Using gem5 simulator to accelerate database operations

Anurag Chakraborty
CS 850 - Final Project Report
a8chakra@uwaterloo.ca

ABSTRACT

The aim of this project is to identify acceleration opportunities for database operations. Traditionally, databases have been designed to be efficient in memory management. With the advent of big data, there has been significant increase of computation pressure which has led to the incorporation of techniques such as vectorization and SIMD parallelism to accelerate database operations.

For this project, two main goals have been achieved: (i) micro-architectural analysis of query execution by running the TPC-H benchmark on the DuckDB database, (ii) identifying the operations which are more "core bound", consume a lot of Instructions-per-cycle (IPC) and simulating them on a hardware accelerator based on the gem5 simulator. Using the GCN3 model in gem5 that uses AMD Radeon Open Compute platform (ROCm) to simulate GPUs at the ISA level, two common operations: hash computation and decompression are simulated.

INTRODUCTION

In recent years due to the exponential growth of data, data management systems have had to deal with significantly higher load on their computational resources. To cope with this load, various techniques to increase instruction parallelism and cache locality such as vectorization and SIMD instructions have been incorporated in databases for faster query execution. Hardware acceleration using GPU and FPGA is another avenue that has shown promise for query acceleration.

However, not all operations are suitable to be offloaded to hardware accelerators. For example, if an operation is more memory bound, i.e the CPU cycles are spent behind waiting to fetch data from the DRAM or due to a cache miss at the L1/L2/L3 level - these operations may not get overall improved execution time even if they were to be run on an accelerator. A thorough microarchitecture level analysis to identify hotspots for complex queries on a database and then identifying the real core bound operations that are bottlenecked by non-availability of hardware resources is essential before deciding which operations should be accelerated.

There has been prior work [1, 2] that have focused on running well known benchmarks on a database to identify the sections of a query execution that consume the most time. These papers focus more on the query plans generated by the database and how to optimize query execution based on better plan generation. Some recent papers such as Sen at al. [3] have focused on microarchitectural analysis by running the LDBC graph benchmark and identifying the percentage of cycles spent behind memory bound and core bound operations. Their analysis reveals a significantly more no. of pipeline slots and cycles are wasted behind memory bound operations compared to core bound.

BACKGROUND

The database used for analysis in this project is DuckDB [4], a popular open source in-process SQL OLAP (Online Analytical Processing) database management system. DuckDB implements a columnar vectorized query execution engine, where processing for an operation is always done on a vector of values. This approach takes advantage of cache locality and instruction parallelism in modern processors to greatly improve query performance compared to traditional relational systems such as PostgreSQL, MySQL or SQLite which process each row sequentially. Vectorized query execution leads to far better performance in OLAP queries. OLAP workloads are usually load intensive queries that handle complex long running queries and process on almost the entire table. OLAP queries are characterized by heavy read and low write workloads.

TPC-H [5] is an industry standard decision support benchmark. It consists of a suite of business-oriented ad hoc queries and concurrent data modifications. This benchmark illustrates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions. The gem5 simulator [6] is a modular platform for computer system architecture research, encompassing system-level architecture as well as processor microarchitecture. It is an event driven simulator which supports different CPU execution models, timing models, cache hierarchy and also full system emulation. gem5 has two execution modes: *SE mode* or the syscall emulation mode emulates the OS syscall, no OS is actually running in the background; *FS mode* or full system emulation mode where gem5 emulates a bare metal system (full virtualization). The GCN3 (Graphics Core Next Generation 3) GPU model used in this project [7] simulates a GPU based on the Radeon Open Compute Platform (ROCm) from AMD. Currently only SE-mode is supported for simulation and all GPU kernel level driver functionality is modeled within the SE mode layer of gem5. Similar to CUDA kernels that run on NVIDIA GPUs, AMD supports Heterogeneous-Compute Interface (HIP) which is a interface in C++ for writing kernels.

Intel's Top down Microarchitecture analysis (TMA) [8] method is a technique that takes advantage of on-chip Performance Monitoring Units (PMUs) to count specific hardware

events such as cache misses or branch misdirections. TMA helps to identify on average how well the CPU's pipelines were being utilized when running a program. Intel VTune profiler [9] and toplev (pmu-tools) [10] are two well known open source tools that use this approach to keep track of performance metrics and have been used in this project to identify bottlenecks for query benchmark runs.

MOTIVATION

Certain database operations such as filtering, computing hash, sorting, decompression are very common in database workloads and present acceleration opportunities. The reason for this being that while for smaller datasets they do not consume much CPU cycles, when the datasets become larger the operations start to add computational pressure on the system. But since there are no workarounds at the application level to accelerate these operations (even though these are pretty straight forward to implement), they significantly increase execution time of queries.

To identify such bottlenecks first, profiling tools like Linux perf [11], toplev and VTune are a good start to identify which database operations consume the most CPU cycles. Specifically, operations which are backend bound and core bound and consume a lot of IPCs are a good starting point. Studying these operations with different GPU configurations can give us a good idea of how much benefits we can get if they were to be offloaded to a similar hardware accelerator.

QUERY EXECUTION OVERVIEW

Table 1 displays execution runtime of the 22 TPC-H queries on DuckDB. The queries were run with scale factor 10 (SF10). A total of 8 tables are present with the largest table being *lineitem* (60 million rows). For the system setup, a dual-socket Intel Xeon CPU E5-2670 @2.60GHz (Sandy Bridge) server with a total of 16 physical (32 logical) cores and 256GB RAM, running Ubuntu 20.04.4 LTS was used. All queries were run on DuckDB utilizing 32 threads and 32 GB memory limit.

From the table, we identify Q18, Q21, Q17, Q9, Q4 as the top 5 queries that take the most time to run. For the scope of this project, the TMA microarchitectural analysis will be done for these queries. In general as expected the most complex queries involve a high number of *table joins on large tables* or *aggregations based on multiple columns*. For example Q18 which runs the longest has 3 joins and 2 group by aggregations, Q21 has 6 table joins, Q17 has 1 group by aggregate and 2 joins, Q9 has 5 joins and 1 group by and Q4 has 1 hash join and 1 aggregate.

MICROARCHITECTURE ANALYSIS & FINDINGS

Table 2 shows the functions for which the most number of Instructions were retired. Linux perf profiling tool (perf record, event instructions) was used for this analysis. The *PatasScan* function is a decompression algorithm that decompresses compressed columns of type double (64 bit size). The compression algorithm that DuckDB uses is Patas, which is a variant of the Chimp compression algorithm. Hence decompression operation is a good candidate to select for acceleration considering 7.75% of all instructions retired were composed of this. Another simple operation that consists of a high % of instructions retired is Hash computation ($Hash < long >$ function). DuckDB uses the murmur hash function for computing hash for 64 bit int type. Two other functions stand out - *FindOrCreateGroup* and *TemplatedMatchOp*. The first function is used during aggregation when group by columns have to be scanned to identify and create unique groups. The second function is used for both join operations and aggregation to compare hash values and check if two groups are equal.

TPC-H Query	Runtime (in millis)	Result (Row x Col)
1	390	4 x 10
2	156	4801 x 8
3	269.2	113067 x 4
4	1003	5 x 2
5	276	5 x 2
6	127	1 x 1
7	805.5	4 x 4
8	262.8	2 x 2
9	1128	175 x 3
10	610	371104 x 8
11	78.33	9322 x 2
12	129.67	2 x 3
13	616	45 x 2
14	211	1 x 1
15	-	-
16	215	27839 x 4
17	1064.2	1 x 1
18	1534	111 x 6
19	353.6	1 x 1
20	633.2	1827 x 2
21	1409.4	3981 x 2
22	315.4	7 x 3

Table 1. Query Execution time for TPC-H benchmark queries

Instructions Retired (%)	Function
7.75	PatasScan<double> ::LoadGroup
6.89	GroupedAggregateHashTable ::FindOrCreateGroupsInternal
6.00	Hash<long>
2.83	TemplatedMatchOp<Equals, false>
2.5	cfree@glibc

Table 2. Most % of Instructions retired (across Q18, 21, 17, 9, 4)

Table 3 shows the functions for which the most number of cycles were spent. This was also measured using the Linux perf tool (cycles event). The most no. of cycles corresponds to the most amount of time spent during query runtime. The *InsertHashes* function is a costly function invoked during the hash join operation for a query where local threads have to merge their results to the global hash table. Even though the function is parallelized, it consists of atomic load and atomic compare and exchange operations leading to expensive usage of CPU cycles. This makes this function inherently serial due to the atomic operations on a global data structure and a difficult candidate for offloading to a hardware accelerator.

The *TemplatedMatchOp* function also consumes a lot of cycles but the cycles are actually spent waiting for memory bound operations (such as *mov* instructions that read the contents of a particular address into a register). Similarly for the *FindOrCreateGroupsInternal* function, most of the cycles are spent in a code section that does pointer chasing leading to frequent cache misses and fetching data from the RAM (characterized by an *lea* instruction that computes some memory address and retrieves its contents into a register using the *mov* instruction).

Cycles spent (%)	Function
8.9	JoinHashTable::InsertHashes
8.7	TemplatedMatchOp<Equals, false>
6.02	GroupedAggregateHashTable ::FindOrCreateGroupsInternal
4.5	PatasScan<double> ::LoadGroup
2	Hash<long>

Table 3. Cycles % (across Q18, 21, 17, 9, 4)

Such operations are not suitable to be offloaded to an accelerator since they are memory bound and may benefit more from efficient memory management. Hence, the *PatasScan* decompression function (which also consumes 4.5% cycles overall) and Hash function which combined account for 6.5% cycles spent and 14% of instructions retired are better acceleration candidates compared to the other functions.

Figure 1 shows the breakdown of processor pipeline slots usage using the top-down analysis technique. The most % of pipeline slots underutilized are due to backend bound operations which comprise of memory bound and core bound. Memory bound under utilization is still the highest out of all causes, the % is almost double that of core bound pipeline slots. This indicates that memory stalls due to L1/L2/L3 cache misses or due to latency / bandwidth saturation from DRAM data access is causing the majority of performance degradation.
While core bound pipeline slots are much lower, they do present an opportunity for improvement since it is a steady bottleneck of about 20% for all the queries the analysis was done for.

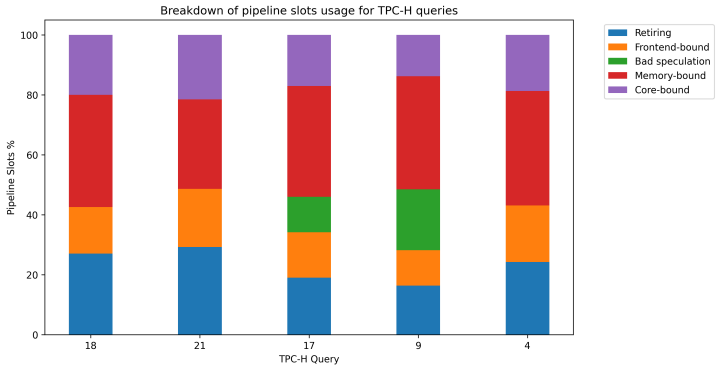

Figure 1. Top down Microarchitecture Analysis (TMA) for queries

GEM5 SIMULATION & IMPLEMENTATION

The gem5 simulation has been done for the following functions: **(i)** murmur64 hash function and **(ii)** Patas decompression of 64 bit values (double precision). As per the analysis done in the previous section, both of these pose an overhead among all benchmark queries analysed.
The hipKernel code for **(i)** is as follows:

```
template <typename T>
__global__ void
hashCompute(T *C_d, const T *A_d, size_t N)
{
    size_t offset = (hipBlockIdx_x *
        hipBlockDim_x + hipThreadIdx_x);

    size_t stride = hipBlockDim_x *
        hipGridDim_x ;

    for (size_t i=offset; i<N; i+=stride) {
        C_d[i] = A_d[i];
        C_d[i] ^= C_d[i] >> 32;
        C_d[i] *= 0xd6e8feb86659fd93U;
        C_d[i] ^= C_d[i] >> 32;
        C_d[i] *= 0xd6e8feb86659fd93U;
        C_d[i] ^= C_d[i] >> 32;
    }
}
```

The main logic within the loop is the murmur64 hash function. Each thread in an APU / GPU has a unique thread index that is calculated first (the offset value). The significance of the hip kernel variables are as follows:

hipBlockIdx_x : the block index of the thread block
hipBlockDim_x : the total no. of threads in a block (256)
hipThreadIdx_x : the thread index in a thread block
hipGridDim_x : the total no. of thread blocks (512)

The stride calculated points to the next value which the thread can operate on. It is basically the product of the total threads in a block and the total thread blocks in a grid.

The hipKernal code for **(ii)** Patas decompression simulation is as follows:

```
typedef struct packed_metadata {
  uint8_t prev_idx;          // 7 bits
  uint8_t significant_bytes; // 3 bits
  uint8_t trailing_zeros;    // 6 bits
} hipLaunchKernelStruct_1;

__global__ void decompress(
hipLaunchKernelStruct_1 *packedMetadata,
uint8_t *byte_input, uint64_t *byte_output,
size_t total_column_segments,
size_t column_segment_size) {

size_t offset = (hipBlockIdx_x * hipBlockDim_x
          + hipThreadIdx_x);

size_t final_idx = min(total_column_segments
          * column_segment_size,
          (offset + 1)
          * column_segment_size);

uint64_t prevVal = 0u;
uint64_t index = 0u;
  for (auto i = offset * column_segment_size; i
       < final_idx; i++) {
```

```
    auto metadata = packedMetadata[i];
    uint64_t temp = 0;
    switch (metadata.significant_bytes) {
    case 1:
      memcpy(&temp, byte_input + index, 1);
      index += 1;
      break;
    /*
    * for other byte sizes, same operation of
        memcpy() will occur
    */
    default:
      if (metadata.trailing_zeros < 8) {
        memcpy(&temp, byte_input + index, 8);
        index += 8;
        break;
      }
      temp = 0;
    }
    temp = (temp << metadata.trailing_zeros) ^
        prevVal;
    prevVal = temp;
    byte_output[i] = temp;
  }
}
```

The decompression algorithm used to compress 64 bit double type values in DuckDB is inherently serial, because every value is XORed with the previous value and then the significant bytes of the result are stored. The number of trailing zeros is also saved as extra metadata. Since this algorithm would be difficult to parallelize, every column in the table is divided into column segments and then each column segment is picked up by a thread to decompress to introduce task parallelism.

For the kernel code here also, a similar approach has been taken where each thread gets mapped to a particular column segment. Similar to the previous task, first we calculate every thread's position out of all the thread blocks. There is another case where the total segments is much larger than the total no. of threads available, but since the simulation time for this task was much longer (noted in the Evaluation section), the total column segments for the input was set as equal to the total threads available (product of threads per thread block and total thread blocks).

To decompress each value, we simply read the packed metadata (2 bytes aligned) and consider how many bytes the XORed value consists of. It is then memcpy-ed into and shifted right the no. of trailing zeros. Then another XOR is required with the previous (decompressed) value to received the original value. The last step of XORing is what makes this algorithm's approach serial.

EVALUATION

Table 4 shows the runtime comparison between APU/GPU and the CPU implementation for the hash computation task. As input for this task, the total no. of hash computation input given was ~211,520,000 which is around the same for Q18 from the benchmark (this query had the most hashCompute() function calls). For the simulation, all the numbers for which hashCompute() is performed in Q18 are collected and provided as the input for both the CPU execution and gem5 simulation.

gem5 has support for both AMD APU and GPU. For the APU setup, 4 CPU cores, 16 compute units (CU) were used with the gfx801 architecture. The GPU set up is also the same (4 CPU cores, 16 compute units) except it is based on the gfx803 architecture. The total no. of thread blocks was set as 512 and each thread block consists of 256 threads. The full specifications of the memory model and the APU/GPU architecture is as follows:

(i) For **DDR5_8400_4x8** model, the base memory bandwidth is - $(8.4 \times 4 \times 8/8)$ = 33.6 GBps. This is the bandwidth per directory controller per memory controller. Since 4 directory controllers are added, the total bandwidth is ~135 GBps. The memory size is 3 GB, 4 L2 cache each of size 2 MB, 1 L3 cache of size 16 MB is present.

(ii) For **HBM_2000_4H_1x64** model, the base memory bandwidth is - $(2 \times 1 \times 64/8)$ = 16 GBps. This is the bandwidth per directory controller per memory controller. With 4 directory controllers, the total bandwidth is 64 GBps. Rest of the cache configuration is same. Since HBM memory from hardware manufacturers can support upto 512 bytes of cache line size, greater cache line size was also used to run the simulation but they resulted in gem5 simulation errors. Only 64 bytes cache line size was stable enough to run the query benchmarks.

In the Ruby memory system in gem5, each directory controller accounts for a range of memory address i.e it services requests for that particular address range. The default memory bandwidth considers only a single directory controller for the whole address range and thus adding 4 directory controllers effectively increases the bandwidth by 4 times. This configuration was required for boosting the simulation execution time, or else the memory bandwidth becomes a bottleneck.

Table 5 shows the execution time comparison for the Patas decompression task between GPU and CPU. The hardware simulator setup was same for both the tasks. The total decompression calls for Q18 is >9 billion, with around 133,000 total column segments and each segment of size ~363,000. Since simulating this exact input on gem5 would take a long time especially for 2 different memory models and then for the APU and GPU, the exact input could not be simulated.
Instead a micro-benchmark was used for the comparison. 131,072 column segments were given as input with each segment having 1,000 elements. This specific no. of column segments was chosen because - total blocks = 512, threads per block = 256, 512×256 = 131,072. This ensures each thread gets mapped to 1 column segment and we take full advantage of the APU/GPU's parallelism.

DISCUSSION AND FUTURE WORK

From Table 4 and 5 the gem5 simulation with gfx801 and gfx803 are both faster than the CPU implementation. On 1 thread, the CPU implementation is 17x slower than APU/GPU for the hashCompute task and 14x slower for the decompression task. After parallelizing the task on 32 threads, the CPU execution time is much closer to the gem5 simulation runtime. For the 1st task, the APU is still faster than the CPU implementation by ~2ms (both memory models). The GPU implementation is slightly slower because there is an additional cost of transferring the memory from the host memory

Memory Model	APU gfx801 (4 cores, 16 cu)	GPU gfx803 (4 cores, 16 cu)	CPU (Intel Xeon Server E5-2670) DDR3 Synchronous memory @**51.2 GBps** 32 threads parallelized using OpenMP
DDR5_8400_4x8 4 directory controllers Bandwidth = 33.6 x 4 = **134.4 GBps**	274,150 μs	274,831 μs	3,587,297 μs - 1 thread 276,925 μs - 32 threads
HBM_2000_4H_1x64 4 directory controllers Bandwidth = 16 x 4 = **64 GBps**	274,335 μs	276,695 μs	3,587,297 μs - 1 thread 276,925 μs - 32 threads

Table 4. hashCompute (murmur64) - Execution time (in microseconds)

Memory Model	APU gfx801 (4 cores, 16 cu)	GPU gfx803 (4 cores, 16 cu)	CPU (Intel Xeon Server E5-2670) DDR3 Synchronous memory @**51.2 GBps** 32 threads parallelized using OpenMP
DDR5_8400_4x8 4 directory controllers Bandwidth = 33.6 x 4 = **134.4 GBps**	180,198 μs	184,561 μs	2,481,545 μs - 1 thread 190,888 μs - 32 threads
HBM_2000_4H_1x64 4 directory controllers Bandwidth = 16 x 4 = **64 GBps**	181,543 μs	186,695 μs	2,481,545 μs - 1 thread 190,888 μs - 32 threads

Table 5. Patas Decompression - Execution time (in microseconds)

to the GPU memory, which is not present for the APU case. Similarly for the 2nd task, the APU runtime is faster than the parallelized CPU version by ~10ms on both memory models while for the GPU it is ~4-6ms faster.
Thus offloading a common database operation such as hash computation and decompression to a hardware accelerator such as a GPU can provide speedup for overall execution time. One thing to note here is that even though the CPU parallelized 32 threads implementation was much closer to the gem5 simulation runtime, this would be the ideal case as not all threads will be always available for the task. Based on task parallelism, different threads maybe allotted different tasks at different point in time of the query execution. Offloading hashing or decompression would be freeing up the CPU to focus on other tasks and help to reduce overall execution time.

As part of future work, running the hashCompute and decompression code on more modern hardware would be interesting since the gem5 GCN3 model is currently not the latest GPU version available. While the memory bandwidth used for the simulation is almost the same as the ones currently available (128-150 GBps) in the market for APU, GPU requires much more bandwidth. GPU hardware is usually pared with high bandwidth HBM memory or GDDR6 DRAM type memory. And the bandwidth for these can go upto 500 GBps or more. Compared to that, the gem5 simulation for HBM was only upto 64 GBps, much lesser than currently available hardware in the market. HBM also uses much greater cacheline size of 512 bytes, but gem5 simulation ran into errors while running the code for cache line size greater than 64 bytes. Adding support for all these changes (more memory bandwidth, higher cacheline size) and then running the comparison with the CPU execution would be a much better comparison.

Another thing to note here is that the task 2 (decompression) could not be parallelized beyond the maximum size of a column segment, due to the inherent serial nature of the decompression procedure. For such cases using a GPU as the hardware accelerator might not be the right choice. There has been prior work on designing custom hardware ASIC accelerators [12] for tasks such as Deserialization, as part of future work exploring such a solution would also be interesting for a Database.

REFERENCES

[1] Markus Dreseler, Martin Boissier, Tilmann Rabl, Matthias Uflacker. "Quantifying TPC-H Choke Points and Their Optimizations"

[2] Peter Boncz, Thomas Neumann, Orri Erling. "TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark"

[3] Rathijit Sen, Yuanyuan Tian. "Microarchitectural Analysis of Graph BI Queries on RDBMS"

[4] https://duckdb.org/

[5] https://www.tpc.org/tpch/

[6] https://www.gem5.org/

[7] https://www.gem5.org/documentation/general_docs/gpu_models/GCN3

[8] https://www.intel.com/content/www/us/en/docs/vtune-profiler/cookbook/2023-1/top-down-microarchitecture-analysis-method.html

[9] https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide/2023-0/command-line-interface.html

[10] https://github.com/andikleen/pmu-tools

[11] https://perf.wiki.kernel.org/index.php/Main_Page

[12] Sagar Karandikar, Chris Leary, Chris Kennelly, Jerry Zhao, Dinesh Parimi, Borivoje Nikolic, Krste Asanovic, Parthasarathy Ranganathan. "A Hardware Accelerator for Protocol Buffers"