A Study on Database Cracking with GPUs

Eleazar Leal\textsuperscript{1}, Le Gruenwald\textsuperscript{2}

\textsuperscript{1}University of Minnesota Duluth
\textsuperscript{2}University of Oklahoma

This project is supported in part by the National Science Foundation (NSF) under Grants No. 1302423 and 1302439
Contents

• Background
• Algorithms Studied
• Experimental Evaluation
• Conclusions
• Future Work
Background: Database Cracking

- Assumptions:
  - No time to build indexes before queries
  - Query workload unavailable

- Cracking:
  - Current query is used as a hint for reorganizing the data

```
SELECT R.A
FROM R
WHERE R.A BETWEEN a AND b;
```
Related Work

• Single-threaded cracking algorithms
  – Standard cracking (Idreos, 2007)
  – Sideways cracking (Idreos et al., 2009)
  – Hybrid cracking (Idreos, Manegold, et al., 2011)
  – Stochastic cracking (Halim, Idreos, et al., 2012)
  – Coarse-granular indexing (CGI)
  – Predication and vectorized cracking (Pirk et al., 2014)

• Multi-threaded cracking algorithms
  – Parallel Standard Cracking (P-SC) (Graefe, Halim, Idreos, 2012)
  – Parallel Chunked CGI (P-CCGI) (Schuhknecht, Jindal, Dittrich, 2016)

Idreos, S., Kersten, M., Manegold, S. Self-organizing tuple reconstruction in column-stores. SIGMOD, 2009
Pirk, H., Petraki, E., Idreos, S., Manegold, S., Kersten, M.L. Database cracking: fancy scan, not poor man’s sort! DaMoN, 2014
Algorithms Studied: Crack GPU

Algorithm 1: Crack GPU

Input: GPU cracker column $gC$, host cracker column $C$, $a$, $b$

Output: Retrieves the values in $C$ that belong to the range $[a, b]$ and cracks the cracker column $gC$ according to $[a, b]$

1. if $numQueries = 0$ then
2. $gC \leftarrow \text{Copy } C \text{ to the GPU’s global memory; }$
3. Initialize the cracker index in the host’s memory;
4. $posLow \leftarrow \text{add}_\text{gpu-crack}(gC, a);$
5. $posHigh \leftarrow \text{add}_\text{gpu-crack}(gC, b);$
6. $result \leftarrow \text{GPuToHost}(gC, posLow, posHigh);$
7. $numQueries +=$;
8. return $result$;
Algorithms Studied: CrackGH300

Algorithm 3: Crack GPU Hybrid

**Input**: GPU cracker column $gC$, host cracker column $C$, $a, b$

**Output**: Retrieves the values in $C$ that belong to the range $[a, b]$ and cracks the cracker column $gC$ according to $[a, b]$

1. if $numQueries = 0$ then
2.     $gC \leftarrow$ Copy $C$ to the GPU’s global memory;
3.     Initialize the cracker index in the host’s memory;
4.   if $numQueries < T$ then
5.     $posLow \leftarrow add\_gpu\_crack(gC, a)$;
6.     $posHigh \leftarrow add\_gpu\_crack(gC, b)$;
7.     $result \leftarrow GPUSThost(gC, posLow, posHigh)$;
8.   else
9.     if $numQueries = T$ then
10.    Copy $gC$ to the cracker column $C$ in the host;
11.    $posLow \leftarrow add\_cpu\_crack(C, a)$;
12.    $posHigh \leftarrow add\_cpu\_crack(C, b)$;
13.    $result \leftarrow C[posLow, posHigh]$;
14. $numQueries++$;
15. return $result$;

Based on CUDA Thrust’s partition function
Experimental Setup

- **Dataset**
  - 100 million rows
  - Each row is a 4-byte integer
  - Random uniform distribution

- **Workloads**
  
  From Halim, Idreos, Karras and Yap, 2012

Figure from F. Halim, S. Idreos, P. Karras, R.H.C. Yap. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. Proceedings of VLDB 2012
Experimental Setup

- **Hardware**
  - 2x Intel Xeon Gold (12 cores) 6136 @ 3GHz
  - TurboBoost disabled
  - 128 GB of RAM
  - Nvidia Quadro P4000 with 8GB of RAM (PCIe)

- **Software**
  - g++ with -O3
  - Cuda 10.0

Figure from Amazon’s product page: https://www.amazon.com/PNY-VCQP4000-PB-NVIDIA-Quadro-P4000/dp/B06X9TNDFF
Experimental Setup

• **Competing Algorithms**
  - Non-cracking algorithms
    • Filter
    • Sort
    • Filter GPU
    • Sort GPU
  - Crack GPU algorithms
    • Crack GPU
    • Crack GPU Hybrid
  - Cracking CPU algorithms
    • Standard Cracking (single-threaded)
    • P-CCGI (16 threads)

Query:
Select R.A
From R
Where R.A ≥ 12
and R.A ≤ 20

Query Result

Cracker column

9 9 10 14

11 2 29 7

33 1 3 21

13 6 8

13 14
Experimental Setup

• **Evaluation Metrics**
  – Individual query response time
  – Cumulative query response time

• **Experimental Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Range of Values</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column Size (# rows)</td>
<td>100 to 600 million</td>
<td>100 million</td>
</tr>
<tr>
<td>Selectivity</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>Num. Queries</td>
<td>1,000 to 10,000</td>
<td>1,000</td>
</tr>
<tr>
<td>Num. Chunks (P-CCGI)</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>
COMPARISON OF NON-CRACKING ALGORITHMS
Comparison of Non-Cracking Algorithms

• **Competing Algorithms**
  – Non-cracking algorithms
    • Filter
    • Sort
    • Filter GPU
    • Sort GPU
  – Crack GPU algorithms
    • Crack GPU
    • Crack GPU Hybrid
  – Cracking CPU algorithms
    • Standard Cracking (single-threaded)
    • P-CCGI (16 threads)
Experiment Results

(a) Random, Selectivity = 0.01

Individual query response times of non-cracking algorithms
Experiment Results

Time decomposition of individual query response times of Sort GPU and Filter GPU
Experiment Results

- **Sort GPU** is the best non-cracking algorithm

- Smaller penalty (14X) on the first query than **Sort**

- Comparable penalty to that of **Filter GPU** and **Filter**

*Cumulative query response times of the non-cracking algorithms*
Comparison of Non-Cracking Algorithms

• Competing Algorithms
  – Non-cracking algorithms
    • Filter
    • Sort
    • Filter GPU
    • Sort GPU (Best non-cracking algorithm)
  – Crack GPU algorithms
    • Crack GPU
    • Crack GPU Hybrid
  – Cracking CPU algorithms
    • Standard Cracking (single-threaded)
    • P-CCGI (16 threads)
COMPARISON OF GPU CRACKING ALGORITHMS
Comparison of Non-Cracking Algorithms

• Competing Algorithms
  – Non-cracking algorithms
    • Filter
    • Sort
    • Filter GPU
    • Sort GPU
  – Crack GPU algorithms
    • Crack GPU
    • Crack GPU Hybrid
  – Cracking CPU algorithms
    • Standard Cracking (single-threaded)
    • P-CCGI (16 threads)
Experiment Results

Time decomposition of individual query response times of GPU cracking algorithms
Experiment Results

- CrackGH300 has up to 50% shorter query response times than Crack GPU on average.
Comparison of Non-Cracking Algorithms

• Competing Algorithms
  – Non-cracking algorithms
    • Filter
    • Sort
    • Filter GPU
    • Sort GPU
  – Crack GPU algorithms
    • Crack GPU
    • Crack GPU Hybrid (best GPU cracking algorithm)
  – Cracking CPU algorithms
    • Standard Cracking (single-threaded)
    • P-CCGI (16 threads)
BEST NON-CRACKING VS. BEST GPU CRACKING VS. BEST CPU CRACKING
Comparison of Non-Cracking Algorithms

• Competing Algorithms
  – Non-cracking algorithms:
    • Filter
    • Sort
    • Filter GPU
    • Sort GPU (best non-cracking)
  – Crack GPU algorithms:
    • Crack GPU
    • Crack GPU Hybrid (best GPU cracking)
  – Cracking CPU algorithms:
    • Standard Cracking (single-threaded)
    • P-CCGI (16 threads)
Experiment Results

(d) Random workload

(f) Non-uniformly random workload
Experiment Results

- **CrackGH300** is **2.4X faster** than **Standard Crack**
- **Sort GPU** and **Standard Crack** are competitive
- **P-CCGI** is **3.4X faster** than **CrackGH300**
Conclusions

- Non-random workloads
  
  Sort GPU was 2.4X faster than P-CCGI

- Random workloads
  
  P-CCGI was 1.98X faster than Sort GPU
Conclusions

1. As Alvarez, Schuhknecht, Dittrich, Richter observed:

   In parallel architectures, **sorting** is a strong competitor against **cracking**.

   In non random workloads

Conclusions

2. Crack GPU and Crack GPU Hybrid do not improve over P-CCGI because:

**Algorithm 1: Crack GPU**

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>if numQueries = 0 then</td>
</tr>
<tr>
<td>2</td>
<td>gC ← Copy C to the GPU’s global memory;</td>
</tr>
<tr>
<td>3</td>
<td>Initialize the cracker index in the host’s memory;</td>
</tr>
<tr>
<td>4</td>
<td>posLow ← add_gpu_crack(gC, a);</td>
</tr>
<tr>
<td>5</td>
<td>posHigh ← add_gpu_crack(gC, b);</td>
</tr>
<tr>
<td>6</td>
<td>result ← GPUToHost(C, posLow, posHigh);</td>
</tr>
<tr>
<td>7</td>
<td>numQueries += 1;</td>
</tr>
<tr>
<td>8</td>
<td>return result;</td>
</tr>
</tbody>
</table>

- Inserting cracks one at a time
- High transfer time from CPU to GPU and vice versa
Future Work

• Consider algorithms that simultaneously apply multiple cracks (GPU Multisplit by Askiani, Davidson, Meyer & Owens, 2016)

• Test these algorithms on a GPU with NVLink instead of PCIe

• Test more complex queries with joins

Thank you!