CUDA-DB – Final Write-up

Tom Chittenden (tchitten)

Summary
I created a C++ library that managed tables of structs/classes on an NVIDIA GPU. The table allows you to define certain fields of the struct/class as indices, which allows for efficient lookups. In addition I created a criteria specification protocol that allows you to narrow down the selected results. I performed benchmarks against the SQLite in memory database.

Background
While the database workload has lots of room for parallelization, little effort has been made to exploit the massively parallel capabilities and high memory bandwidths of GPUs. This is likely due to the limited memory available on a GPU. Despite this limitation, I still found it to be an interesting academic experiment to see what kind of speedup could be obtained. By parallelizing the selection of elements from indices I was able to match the speed of SQLite’s highly optimized in memory database and I have no doubt with further optimizations my database could surpass SQLite’s selection speed.

The primary data structure used in the database is a 16-32 b-tree. B-trees are simply a generalization of binary search trees optimized for systems that read/write blocks of data. In databases they are typically used due to the access characteristics of the disks that generally store these indices. The principle behind why they are efficient is equally applicable in a SIMD context that is optimized for 32 wide loads/stores. I chose 16-32 B-trees since they allow for efficient reading by CUDA warps: each lane in a warp can read a key or child of the node. Due to their high branching factor, these indices can be traversed in no more than about 6 steps. The b-tree needed to support essentially two operations, insert and lookup queries.

Approach
My initial approach at b-tree construction was to create a forest of 15 b-trees, one for each of the streaming multiprocessors on the GPU. This allowed for extremely fast insertion/lookup, however it did not maintain any ordering property among separate b-trees; a given record could be located in any of the trees. I thus chose to represent indices by only a single b-tree in order to maintain order. In the future I would like to find a better way to parallelize construction/search of the indices as currently it is performed by only a single warp and only utilizes the GPU’s wide SIMD support for parallelism. Each index contains a key and a pointer to the element it references.

Another major issue faced by using the GPU to perform queries is high latency between the host and device. I effectively hid this by returning results incrementally. As chunks of 128 results became available they would be copied back to the host before the next 128 were found. As I’ll show later in the results section, this led to a very low amortized access speed.

In terms of optimizations I didn’t get a chance to perform as much as I would have hoped since a lot of time was spent wrestling with the database host interface and C++ templates. I’ve identified the criteria evaluation on the device as a major bottleneck, which I hope to eliminate with further optimization. Currently criteria are interpreted from a byte string on the GPU in a fashion that results in high warp divergence and poor memory efficiency. The biggest
optimization I performed on the b-tree was an implementation of inserts using a single pass versus two passes as detailed in Mond & Raz’s 1985 paper on B+-trees.

Results
I measured performance in terms of average access time for each row in a given query and compared this to average access time for each row in an equivalent SQLite query using their in memory database. I performed a series of selects on tables that ranged from size 1000 to 500,000 entries. I tested various degrees of criteria specification in the different selects. The tests revealed that performance scaled fairly linearly with database size which is to be expected with b-trees with high branching factors; a tree with 32x as many members only has a depth one greater.

Ultimately I believe speedup was not limited by memory/bus bandwidth but by my algorithm’s criteria evaluation. Criteria are compiled to byte strings that are transferred to the device and then interpreted. The implementation suffered from both warp divergence and inefficient memory compares (memcmp was used to support variable sized data). In the future I would like to explore an approach using C++ templates to generate efficient compare functions.

The following charts are the results of selecting every row from the database. It is clear we hit the PCI bus bandwidth limitation quickly and cannot service each row as fast as the main memory bus can. This average select time represents a hard limit on how fast queries can be serviced by CUDA-DB, about 400 nanoseconds. An important note is that the test program performed no computation on the selected records and so there was no chance that this latency could be hidden. If the CPU performed even minimal processing of each record it could effectively hide this latency since GPU processing is performed entirely asynchronously to the CPU. This latency hiding ability is is not something a regular in memory database can boast since they operate synchronously.
SELECT * FROM record

**Graph 1:**
- **x-axis:** Table Size
- **y-axis:** Time (ms)
- **Legend:**
  - cudadb total
  - sqlitedb total

**Graph 2:**
- **x-axis:** Table Size
- **y-axis:** Time (ms)
- **Legend:**
  - cudadb average
  - sqlitedb average
The following charts detail a filtered query. Each record in the database is assigned an id in a uniform distribution between 1 and 20 thus this query selects exactly 1/20\textsuperscript{th} of the database. We see that due to the extra processing necessary CUDA-DB and SQLite perform almost identically and scale linearly up to tables of size 500,000 records. I believe with further optimization the average select time for CUDA-DB for this sort of query could be brought down from 900 nanoseconds to the hard limit established by the previous query of 400 nanoseconds.
SELECT * FROM record WHERE id = 2

Time (ms) vs Table Size

Table:
```
<table>
<thead>
<tr>
<th>Table Size</th>
<th>cudadb total</th>
<th>sqlitedb total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0.0002</td>
<td>0.0004</td>
</tr>
<tr>
<td>0.0006</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0012</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0014</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

SELECT * FROM record WHERE id = 2

Time (ms) vs Table Size

Table:
```
<table>
<thead>
<tr>
<th>Table Size</th>
<th>cudadb average</th>
<th>sqlitedb average</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0.0002</td>
<td>0.0004</td>
</tr>
<tr>
<td>0.0006</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0012</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0014</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
The final set of charts show the impact of adding further filtering to the query. We see that while SQLite’s average select time stays approximately constant at around 800 nanoseconds, CUDA-DB’s average rises to about 1400 nanoseconds. This is evidence that the criteria evaluation on the device is severely suboptimal. In future iterations of the database I would like to address this bottleneck because there is no reason this query should perform significantly worse than the previous.
In conclusion, I believe for certain workloads where high performance is necessary, GPU hosted databases have potential due to their asynchronous nature. While the host can be performing sequential IO/processing with a client, the database can be performing queries. This can result in excellent latency hiding of query results.

Finally I believe the GPU can scale better than an in memory database in terms of simultaneous queries. Where in traditional in memory databases the only bus available must be used for all parts of a query: traversing indices, filtering data, etc., in a GPU hosted database, the PCI bus is used only for query initiation and result transmission. All memory intensive actions are performed over the high speed GPU memory bus.

References (or, thank you SQLite for your excellent documentation!)
http://www.sqlite.org/arch.html
http://www.sqlite.org/queryplanner.html
http://www.sqlite.org/optoverview.html

B-tree single pass insert algorithm
http://www.informatik.uni-trier.de/~ley/db/conf/vldb/MondR85.html