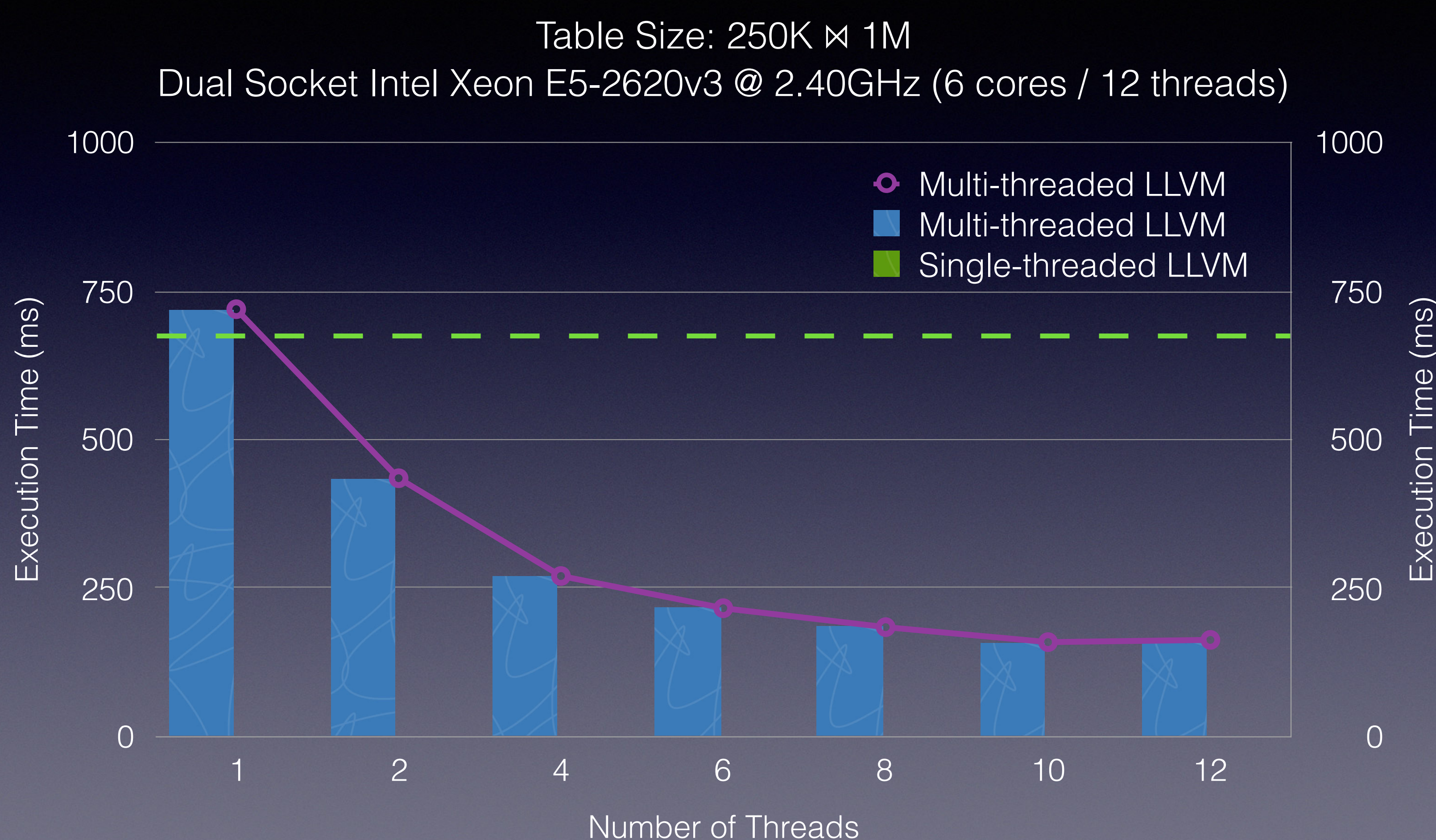
Design Decision: Code Generation

Shared Code Among Threads



- The amount of code it generates is pretty much the same as single-threaded version
- Threads are independent of each other and thus can be easily bound to thread instance in a thread pool
- C++ written context and its proxies can make life easier. And the function calls won't affect performance if they are not on the critical path.

Performance: Hash-Join



Design Decision: Global Hash Table Construction

❌ One thread takes care of merging all local hash tables into a global one and notify all other threads when finish.

- Threads would be aware of others, which could complicate the multi-threading model.
- Constructing local hash tables on their own stacks is more efficient, but stuff on stack cannot be reached by other threads.

✔ Every thread takes care of merging its own local hash table into the shared global hash table (on heap)

- Threads can do the merging without being aware of other threads.
- The hash table in Peloton is written in LLVM. It can efficiently operate on raw data but it's hard (almost impossible) to make it concurrent. In our implementation, the merging is blocking and will be served in a first-come-first-serve order.

Summary

Done: Implemented multi-threaded LLVM versions of Sequential Scan with Filters and Hash-join, and the results are good.

Todo: keep working on Aggregations and refactoring before merging to the master branch

Writing LLVM is non-trivial, and it's even trickier to write LLVM to generate multi-threaded code. ***Thank you for the help, Prashanth!!!!***

Thank You!