EXPLOITING COARSE-GRAINED PARALLELISM IN B+ TREE SEARCHES ON APUS

Mayank Daga
AMD Research

Mark Nutter
AMD Heterogeneous System Software
B+ TREE SEARCHES

- B+ Tree is a fundamental data structure used in
  - Relational Database Management Systems (RDBMS)
    - IBM DB2
    - MySQL
    - Oracle
    - SQLite
  - Key-Value Database Management Systems

- High-throughput, read-only index searches are gaining traction in
  - Audio-search
  - Video-copy detection
  - Online Transaction Processing (OLTP) Benchmarks

- Increase in memory capacity allows many database tables to reside in memory
  - Brings computational performance to the forefront
DATABASE PRIMITIVES ON ACCELERATORS

- Discrete graphics processing units (dGPUs) provide a compelling mix of
  - Performance per Watt
  - Performance per Dollar

- dGPUs have been used to accelerate critical database primitives
  - scan
  - sort
  - join
  - aggregation
  - B+ Tree Searches?
B+ TREE SEARCHES ON ACCELERATORS

- B+ Tree searches present significant challenges
  - Irregular representation in memory
    - An artifact of malloc() and new()
  - Today’s dGPUs do not have a direct mapping to the CPU virtual address space
    - Indirect links need to be converted to relative offsets
  - Requirement to copy the tree to the dGPU, which entails
    - One is always bound by the amount of GPU device memory
OUR SOLUTION

- Accelerated B+ Tree searches on a fused CPU+GPU processor (or APU\(^1\))
  - Eliminates data-copies by combining x86 CPU and vector GPU cores on the same silicon die
- Developed a memory allocator to form a regular representation of the tree in memory
  - Fundamental data structure is not altered
  - Merely parts of its layout is changed

[1] www.hsafoundation.com
OUTLINE

- Motivation and Contribution
- Background
  - AMD APU Architecture
  - B+ Trees
- Approach
  - Transforming the Memory Layout
  - Eliminating the Divergence
- Results
  - Performance
  - Analysis
- Summary and Next Steps
The APU consists of a dedicated IOMMUv2 hardware
- Provides direct mapping between GPU and CPU virtual address (VA) space
- Enables GPUs to access the system memory
- Enables GPUs to track whether pages are resident in memory

AMD 2nd Gen. A-series APU
**B+ TREES**

- A B+ Tree …
  - is a dynamic, multi-level index
  - Is efficient for retrieval of data, stored in a block-oriented context
  - has a high fan-out to reduce disk I/O operations

- Order (b) of a B+ Tree measures the capacity of its nodes

- Number of children (m) in an internal node is
  - \([b/2] \leq m \leq b\)
  - Root node can have as few as two children

- Number of keys in an internal node = \((m – 1)\)
APPROACH FOR PARALLELIZATION

- Fine-grained (Accelerate a single query)
  - Replace Binary search in each node with K-ary search
  - Maximum performance improvement = \( \log(k)/\log(2) \)
  - Results in poor occupancy of the GPU cores

- Coarse-grained (Perform many queries in parallel)
  - Enables data-parallelism
  - Increases memory bandwidth with parallel reads
  - Increases throughput (transactions per second for OLTP)
TRANSFORMING THE MEMORY LAYOUT

- **Metadata**
  - Number of keys in a node
  - Offset to keys/values in the buffer
  - Offset to the first child node
  - Whether a node is a leaf

- Pass a pointer to this memory buffer to the accelerator
ELIMINATING THE DIVERGENCE

- Each work-item/thread executes a single query
- May increase divergence within a wave-front
  - Every query may follow a different path in the B+ Tree

- Sort the keys to be searched
  - Increases the chances of work-items within a wave-front to follow similar paths in the B+ Tree
  - We use Radix Sort\textsuperscript{1} to sort the keys on the GPU

Impact of Divergence in B+ Tree Searches

Impact of Divergence on GPU – 3.7-fold (average)
Impact of Divergence on CPU – 1.8-fold (average)
OUTLINE

- Motivation and Contribution
- Background
  - AMD APU Architecture
  - B+ Trees
- Approach
  - Transforming the Memory Layout
  - Eliminating the Divergence
- Results
  - Performance
  - Analysis
- Summary and Next Steps
EXPERIMENTAL SETUP

- **Software**
  - A B+ Tree w/ 4M records is used
  - Search queries are created using
    - `normal_distribution()` (C++-11 feature)
    - The queries have been sorted
  - CPU Implementation from
  - Driver: AMD Catalyst™ v12.8
  - Programming Model: OpenCL™

- **Hardware**
  - AMD Radeon HD 7660 APU (Trinity)
    - 4 cores w/ 6GB DDR3, 6 CUs w/ 2GB DDR3
  - AMD Phenom II X6 1090T + AMD Radeon HD 7970 (Tahiti)
    - 6 cores w/ 8GB DDR3, 32 CUs w/ 3GB GDDR5
  - Device Memory does *not* include data-copy time

---

<table>
<thead>
<tr>
<th>E ID (PKey)</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000001</td>
<td>34</td>
</tr>
<tr>
<td>4  million</td>
<td></td>
</tr>
<tr>
<td>4194304</td>
<td>50</td>
</tr>
</tbody>
</table>

---

![Frequency Distribution Chart](image-url)
RESULTS – QUERIES PER SECOND

- dGPU (device memory): ~350M Queries/Sec. (avg.)
- dGPU (pinned memory): ~9M Queries/Sec. (avg.)
- Phenom CPU: ~18M Queries/Sec. (avg.)
RESULTS – QUERIES PER SECOND

APU (device memory) ~66M Queries/Sec. (avg.)
APU (pinned memory) ~40M Queries/Sec. (avg.)

APU (pinned memory) is faster than the CPU implementation
RESULTS - SPEEDUP

Baseline: six-threaded, hand-tuned, SSE-optimized CPU implementation.

Average Speedup – 4.3-fold (Device Memory), 2.5-fold (Pinned Memory)

• Efficacy of IOMMUv2 + HSA on the APU

<table>
<thead>
<tr>
<th>Platform</th>
<th>Size of the B+ Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&lt; 1.5GB</td>
</tr>
<tr>
<td>Discrete GPU (memory size = 3GB)</td>
<td>✓</td>
</tr>
<tr>
<td>APU (prototype software)</td>
<td>✓</td>
</tr>
</tbody>
</table>
ANALYSIS

- The accelerators and the CPU yield best performance for different orders of the B+ Tree
  - CPU $\rightarrow$ order = 64
    - Ability of CPUs to prefetch data is beneficial for higher orders
  - APU and dGPU $\rightarrow$ order = 16
    - GPUs do not have a prefetcher $\rightarrow$ cache line should be most efficiently utilized
    - GPUs have a cache-line size of 64 bytes
      - Order = 16 is most beneficial (16 * 4 bytes)
**ANALYSIS**

- Minimum batch size to match the CPU performance

<table>
<thead>
<tr>
<th></th>
<th>Order = 64</th>
<th>Order = 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>dGPU (device memory)</td>
<td>4K queries</td>
<td>2K queries</td>
</tr>
<tr>
<td>dGPU (pinned memory)</td>
<td>N.A.</td>
<td>N.A.</td>
</tr>
<tr>
<td>APU (device memory)</td>
<td>10K queries</td>
<td>4K queries</td>
</tr>
<tr>
<td>APU (pinned memory)</td>
<td>20K queries</td>
<td>16K queries</td>
</tr>
</tbody>
</table>

- `reuse_factor` - amortizing the cost of data-copies to the GPU

\[
\begin{align*}
  Time_{accel} & = T_{copy} + (T_{accl\,Exec} \times reuse\_factor) \\
  Time_{cpu} & = T_{cpu\,Exec} \times reuse\_factor \\
  \text{or } reuse\_factor & \leq \frac{T_{copy}}{T_{cpu\,Exec} - T_{accl\,Exec}}
\end{align*}
\]

<table>
<thead>
<tr>
<th></th>
<th>90% Queries</th>
<th>100% Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>dGPU</td>
<td>15</td>
<td>54</td>
</tr>
<tr>
<td>APU</td>
<td>100</td>
<td>N.A.</td>
</tr>
</tbody>
</table>
```c
int i = 0, j;
node * c = root;
__m128i vkey = _mm_set1_epi32(key);
__m128i vnodekey, *vptr;
short int mask;
/* find the leaf node */
while( !c->is_leaf ){
    for( i = 0; i < (c->num_keys-3); i+=4){
        vptr = (__m128i *)&(c->keys[i]);
        vnodekey = _mm_load_si128(vptr);
        mask = _mm_movemask_ps(_mm_cvtepi32_ps(_mm_cmplt_epi32(vkey, vnodekey)));
        if((mask) & 8) break;
    }
    for( j = i; j < c->num_keys; j++){
        if(key < c->keys[j]) break;
    }
    c = (node *)c->pointers[j];
}
/* match the key in the leaf node */
for (i = 0; i < c->num_keys; i++)
    if (c->keys[i] == key) break;
/* retrieve the record */
if( i != c->num_keys )
    return (record *)c->pointers[i];
```
RELATED WORK

- J. Fix, A. Wilkes, and K. Skadron, "Accelerating Braided B+ Tree Searches on a GPU with CUDA." In Proceedings of the 2nd Workshop on Applications for Multi and Many Core Processors: Analysis, Implementation, and Performance, in conjunction with ISCA, 2011
  - Authors report ~10-fold speedup over single-thread-non-SSE CPU implementation, using a discrete NVIDIA GTX 480 GPU (do not take data-copies into account)

  - Authors report ~100M queries per second using a discrete NVIDIA GTX 280 GPU (do not take data-copies into account)

  - Applicable for B+ Tree modifications on the GPU
SUMMARY

- B+ Tree is the fundamental data structure in many RDBMS
  - Accelerating B+ Tree searches is critical
    - Presents significant challenges on discrete GPUs

- We have accelerated B+ Tree searches by exploiting coarse-grained parallelism on a APU
  - 2.5-fold (avg.) speedup over 6-threads+SSE CPU implementation

- Possible Next Steps
  - HSA + IOMMUv2 would alleviate the issue of modifying B+ Tree representation
    - Investigate CPU-GPU co-scheduling
  - Investigate modifications on the B+ Tree
Trademark Attribution

AMD, the AMD Arrow logo and combinations thereof are trademarks of Advanced Micro Devices, Inc. in the United States and/or other jurisdictions. Other names used in this presentation are for identification purposes only and may be trademarks of their respective owners.

©2012 Advanced Micro Devices, Inc. All rights reserved.