Exploiting Coarse-grained Parallelism in B+ Tree Searches on an APU

Mayank Daga
AMD Research
Advanced Micro Devices, Inc., USA
Mayank.Daga@amd.com

Mark Nutter
Heterogeneous System Software
Advanced Micro Devices, Inc., USA
Mark.Nutter@amd.com

***Abstract*—B+ tree structured index searches are one of the fundamental database operations and hence, accelerating them is essential. GPUs provide a compelling mix of performance per watt and performance per dollar, and thus are an attractive platform for accelerating B+ tree searches. However, tree search on discrete GPUs presents significant challenges for acceleration due to (i) the irregular representation in memory and (ii) the requirement to copy the tree to the GPU memory over the PCIe bus.**

In this paper, we present the acceleration of B+ tree searches on a fused CPU+GPU processor (an accelerated processing unit or APU). We counter the aforementioned issues by reorganizing the B+ tree in memory and utilizing the novel heterogeneous system architecture, which eliminates (i) the need to copy the tree to the GPU and (ii) the limitation on the size of the tree that can be accelerated. Our approach exploits the coarse-grained parallelism in tree search, wherein we execute multiple searches in parallel to optimize for the SIMD width *without* modifying the inherent B+ tree data structure. Our results illustrate that the APU implementation can perform up to 70M[1] queries per second and is *4.9x faster* in the best case and *2.5x faster* on average than a hand-tuned, SSE-optimized, six-core CPU implementation, for varying orders of the B+ tree with 4M keys. We also present an analysis of the effect of caches on performance, and of the efficacy of the APU to eliminate data-copies.

Keywords: B+ Tree; APU; Coarse-Grained Parallelism; AMD; Performance Acceleration.

I. Introduction

B+ trees continue to be popular in database applications due to their remarkable efficiency in retrieval of the stored data [1]. High-throughput, read-only index searches are gaining traction in fields like audio-search and video-copy detection [2], [3]. Traditionally, database operations were bottlenecked by disk bandwidth, so increasing the computational capacity of the processor did not have a substantial impact on the performance of index searches. However, the increase in memory capacities over the years now allows many database tables to reside in memory, thereby eliminating the disk I/O and bringing the computational performance to the forefront.

Graphics processing units (GPUs) are being adopted increasingly due to their remarkable performance-price ratio [4]. GPUs that reside across the PCIe bus, known as discrete GPUs and referred to as dGPUs henceforth, are effective in accelerating applications that (i) demonstrate regular memory access patterns and (ii) amortize the cost of copying data to the dGPU. Researchers have recently employed dGPUs to accelerate several critical database primitives like scan, sort, join, and aggregation [5], [6]. Unlike these primitives, index searches on dGPUs present significant challenges due to (i) the irregular representation in memory and (ii) the requirement to copy the tree to the dGPU [7].

Irregular memory representation is an artifact of using dynamic memory allocators for building the tree on the CPU. Today's dGPUs do not have a direct mapping to the CPU virtual address space, thereby requiring indirect links in the tree to be converted into relative offsets. Once the links are converted, traversing a tree residing in system memory from the dGPU does not yield optimum performance, requiring the tree to be copied to dGPU. The process of copying the tree, based on the size, can take orders of magnitude longer than performing the search itself. A side-effect of copying the tree is that one is always limited by the memory size on a dGPU, which is at most 8 GB.

In this paper, we accelerate B+ tree structured index searches on a fused CPU+GPU processor (or APU). APUs are AMD's implementation of the Heterogeneous Systems Architecture (HSA) and help eliminate data-copies by combining the general-purpose x86 CPU cores with the programmable vector-processing engines of a GPU on a single silicon die [8]. To the best of our knowledge, we are the first to implement index searches on the APU. We overcome the issue of irregular memory representation of the B+ tree by developing our own memory allocator that ensures all nodes of the tree are laid out contiguously in memory. We do *not* alter the fundamental data structure; we merely alter parts of its layout. The tree is then traversed using offsets into the contiguously allocated memory region.

One index search on the B+ tree is independent from all other index searches. This paradigm perfectly suits the SIMD nature of the APUs and hence, enables us to exploit the coarse-grained parallelism in index searches. We perform a detailed experimental analysis on the effect of divergence as well as the order (fan-out) of the B+ tree in our imple-

[1]Throughout this article K refers to thousand, M refers to million, and B refers to billion.

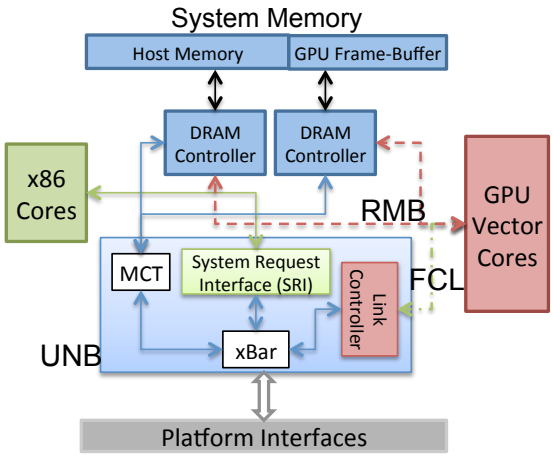


Figure 1. Block diagram of the AMD A10 APU architecture, codenamed Trinity [10]. UNB - Unified Northbridge, MCT - Memory Controller, RMB - Radeon Memory Bus, FCL - Fusion Compute Link

mentation. We illustrate that the divergence affects the APU as well as the CPU. We mitigate thread-divergence by first sorting all the keys to be searched [9]. The presence of a prefetcher on the CPU and the difference in the behavior of caches leads the CPU and the APU to yield optimum performance for different orders of the B+ tree.

Specifically, the APU performs best when a node of the tree fits exactly in a cache-line, whereas best performance on the CPU is achieved when a node is substantially larger than the cache-line size. We also present an analysis of data-copies and conclude that a tree should at least be used *54 times* to amortize the cost of copying it to the dGPU. Our results indicate that the APU implementation can perform up to 70M queries per second for a B+ tree with 4M keys, and is *4.9x faster* in the best case and *2.5x faster* on average than a hand-tuned, SSE-optimized, six-core CPU implementation. The APU outperforms the CPU even when the tree is not copied, illustrating its capability in eliminating the data-copies and ameliorating the constraint on the size of the tree that can be accelerated.

In the rest of this paper, Section II provides a background on the AMD APU architecture and B+ trees. Section III explains our APU implementation and the memory allocator used to generate the transformed memory layout of the B+ tree, followed by how we mitigate thread-divergence. Section IV discusses the experimental results and presents an analysis on (i) the effect of caches and (ii) the cost of data-copies. Section V presents the related work. Section VI proposes future work and presents a conclusion.

II. Background

In this section, we present a background information on the APUs as well as the B+ tree data structure.

A. Accelerated Processing Units, or APUs

Fundamentally, an APU combines general-purpose scalar and vector processor cores on a single silicon die, thereby forming a true heterogeneous computing processor. Figure 1 depicts a block diagram of the AMD A10 APU architecture, codenamed Trinity (i.e., the APU in its latest incarnation). An important aspect of the APU is that it allows both the x86 CPU and the vector GPU cores to access system memory via the DRAM controllers, albeit through different paths internally. This architectural artifact allows the APU to alleviate the fundamental PCIe constraint that has traditionally limited the performance on a dGPU. The APU also consists of an I/O controller, a unified video decoder, a display output, and bus interfaces, all on the same die.

The GPU cores on the APU can access both the cached and uncached regions of the system memory. They do so via two different memory buses called the Fusion Compute Link (FCL) and the Radeon™Memory Bus (RMB), respectively. The FCL is 128 bits wide in each direction and connects the graphics memory controller to the unified northbridge (UNB). The UNB contains the system request interface (SRI), which is capable of snooping the caches. In-case of a cache miss, system memory can be accessed via the DRAM controllers from the UNB. The RMB is 256 bits wide in each direction, per memory channel, and directly connects the graphics memory controller to the DRAM controllers. The APU also provides a dedicated GPU framebuffer, analogous to device memory on a dGPU, for full bandwidth access. The framebuffer is a region of the system memory managed by the GPU cores and accessed using the RMB. It bypasses the cache coherency mechanism, so is the fastest path to system memory from the GPU. The x86 cores can also access the framebuffer using the FCL.

The APU consists of a dedicated IOMMUv2 hardware that can provide a direct mapping between the GPU and the CPU virtual address space. IOMMUv2 also allows the GPU to track whether a page is resident in memory and bring it in memory on demand, which means that an application is no longer limited to the amount of available system memory or to the size of the GPU framebuffer. However, in the present-generation APUs, the GPU cores can access the x86 virtual address space only at a granularity of continuous chunks of memory, and not at a granularity of one page. This limitation curbs the programmer to pass a host-side pointer to the GPU and use indirect addressing mechanisms to access the entire memory region.

The GPU cores on the AMD Trinity APU feature a VLIW4 design in which each SIMD engine consists of 16 four-way VLIW thread processors. Each thread processor consists of one branch execution unit. There are six SIMD engines in total, amounting to 384 ALUs on the entire GPU. Each SIMD engine has its own local data store and texture caches. The GPU can run at a maximum of 800 MHz. Programming on the APU is facilitated by the emerging OpenCL™ standard [11]. We use OpenCL™ terminology in the rest of this article.

B. B+ Trees

B+ trees were instrumental in optimizing the disk I/O by virtue of several characteristics: (i) they have a very high fan-out (i.e., large number of branches at each node); (ii) they are always height-balanced (i.e., all the leaf nodes are at the same level); (iii) they keep related data within the same block or a disk page, which takes advantage of the locality of reference; and (iv) they guarantee that all the nodes are full at least to a certain percent, thus improving space efficiency. For example, a B+ tree with a branching factor of 1001 and a height of two can store more than 1B keys, and it requires only two disk accesses to find a key because the root node can be stored in main memory.

In a B+ tree, the records (or values) are stored only in the leaf nodes, whereas the internal nodes store the search-keys for those records. The branching factor or the *order* of a B+ tree measures the capacity of each of its nodes. A node in a B+ tree of *order* 'm' contains up to 'm' branches (or children) and up to 'm-1' search-keys. The leaf nodes of the tree are linked to form a linked list to allow faster processing of range-search queries. The order of a B+ tree is instrumental in affecting the memory access patterns because a higher-order tree translates to less height and hence, fewer disk or memory accesses. A well-known optimization technique to minimize disk accesses is to have the order of a tree be such that each node of the tree fits in one page of the disk. However, for in-memory databases it may be worthwhile to investigate having trees with a smaller order to optimize for the number of cache hits [12].

Index search in a B+ tree is performed as an iterative process between searching the keys in a node and then traversing to the next node in the tree, until the value being searched is found in one of the leaf nodes.

III. Implementation

In this section, we present an overview of our APU implementation followed by a deep dive into the details of our memory allocator which transforms the B+ tree to a contiguous memory layout. We then discuss how we eliminate thread-divergence.

A. Overview

Today's GPU cores lack the ability to indirectly address the x86 virtual address space (i.e., one cannot use pointers to traverse from one node to another node in the tree, as mentioned in Section II-A). However, a tree is generally built using the dynamic memory allocation functions like malloc() and new() in the CPU realm, and hence cannot be traversed on the GPU as-is. Therefore, the entire tree must be laid out in a contiguous memory location and traversed using offsets, as described in Section III-B.

B+ tree searches can be accelerated using the following two approaches: (i) accelerating a single query across the entire APU (i.e., via fine-grained parallelism), or (ii) performing multiple queries simultaneously in parallel (i.e., via coarse-grained parallelism). The fine-grained approach replaces the binary search to be performed at every node to find the next node in the search path, by K-ary search, and parallelizes it across the APU. The maximum performance benefit one can obtain by this approach is log(K)/log(2). To keep the APU busy and to hide the memory-access latency, a node should contain tens of thousands of elements to be searched. Realistically, databases employ B+ trees with at most 200 elements in every node. Therefore, this approach does not provide enough work to maintain high occupancy of the APU. The coarse-grained approach, on the other hand, executes batches of queries in parallel. Because search-queries in a B+ tree are data-parallel, they efficiently map onto the APU. The coarse-grained approach also benefits from the parallel reads from various work-items on the APU, which results in better utilization of the available memory bandwidth.

Specifically, we assign one work-item on the APU to execute one search-query. We launch 256 work-items, or four wavefronts, in a work-group to have enough work-items in flight to hide the memory-access latency. A maximum of 32,768 work-items (the upper bound on the APU) are launched; if the number of queries in a batch is greater than that, more than one query is assigned to a work-item. We do not launch more work-items to reduce the scheduling overhead. We perform register preloading to utilize the registers present in the SIMD engine efficiently [13]. The top of the tree may get cached in the L1 cache of the SIMD engines because it is accessed by all the work-items, though that depends on the order of the tree. Higher order entails a node being wider, so it may not fit in a cache line. Section IV presents an analysis between performance and the order of the tree. The kernel for our APU implementation is shown in Figure 2.

The coarse-grained approach results in a substantial amount of divergence within a wavefront because adjacent work-items may follow completely different paths in the tree, as illustrated in Figure 3. In the figure, work-item #1 is tasked to search for key 1 and work-item #2 is tasked to search for key 7. Both of these keys reside on different paths from the root and result in divergence. The execution of divergent branches is serialized because there is only one instruction sequencer present per SIMD engine to perform the predication. Ideally, the adjacent work-items should follow similar paths as much as possible. Referring back to Figure 3, this means that work-item #2 should search for key 2 and so on. One way to achieve this execution pattern is to sort the search-keys before assigning them to work-items, as described in Section III-C.

B. Transforming the Memory Layout

We developed a memory allocator to ensure that the tree is laid out in a continuous memory region. Specifically, we

```
1  // TYPE is the datatype of the <key, value> pair
2  global void B+TreeSearch(void *root, TYPE *search_keys,
3                           TYPE *values) {
4      int tid = get_global_id(0);
5      int lid = get_local_id(0);
6      node n = ((global node *) root)[0];
7      TYPE key_to_search = search_keys[tid];
8      local TYPE values_found[256];
9
10     // find the leaf node
11     while( !n->is_leaf) {
12         while( n->num_keys_in_node) {
13             /* find the first key in the node
14             greater than key_to_search */
15         }
16         n = (root + offset_into_next_child_node);
17     }
18
19     // match the key in the leaf node
20     while( n->num_keys_in_node) {
21             /* break when a (key ==  key_to_search) */
22     }
23
24     // retrieve the value
25     values_found[lid] = (root + offset_to_the_value);
26     barrier(CLK_MEM_FENCE);
27     values[tid] = values_found[lid];
28 }
```

Figure 2. Pseudocode of the APU kernel for B+ tree searches.

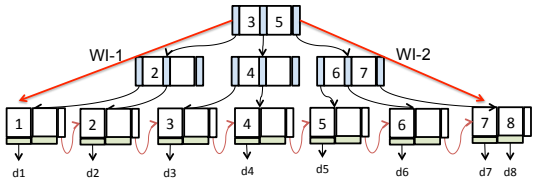


Figure 3. Illustration of divergence in the coarse-grained approach of B+ tree searches. (WI - Work-Item).

allocate a sufficiently large buffer and represent the tree in it in the following order: first, all the nodes along with some metadata; next, all the keys for every node; and, finally, all the values for those keys. A pictorial representation of our tree layout is illustrated in Figure 4. If the tree outgrows the allocated buffer, we allocate a new buffer that is twice as big as the previous one and copy the tree into it. We do *not* modify the inherent data structure and only lay the B+ tree in a different way. We do not store the keys in the nodes to keep the nodes small, and fit as many nodes as possible in a cache-line of the GPU. There is a high probability that the adjacent nodes and keys would be traversed by various work-items, thereby improving the memory subsystem performance. The metadata allows us to access the keys and values present elsewhere in the buffer; it consists of (i) number of keys, (ii) offset to the first key, (iii) offset to the first child node/value and (iv) a flag to check whether the node is a leaf node.

The buffer is created using the OpenCL™ flag `CL_MEM_ALLOC_HOST_PTR`, which lets the GPU access the host memory either by pinning it or via IOMMUv2. A pointer to the buffer is then passed as an argument to the OpenCL™ kernel as shown in Figure 2. The process of transforming the memory layout of the tree is not free but it

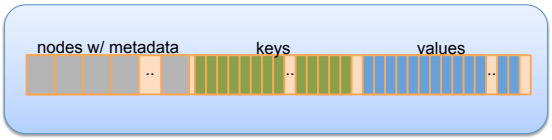


Figure 4. Pictorial representation of the modified B+ tree in memory.

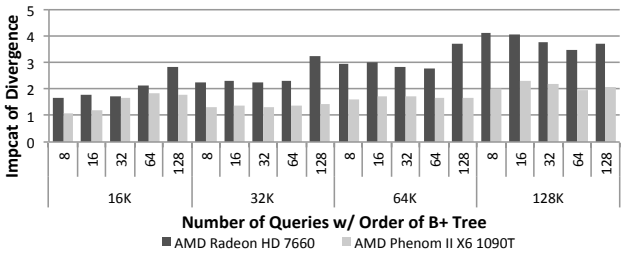


Figure 5. Impact of divergence due to the coarse-grained approach.

is likely to be performed only once. The cost of transforming the memory layout decreases with the increase in the order of the tree (i.e., wider tree results in faster transformation). This is because a deeper tree results in a greater number of memory accesses, which adversely affects performance. With the advances in the AMD HSA, this step is expected to be eliminated in future.

C. Eliminating the Divergence

The coarse-grained parallelization approach results in a high number of divergent branches. To overcome this challenge, we sort the search-keys before assigning them to the work-items. This increases the probability that adjacent work-items follow similar paths in the tree. We use radix sort for this purpose and our implementation is based on the work by Merrill et al. [14].

Figure 5 illustrates the impact of divergence due to the coarse-grained approach on the AMD Trinity APU and the AMD Phenom™ II X6 1090T CPU, for varying orders and number of queries of the B+ tree. The impact has been computed by executing the searches on a B+ tree with 4M keys, with and without sorting the keys, and calculating the difference in execution times of only the kernel. The average impact on the APU is 3.0x while that on the CPU it is 1.8x. The maximum impact on the APU can be as high as *4.1x*. The lower impact on the CPU can be attributed to the fact that the CPU has only four-way wide SIMD units, whereas the GPU cores have 64-way wide SIMD units. Sorting the search-keys does not incur substantial costs because no extra data-copies are required.

IV. Results and Discussion

In this section, we first explain our experimental set-up and illustrate the performance results of our coarse-grained B+ tree searches on the APU compared to a hand-tuned, SSE-optimized, six-core CPU implementation. For the sake

of completeness, we also compare our results to a high-end dGPU. We then present an analysis of (i) the relationship between the order of the B+ tree and the performance achieved on all three platforms and (ii) the cost of data-copies on the APU and the dGPU.

A. Experimental Set-up

We have used two accelerators for our study: (i) an AMD A10 Trinity APU that combines a four-core AMD FX CPU, codenamed Piledriver, running at a base clock of 3.8 GHz and an AMD Radeon HD 7660 GPU, and (ii) an AMD Radeon HD 7970 dGPU, codenamed Tahiti. An overview of the accelerators is presented in Table I. These were programmed via the AMD OpenCL™ SDK v2.6 with OpenCL™ v1.2 programming model and the AMD Catalyst™ driver version 12.8. The test machine with the APU has 4 GB of 1600 MHz DDR3 SDRAM. The host machine for the dGPU consists of an AMD Phenom™ II X6 1090T six-core CPU running at 3.2 GHz with 8 GB DDR3 SDRAM. The operating system used is a 64-bit version of Windows 7. All of our results are an average of 500 runs.

Table I
OVERVIEW OF ACCELERATORS

Accelerator	**AMD Trinity A10 APU**	**AMD Radeon HD 7970 GPU**
CPU	4-core Piledriver CPU	N.A.
Compute Units (CU)	6	32
VLIW	4-way	N.A.
Processing Elements	384	2048
Core Clock Rate	800 MHz	925 MHz
Memory Clock Rate	1600 MHz	1375 MHz
Memory Bus type	DDR3	GDDR5
Frame Buffer size	1024 MB (configurable)	3072 MB
Peak Memory Bandwidth	25.6 GB/s	264 GB/s
Local Memory (LDS) per CU	32 KB	64 KB
L1 Cache Size per CU	8 KB	16 KB
Registers per CU	256 KB	64 KB
Single Precision FP Performance	736 GFLOPS	3,789 GFLOPS

For experimental purposes, we have created a B+ tree with 4M entries. An example database table for such a tree is shown in Table II. For the CPU implementation, the tree is represented in memory in the original layout without any transformations. Batches of keys ranging from 16K-128K search-keys are created using *normal_distribution()* and the Mersenne Twister pseudo-random number generator in C++-11 to ensure that our tests access the entire tree. We use sorted keys on all three platforms. We based our CPU implementation on the implementation from Amittai Aviram [15]. It has been extended to use hand-tuned SSE intrinsics and parallelized using the OpenMP™ programming model. The performance numbers have been collected for varying orders of the B+ tree, ranging from four to 128.

Table II
SAMPLE DATABASE TABLE USED

Emp. ID (P Key)	**Age**
0000001	34
4 million entries	
4194304	50

B. Performance

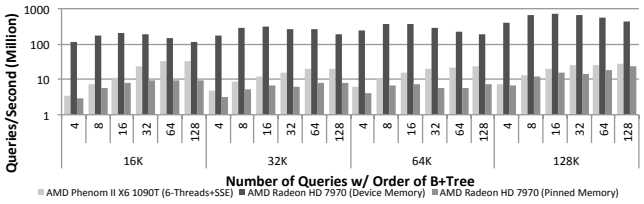
In this section, we present the queries per second (in millions) (MQPS) achieved by our coarse-grained implementation of B+ tree searches on the following three platforms: (i) a 6-core CPU, (ii) an APU and (iii) a high-end dGPU. For the accelerators, we illustrate results for both when the tree is present in the GPU device memory[1] and in the system memory (referred to as pinned memory henceforth). When using pinned memory, the buffer has been marked cacheable.

Figure 6a shows the MQPS achieved by our implementation on a six-core CPU and a dGPU. When the tree resides in the device memory, our implementation on the dGPU can perform *346 MQPS (average)* and *714 MQPS (best-case)*. When the tree does not reside in the device memory, the dGPU can perform only 10 MQPS on average. This is because every memory access needs to go over the PCIe bus that is an order of magnitude slower than the dGPU memory subsystem. The optimized CPU implementation can perform 18 MQPS. Therefore, the only way to achieve better performance on a dGPU is by copying the entire tree to the dGPU device memory, which limits the size of the tree that can be operated on.

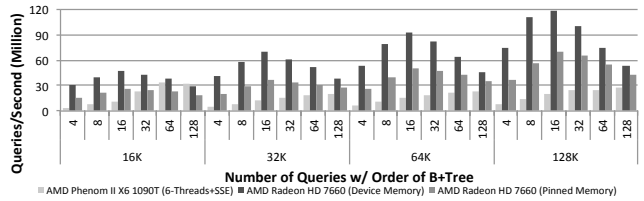
Figure 6b shows the MQPS achieved by our implementation on a six-core CPU and an APU. When the tree resides in the device memory, our APU implementation can perform *66 MQPS (average)* and *118 MQPS (best-case)*. When the tree resides in the pinned memory, the APU can perform *40 MQPS (average)* and *70 MQPS(best-case)*. These numbers are as expected when one takes into account that the APU is approximately only *one-fifth* as powerful as the dGPU. The RMB is used when the tree resides in the device memory and the FCL is used when the tree is in the pinned memory. The performance while using pinned memory is affected because the FCL is slower than the RMB, as described in Section II-A. The APU implementation is faster than the 18 MQPS achieved on the CPU even when the tree is not copied to the device memory, thereby demonstrating the potential of the APU in eliminating the data-copies and alleviating the constraint on the size of the problem to be accelerated.

Figure 6 shows that the performance on accelerators improves with the increase in number of keys to be searched, irrespective of the order of the tree and its location in memory. This is because the large number of searches translate to a larger number of work-items to be launched,

[1]Results for device memory do *not* include the time to copy the data to the device.



(a) AMD Radeon HD 7970 Tahiti dGPU



(b) AMD A10 Trinity APU

Figure 6. Performance.

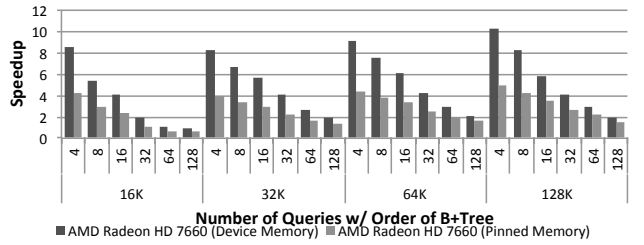


Figure 7. Speed-up. Baseline: six-threaded, hand-tuned, SSE-optimized CPU implementation.

which in turn results in improved latency hiding of memory accesses and hence, better performance. We also note that the CPU and the accelerators provide optimum performance for different orders of the B+ tree, for reasons explained in Section IV-C.

In Figure 7, we present the speed-up attained by our APU implementation compared to a six-threaded, hand-tuned, SSE-optimized CPU implementation. We note that the maximum of *10x* speed-up is attained for device memory and order = 4. The average speed-up when device memory is used is 4.3x. Though the orders 4 and 8 do not form a realistic use-case for B+ trees, these results are significant for application domains that use quad- or oct-trees. When the tree has not been copied to the device memory, the average speed-up is *2.5x* while the maximum speed-up attained is *4.9x*. Barring the case when the number of search-queries is 16K, the pinned memory implementation on the APU always outperforms the CPU.

We also developed a proof of concept to demonstrate the efficacy of the IOMMUv2 and the AMD HSA on the APU.

Platform	Size of the B+ Tree		
	< 1.5GB	1.5GB – 2.7GB	> 2.7GB
Discrete GPU *(memory size = 3GB)*	✓	✓	✗
APU *(prototype software)*	✓	✓	✓

Figure 8. Efficacy of IOMMUv2 + AMD HSA on the APU.

Figure 8 illustrates that the APU can traverse a tree larger than the memory size on a dGPU. Specifically, our tests traversed trees with up to 110M keys on the APU. We collected this result using *prototype* software wherein we modified the OpenCL™ runtime to remove the restriction on the size of the OpenCL™ buffer that can be created. Therefore, the HSA and IOMMUv2 will enable two advantages, which are especially beneficial for the big data applications: (i) eliminating data-copies and (ii) eliminating the limitation on the amount of memory available.

C. Analysis

Figure 6 demonstrates that considering any one batch size, both CPU and the accelerators achieve the best-case performance for different orders of the B+ tree. For the CPU, this order is 64 and for the accelerators, it is 16. Higher orders yield better performance on the CPU. We believe that this is due to the ability of the CPUs to prefetch data. The keys and child-offsets in a node are laid out sequentially which results in a simple memory access pattern, thereby enabling the CPU prefetcher to be quite effective in loading this data into the cache before it is required. For larger orders of the B+ tree, the prefetcher is more efficient because the data prefetched is actually used by the application, thus resulting in more cache hits. For smaller orders, the application often requires a new node, that may not be laid out next to the current node in the memory, and so the CPU prefetcher does not provide substantial benefit. This is corroborated by the profiling numbers provided by the AMD CodeAnalyst [16]. Larger orders result in a fewer number of cache misses than the tree with small orders.

The accelerators have a cache-line size of 64 bytes but they do not have a prefetcher. Therefore, the ideal performance should be obtained when an entire cache-line is most efficiently used. In our accelerated implementation, the keys and values, both 4 bytes each, are laid out sequentially, which means that a B+ tree with an order of 16 should be most efficient. An order of 16 means each node approximately holds 64 bytes to store 16 keys or values (i.e., $16 * 4 = 64$). The same is corroborated by Figure 6 which depicts that the best performance on the APU or the dGPU is achieved when the order of the B+ tree is 16.

Given that we know the order of the tree for which the CPU and the accelerators perform the best (i.e., 64 and 16, respectively), we computed the minimum batch size required to match the CPU performance. For order = 16, the dGPU (device memory) requires at least 2K search-keys batched together, whereas for order = 64 it requires 4K batched

queries. Having the tree resident in the pinned memory does not enable the dGPU to match CPU's performance for any number of queries. When using the pinned memory, the APU requires 10K and 20K queries, whereas when using the device memory, it requires 4K and 10K queries for orders 16 and 64, respectively.

Figure 6 shows that the performance on both the dGPU and the APU is much better when the tree is copied to device memory because device memory provides accesses at a higher bandwidth than system memory. However, when one takes into account the cost of copying the data to the device, the performance ceases to be impressive, at least if the tree is used only once to perform the search. We performed an analysis to compute the *reuse_factor*, defined as the number of times a tree must be reused to amortize the cost of copying the tree to the accelerator. The equations we used are:

$$\begin{aligned} Time_{accel} &= T_{copy} + (T_{acclExec} * reuse\_factor) \\ Time_{cpu} &= T_{cpuExec} * reuse\_factor \\ \text{or } reuse\_factor &<= T_{copy}/(T_{cpuExec} - T_{acclExec}) \end{aligned}$$

Table III depicts the reuse_factor for the accelerators to perform as well as the CPU for two cases: (i) for 90% of the total queries and (ii) for all queries. The table illustrates that copying the tree to the accelerator comes at a great cost that cannot always be amortized, as in the case of the APU for 100% queries, and should be avoided when possible. This further strengthens the case for accelerators to eliminate the data-copies for optimal performance, which the APUs have proved capable of accomplishing, as illustrated in figure 7.

Table III
REUSE FACTOR TO AMORTIZE THE COST OF COPYING

Platform	90% Queries	100% Queries
dGPU	15	54
APU	100	N.A.

V. RELATED WORK

B+ trees were designed to accelerate disk-based database management systems [17]. As main memory capacities have increased, substantial research has been carried out on the CPUs to optimize in-memory databases. Lehman et al. proposed T-trees specifically tuned for the main memory index structure [18]. Rao et al. argued that although T-trees provide less storage overhead, they are cache-inefficient and proposed the use of cache-conscious B+ trees called the CSB+ trees [12]. Research has also been carried out to find the optimum node size of a B+ tree. In [19], Hankens et al. proposed the node size of the tree should be greater than the cache-line size for efficient use of the TLB. Prefetching has also been proposed to improve the performance of B+ tree searches by optimizing for both disk I/O and the caches [20], [21].

The recent rise in the adoption of dGPUs has made them a popular platform to accelerate database searches. In [22], Kim et al. presented a novel, architecture-sensitive layout of the index tree to accelerate searches on modern CPUs and dGPUs. They proposed the use of a binary tree optimized for architecture features like page size, cache-line size, and the SIMD width. In [23], [24], authors proposed the use of a dictionary structure to benefit from the massive parallelism on the dGPUs to accelerate relational query co-processing. Bakkum et al. accelerated SQLite queries on the dGPU by transforming the database to a row-column format [25]. Therefore, all of them proposed a change in the inherent data structure used by the databases (i.e., the B+ tree).

Sewall et al. presented latch-free modifications to B+ trees on many-core processors using the bulk synchronous parallel model to perform multiple queries on in-memory B+ trees [26]. Heimel et al. proposed the use of dGPUs for query optimization which however, is not as significant as the actual query execution [27]. They argued that the query execution phase is plagued by the fundamental constraints of dGPUs, and hence is not suitable for acceleration. Fix et al. used braided parallelism to accelerate B+ tree searches on a dGPU [28]. They used a modified memory representation of a B+ tree that can be effectively used on the dGPU.

In this paper, we accelerated B+ tree searches on the APU. To the best of our knowledge, we are the first to do so. APUs, while preserving all the virtues of dGPUs, improve on their biggest criticism, which is that they eliminate the need to copy the data to the GPU. Our approach does not modify the inherent data structure used in databases, unlike those discussed. All we do is modify the representation of the B+ tree in memory, a step that is expected to be eliminated in coming years with the advances in the AMD HSA.

VI. CONCLUSIONS AND FUTURE WORK

B+ trees are used heavily in database management systems; hence, accelerating tree search is critical. Accelerating data-parallel problems is the stronghold of dGPUs. Although tree search is data-parallel, it presents significant challenges for acceleration, primarily due to the irregular memory representation of the tree and the cost of copying the tree to the dGPU. To overcome these challenges, we use the accelerated processing unit (APU) to accelerate B+ tree searches. The APU helps eliminate the need to copy the data over the slow PCIe bus. We reorganize the B+ tree in memory to form a regular representation and exploit the coarse-grained parallelism in tree searches. Of particular importance, we do *not* modify the inherent data structure used in the databases. Our APU implementation can perform up to 70M queries per second and results in a *4.9x (best-case) and 2.5x (on average)* speed-up over a six-threaded, hand-tuned, SSE-optimized, CPU implementation.

Our implementation, at present, requires us to reorganize the B+ tree in a contiguous memory region. We particularly

want to eliminate this step with the use of IOMMUv2 and the AMD HSA. The use of the similar data structure on both the CPU and GPU cores opens up the possibility of CPU-GPU co-scheduling, which is also assisted by the fact that the APU bridges the gap between them like never before. In the future, we would also like to enhance our APU implementation to perform modifications on the B+ tree and then work on accelerating a real-world database management system on the APU.

ACKNOWLEDGMENTS

We thank Jay Cornwall and Sean Keely for helping with the technical aspects, Takahiro Harada and Lee Howes for providing the radix sort implementation, and Dinesh Manocha and Vinod Tipparaju for providing feedback on the manuscript.

REFERENCES

[1] J. Gray and A. Reuter, "Transaction processing: Concepts and techniques," 1993.

[2] A. Wang, "An industrial strength audio search algorithm," in *ISMIR*, 2003.

[3] H. Jegou, J. Delhumeau, J. Yuan, G. Gravier, and P. Gros, "Babaz: A large scale audio search system for video copy detection," in *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, March 2012.

[4] "The Top500 Supercomputer Sites," http://www.top500.org.

[5] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner, "Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units," Aug 2009.

[6] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "Gputerasort: High performance graphics coprocessor sorting for large database management." SIGMOD, 2006.

[7] T. Kaldewey, J. Hagen, A. Di Blas, and E. Sedlar, "Parallel search on video cards," in *Proceedings of the First USENIX conference on Hot topics in parallelism*, ser. HotPar'09. Berkeley, California, USA: USENIX Association, 2009.

[8] "The HSA Foundation," 2012, http://hsafoundation.com/.

[9] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," *ACM Trans. Graph.*, vol. 27, no. 3, Aug. 2008.

[10] S. Nussbaum, "AMD Trinity Fusion APU," in *Proceedings of the Hot Chips: A Symposium on High Performance Chips*, 2012.

[11] A. Munshi, "The OpenCL Specification," 2012, http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf.

[12] J. Rao and K. A. Ross, "Making b+-trees cache conscious in main memory," in *In Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, 2000.

[13] M. Daga, T. Scogland, and W. Feng, "Architecture-aware mapping and optimization on a 1600-core gpu," in *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, Dec. 2011.

[14] D. G. Merrill and A. S. Grimshaw, "Revisiting sorting for gpgpu stream architectures," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010.

[15] A. Aviram, "Original B+ Tree Source," 2010, http://www.amittai.com/prose/bplustree.html.

[16] AMD, "CodeAnalyst Performance Analyzer," 2012, http://developer.amd.com/tools/hc/CodeAnalyst/Pages/default.aspx.

[17] D. Comer, "Ubiquitous b-tree," *ACM Comput. Surv.*, vol. 11, no. 2, Jun. 1979.

[18] T. J. Lehman and M. J. Carey, "A study of index structures for main memory database management systems," in *Proceedings of the 12th International Conference on Very Large Data Bases*, ser. VLDB '86. San Francisco, California, USA: Morgan Kaufmann Publishers Inc., 1986.

[19] R. A. Hankins and J. M. Patel, "Effect of node size on the performance of cache-conscious b+-trees," in *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, ser. SIGMETRICS '03. New York, NY, USA: ACM, 2003.

[20] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin, "Fractal prefetching b+-trees: optimizing both cache and disk performance," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '02. New York, NY, USA: ACM, 2002.

[21] S. Chen, P. B. Gibbons, and T. C. Mowry, "Improving index performance through prefetching," in *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '01. New York, NY, USA: ACM, 2001.

[22] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey, "Fast: fast architecture sensitive tree search on modern cpus and gpus," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010.

[23] K. Kaczmarski, "Experimental b+-tree for gpu," in *ADBIS (2)'11*, 2011.

[24] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander, "Relational query coprocessing on graphics processors," *ACM Trans. Database Syst.*, vol. 34, no. 4, Dec. 2009.

[25] P. Bakkum and K. Skadron, "Accelerating sql database operations on a gpu with cuda," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU '10. New York, NY, USA: ACM, 2010.

[26] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey, "Palm: Parallel architecture-friendly latch-free modifications to b+ trees on many-core processors." *PVLDB*, vol. 4, no. 11, 2011.

[27] M. Heimel and V. Markl, "A first step towards gpu-assisted query optimization," in *Proceedings of the Third International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, 2012.

[28] J. Fix, Wilkes, A, 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 (A4MMC)*, 2011.