Analytics using GPU vs CPU: How different are they really?

ABSTRACT

There has been a significant excitement amount and recent work on gpu-based database systems. Previous work have shown that these systems can perform an order of magnitude better than cpu-based database system on analytical workloads such as those found in data warehouses, decision support and business intelligence applications. The elevator pitch behind this performance difference is straightforward: gpu's have higher compute and memory bandwidth than the cpu.

A hardware guy would view this as hype. Given the general notion that database operators are bandwidth-bound, one would expect the maximum gain would be the ratio of the memory bandwidth of GPU to that of GPU. Also, given the restrictive GPU programming model which requires running program over a large number of threads, one would expect additional materializations leading to further degradation in gain. In this paper, we demonstrate that both these assumptions are false. We first show that join and sort achieve speedup larger than the bandwidth ratio. Second, using known GPU optimization techniques, it is possible to avoid the 'paradigm' tax. Finally, it is known that GPUs are more expensive that CPU. We do a performance/cost analysis to show that GPUs can be around 3x more cost-efficient when the critical section of the computation fits in GPU memory.

1 INTRODUCTION

There is a lot of excitement around using accelerators like GPU and FPGAs to accelerate analytics. However, GPUs also have a lot of potential to accelerate memory-bound applications like the ones we considered in NVL. There are two main contributing factors for why GPU's are appealing now:

- The latest generation of GPU's have large amount of memory (the latest K80 Tesla card has 24GB memory, 3 years ago most papers had graphics card with at max 4 GB memory) and significantly higher memory bandwidth than a CPU (CPU bandwidth 60 GBps compared to 480 GBps in K80). On a local machine with Titan X GPU, we observed a memory bandwidth of 280 GBps on GPU (listed as max 330 GBps in device spec) compared to 47GBps on CPU. The PCIe transfer speed is much slower. As a result, all the works so far, which had to ship data from CPU to GPU, suffered from limited gains as PCIe transfer time becomes the bottleneck. The large memory allows us to keep/cache all or a good fraction of the dataset on the GPU itself, eliminating the PCIe transfer overhead.
- GPUs are becoming commodities. Azure recently launched the N-series which are machines with GPUs. The largest of them comes with 2 K80 cards having an aggregate of 48GB GPU memory. The price point is \$2.48*ph* for 24 cores/224GB RAM in addition to 2 GPUs. This is comparable to \$1.83 for 20 core/140GB RAM machine with no GPU. More cards can be stacked, MapD has machines with 8 K80 cards attached on the same machine.

Existing work does apples-to-oranges comparison 1) fails to compare against optimal implementation of both (applies to optimization to only one side) or compares against systems known to be slow 2) fail to exploit the large memory of modern GPU to cache data.

2 ALGORITHMS

There has been considerable research on optimizing standard relational operators for main-memory databases. The same is not true for GPUs. There is still considerable room on coming up with better algorithms and deciding what is the algorithm to use for each operation. One operator I looked at so far is Top-k. On CPU, a sequential version can be done using a priority queue to have $O(nlogk)$. On the GPU, everyone just does a sort followed by selecting the top k entries. I have a sketch of an algorithm that has a $O(n(logk)^2)$ work complexity and $logk$ delay. There is probably some tricks to implement it efficiently and do come with similar algorithms for other operations.

3 HETEROGENOUS COMPUTING

Going back to the Azure VM, the machine has 20 cores, 224GB RAM and 2 K80 with 48GB RAM in total. Complete utilization of the machine requires using both CPU and GPU resources. There are a number of ways to do this. There are a number of ways to look at the problem too.

Scope

- Look at relational operations
- Look at general operators a.la tensorflow

Where is the data

- Data is split between GPU and CPU
- There are two copies of data: one on CPU, one on GPU
- Data is only on the CPU, moved to GPU

How to partition

- Horizontally partition the problem, process fraction of the data on GPU / fraction on CPU
- Look at inter-operator parallelism
- Look at query level parallelism: Query gets scheduled on GPU or CPU

The specific case I am looking at is relational operators with data being on the CPU. We will attempt to run the incoming query on the GPU and move/cache the columns used on the GPU. They are not deleted. When a subsequent query wants to run on the GPU, it will move any additional columns to the GPU in order to execute. Since many queries might access the same columns, we save on the transfer cost by caching it. When we exhaust the GPU memory, we can replace an existing column with the needed column. This is similar to caches in CPU land, but the difference is that there is one full copy of the data on the CPU. So, if the query requires many new columns which we predict maybe by forcasting will not be used in future, we can execute the query on CPU and avoid moving out columns from the GPU.

There is no good open-source GPU-based database. The database resulting from this effort could be an useful resource. NVL could be a good substrate for compiling the query plans.

4 COST MODELLING

It turns out Tensorflow does not have any cost model. Atleast there is no such thing in the public release. User has to manually place computation on different devices. There has been some work on cost modelling of relational operators for GPU operators [2]. This was relatively easy due to the small number of operators. For arbitrary operators, it can possibly be done but would require much more work.

4.1 Selection

Symbols:

label=,noitemsep B_r - read bandwidth of global memory
label=,noitemsep B_w - write bandwidth of global memory
label=,noitemsep C_r - read segment size of global memory
label=,noitemsep C_w - write segment size of global memory
label=,noitemsep W - number of threads in the thread group
label=,noitemsep R - cardinality of table R
label=,noitemsep n - number of projected columns
label=,noitemsep K_i - attribute size of the ith projected column
label=,noitemsep m - number of predicate columns
label=,noitemsep P_i - the attribute size of the ith predicate columns
label=,noitemsep r - selectivity of the predicates

Old Model: Scan and write out filter per column with predicate

$$T_1 = \sum_{i=1}^{m}\left(\frac{P_i W}{C_r} + \frac{4W}{C_r}\right) \times \frac{R}{W} \times \frac{C_r}{B_r} + \sum_{i=1}^{m} \frac{4W}{C_w} \times \frac{R}{W} \times \frac{C_w}{B_w}$$

Read the filter and write results to global memory.

$$T_2 = \sum_{i=1}^{n}\left(\frac{4W}{C_r} + \frac{K_i}{4}\right) \times \frac{R}{W} \times \frac{C_r}{B_r} + R \times r \times \sum_{i=1}^{n} \frac{K_i}{4} \times \frac{C_w}{B_w}$$

New Model: Scan all the columns and write out the filter

$$T_1 = \sum_{i=1}^{m} \frac{P_i W}{C_r} \times \frac{R}{W} \times \frac{C_r}{B_r} + \frac{4W}{C_w} \times \frac{R}{W} \times \frac{C_w}{B_w}$$

Read the filter and write results to global memory.

$$T_2 = \left(\frac{4W}{C_r} + \sum_{i=1}^{n} \frac{K_i}{4}\right) \times \frac{R}{W} \times \frac{C_r}{B_r} + \sum_{i=1}^{n} \frac{rRK_i}{C_w} \times \frac{C_w}{B_w}$$

CPU Model:

REFERENCES

[1] Kaibo Wang, Kai Zhang, Yuan Yuan, Siyuan Ma, Rubao Lee, Xiaoning Ding, and Xiaodong Zhang. 2014. Concurrent analytical query processing with GPUs. *Proceedings of the VLDB Endowment* 7, 11 (2014), 1011–1022.

[2] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of processing data warehousing queries on GPU devices. *Proceedings of the VLDB Endowment* 6, 10 (2013), 817–828.