Crystal GPU Database

By Devesh
Intro to GPUs
1. Copy input data from CPU memory to GPU memory
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory
Latencies, Bandwidths, and Limits

NOTE: The width of the black lines is proportional to the bandwidth.
It's all about threads

This is how your C code looks like:

This is how the code gets executed on the hardware in heterogeneous computing. GPU calls are asynchronous...
Executing the blocks
Memory Hierarchy

Figure 1: GPU Memory Hierarchy
Or as the Mythbusters explained
**GPU as a Co Processor**

Data resides in CPU memory and is moved to the GPU during query execution.
SSB Benchmark

13 different queries like this

```
SELECT SUM(lo_extendedprice * lo_discount) AS revenue
FROM lineorder
WHERE lo_quantity < 25
AND lo_orderdate >= 19930101 AND lo_orderdate <= 19940101
AND lo_discount >= 1 AND lo_discount <= 3;
```

Using SF = 20
Comparison against other OLAP databases

On average GPU co processing is 1.4x times slower than Hyper
Crystal
Solution

Store the working set in the GPU memory itself as done by

Crystal builds on these using a tile based execution model
Problem with current GPU approach

Motivating query

Q₀: SELECT y FROM R WHERE y > v;
How current systems do it

“Each thread strides through the dataset and determines number of matching rows”

If 100 total threads then
Thread 0 checks rows 0, 100, 200, ...
Thread 1 checks rows 1, 101, 201, ...

Example count arr: [1, 3, 4, 2, 5]

Prefix sum: [0, 1, 4, 8, 10, 15]

Thread 0 will write result in indices [0, 1)
Thread 1 will write result in indices [1, 4)
Thread 2 will write result in [4, 8)
...

Problem: Requires 3 kernels and 2 iterations over the dataset
Introducing Tile Based Execution Model

Tiles rather than threads are basic units of execution.

Load all of tile’s data into shared memory once and reuse it.
Going back to the example

SELECT Y FROM R WHERE Y > 5

(b) With Tile-based processing
## Converting this to code

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlockLoad</td>
<td>Copies a tile of items from global memory to shared memory. Uses vector instructions to load full tiles.</td>
</tr>
<tr>
<td>BlockLoadSel1</td>
<td>Selectively load a tile of items from global memory to shared memory based on a bitmap.</td>
</tr>
<tr>
<td>BlockStore</td>
<td>Copies a tile of items in shared memory to device memory.</td>
</tr>
<tr>
<td>BlockPred</td>
<td>Applies a predicate to a tile of items and stores the result in a bitmap array.</td>
</tr>
<tr>
<td>BlockScan</td>
<td>Co-operatively computes prefix sum across the block. Also returns sum of all entries.</td>
</tr>
<tr>
<td>BlockShuffle</td>
<td>Uses the thread offsets along with a bitmap to locally rearrange a tile to create a contiguous array of matched entries.</td>
</tr>
<tr>
<td>BlockLookup</td>
<td>Returns matching entries from a hash table for a tile of keys.</td>
</tr>
<tr>
<td>BlockAggregate</td>
<td>Uses hierarchical reduction to compute local aggregate for a tile of items.</td>
</tr>
</tbody>
</table>

Crystal provides primitives for each of these steps.
Tuning some key parameters

- **Thread Block Size**: Number of threads per thread block
- **Items per thread**: Number of rows each thread will process

- R has $2^{29}$ rows
- Selectivity factor: 0.5
Projection Queries

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

Note: Q2 is performing a sigmoid operation
Will Crystal against

- CPU based multi threaded implementation
- CPU-Opt: CPU + Non temporal writes (write out cache line to main memory) + SIMD optimizations
- Modeling: Calculating expected runtimes given hardware specifications
How does modeling work

For query

```
SELECT ax1 + bx2 FROM R;
```

\[
\text{runtime} = \frac{2 \times 4 \times N}{Br} + \frac{4 \times N}{Bw}
\]

- \(N = \# \text{ of rows}\)
- \(Br = \text{Read bandwidth}\)
- \(Bw = \text{Write bandwidth}\)

Time to load both columns

Time to write result back
Projection Performance

Table with $2^{29}$ rows
Selection

Q3: SELECT y FROM R WHERE y < v;

Two possible approaches:

for each y in R:
    if y > v:
        output[i++] = v

(a) With branching

GPU IF

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

(b) With predication

GPU Pred
Selection Performance

![Graph showing selection performance comparison between CPU and GPU.]
Hash Join

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

Important points:

- Probe table: 256 million rows, 50% fill rate, linear probing
- Crystal approach:
  1. Load dating using BlockLoad
  2. Each thread finds matching entries and maintains a local sum
  3. Get overall sum using BlockAggergate
Hash Join Performance

![Graph showing the performance of hash join with different build hash table sizes](image)

**Build hash table size**
SSB Comparison

Takeaway: Crystal is around 25x than SOTA OLAP
The $$$ effect

<table>
<thead>
<tr>
<th></th>
<th>Purchase Cost</th>
<th>Renting Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>$2-5K</td>
<td>$0.504 per hour</td>
</tr>
<tr>
<td>GPU</td>
<td>$CPU + 8.5K</td>
<td>$3.06 per hour</td>
</tr>
</tbody>
</table>

GPUS are 6x more expensive but 25x faster
Limitations

Limited amount of GPU memory:

- Use multiple GPUs
- Bit compression to store more data

Only supports numeric formats

- Strings, Dates, etc..
Thoughts?