FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs

Changkyu Kim[†], Jatin Chhugani[†], Nadathur Satish[†], Eric Sedlar[⋆], Anthony D. Nguyen[†], Tim Kaldewey[⋆], Victor W. Lee[†], Scott A. Brandt[◇], and Pradeep Dubey[†]

changkyu.kim@intel.com

[†]Throughput Computing Lab, Intel Corporation

[⋆]Special Projects Group, Oracle Corporation

[◇]University of California, Santa Cruz

ABSTRACT

In-memory tree structured index search is a fundamental database operation. Modern processors provide tremendous computing power by integrating multiple cores, each with wide vector units. There has been much work to exploit modern processor architectures for database primitives like scan, sort, join and aggregation. However, unlike other primitives, tree search presents significant challenges due to irregular and unpredictable data accesses in tree traversal.

In this paper, we present FAST, an extremely fast architecture sensitive layout of the index tree. FAST is a binary tree logically organized to optimize for architecture features like page size, cache line size, and SIMD width of the underlying hardware. FAST eliminates impact of memory latency, and exploits thread-level and data-level parallelism on both CPUs and GPUs to achieve 50 million (CPU) and 85 million (GPU) queries per second, 5X (CPU) and 1.7X (GPU) faster than the best previously reported performance on the same architectures. FAST supports efficient bulk updates by rebuilding index trees in less than 0.1 seconds for datasets as large as 64M keys and naturally integrates compression techniques, overcoming the memory bandwidth bottleneck and achieving a 6X performance improvement over uncompressed index search for large keys on CPUs.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems

General Terms

Performance, Algorithms

1. INTRODUCTION

Tree structured index search is a critical database primitive, used in a wide range of applications. In today's data warehouse systems, many data processing tasks, such as scientific data mining, network monitoring, and financial analysis require handling large volumes of index search with low-latency and high-throughput.

As memory capacity has increased dramatically over the years, many database tables now reside completely in memory, thus eliminating disk I/O operations. Modern processors integrate multiple cores in a chip, each with wide vector (SIMD) units. Although memory bandwidth has also been increasing steadily, the bandwidth to compute ratio is reducing, and eventually memory bandwidth will become the bottleneck for future scalable performance.

In the database community, there is growing interest in exploiting the increased compute in modern processors. Recently, researchers have explored speeding up critical primitives like scan, sort, join and aggregation [30, 11, 22, 12]. However, unlike these primitives, index tree search presents significant challenges in utilizing the high compute and bandwidth resources. Database search typically involves long latency for the main memory access followed by small number of arithmetic operations, leading to ineffective utilization of large number of cores and wider SIMD. This main memory access latency is difficult to hide due to irregular and unpredictable data accesses during the tree traversal.

In this paper, we present **FAST** (Fast Architecture Sensitive Tree) search algorithm that exploits high compute in modern processors for index tree traversal. FAST is a binary tree, managed as a hierarchical tree whose elements are rearranged based on architecture features like page size, cache line size, and SIMD width of underlying hardware. We show how to eliminate the impact of latency with hierarchically blocked tree, software pipelining, and prefetches.

Having eliminated memory latency impact, we show how to extract parallelism to achieve high throughput search on two high performance commodity architectures – CPUs and GPUs. We report the **fastest** search performance on both platforms by utilizing many cores and wide SIMD units. Our CPU search performance on the Core i7 with 64M[1] 32-bit (key, rid) pairs is **5X** faster than the best reported number, achieving a throughput of **50M** search queries per second. We achieve peak throughput even within a stringent response time constraint of 1 microsecond. Our GPU search on the GTX 280 is around 1.7X faster than the best reported number, achieving **85M** search queries per second. Our high throughput search is naturally applicable to look-up intensive applications like On-Line Analytical Processing (OLAP), DSS and data mining. Also, we can re-build our index trees in less than 0.1 seconds for 64M keys on both CPU and GPU, enabling fast bulk updates.

We compare CPU search and GPU search and provide analytical models to analyze our optimized search algorithms for each platform and identify the compute and bandwidth requirements. When measured with various tree sizes from 64K elements to 64M elements, CPU search is 2X faster than GPU search for small trees where all elements can fit in the caches, but 1.7X slower than GPU search for large tree where memory bandwidth limits the perfor-

[1] 1M refers to 1 million.

SIGMOD'10, June 6–11, 2010, Indianapolis, Indiana, USA.

mance. We also evaluate FAST search algorithm on the Intel Many-Core Architecture Platform (MICA), a Larrabee [29] based silicon platform for many-core research and software development. By exploiting wider SIMD and large caches, search on the MICA platform results in 2.4X - 3X higher throughput than CPU search and 1.8X - 4.4X higher than GPU search.

Search on current GPUs is compute bound and not bandwidth bound. However, CPU search becomes close to memory bandwidth bound as the tree size grows. Compressing the index elements will reduce bandwidth consumption, thereby improving CPU search performance. We therefore propose **compression techniques** for our CPU search that seamlessly handles variable length string keys and integer keys. Our compression extends the commonly used prefix compression scheme to obtain order preserving partial keys with low overhead of false positives. We exploit SSE SIMD execution to speed up compression, compressing 512MB string keys in less than 0.05 seconds for key sizes up to 100B on the Core i7. We integrate our fast SIMD compression technique into the FAST search framework on CPUs and achieve up to **6X** further speed up for 100B keys compared to uncompressed index search, by easing the memory bandwidth bottleneck.

Finally, we examine four different search techniques – FAST, FAST with compression, buffered scheme, sort-based scheme – under the constraint of response time. For the response time of 0.001 to 25 ms, FAST on compressed keys provides the maximum throughput of around 60M queries per second while the sort-based scheme achieves higher throughput for the response time of greater than 30 ms. Our architecture sensitive tree search with efficient compression support lends itself well to exploiting the future trends of decreasing bandwidth-to-compute ratio and increasing compute resource with more cores and wider SIMD.

2. RELATED WORK

B+-trees [13] were designed to accelerate searches on disk-based database systems. As main memory sizes become large enough to store databases, T-trees [23] have been proposed as a replacement, specifically tuned for main memory index structure. While T-trees have less storage overhead than B+-trees, Rao et al. [25] showed that B+-trees actually have better cache behavior on modern processors because the data in a single cache line is utilized more efficiently and used in more comparisons. They proposed a CSS-tree, where each node has a size equal to the cache line size with no child pointers. Later, they applied cache consciousness to B+-trees and proposed a CSB+-tree [26] to support efficient updates. Graefe et al. [16] summarized techniques of improving cache performance on B-tree indexes.

Hankins et al. [18] explored the effect of node size on the performance of CSB+-trees and found that using node sizes larger than a cache line size (i.e., larger than 512 bytes) produces better search performance. While trees with nodes that are of the same size as a cache line have the minimum number of cache misses, they found that TLB misses are much higher than on trees with large node sizes, thus favoring large node sizes. Chen at al. [9] also concluded that having a B+-tree node size larger than a cache line performs better and proposed pB+-trees, which tries to minimize the increase of cache misses of larger nodes by inserting software prefetches. Later, they extended pB+-trees (called "fpB+-trees") to optimize for both disk I/O and CPU caches for disk-based DBMS [10].

While using prefetches in pB+-trees reduces cache misses in the search within a node, all published tree structured indexes suffer from a full cache miss latency when moving from each node to its child. Chen at al. argue that prefetching the children is not efficient because the tree node fanout is large and each child will be visited with the same probability. Instead of prefetching all (or a subset of) children speculatively, our scheme waits until we know the correct child to move to, and only prefetches the correct child. To increase the prefetch distance (i.e., the number of cycles between a prefetch initiation and the use of the prefetched data), we use software pipelining [3]. Note that we do not waste main memory bandwidth due to the issue of unnecessary prefetches speculatively - this is important because memory bandwidth is a critical resource.

Another way of reducing TLB/cache miss overheads is to amortize the penalties by processing data in batches [4, 32]. Buffers are attached to nodes or sub-trees and queries are processed in batch. These batched search schemes essentially sacrifice response time to achieve high throughput and can only be used for applications where a large number of query requests arrive in a short amount of time, such as stream processing or indexed nested loop join.

Modern processors exploit SIMD execution to increase compute density. Researchers have used SIMD instructions to improve search performance in tree structured index [28, 31]. Zhou et al. [31] employ SIMD instructions to improve binary search performance. Instead of comparing a search key with one tree element, SIMD instructions allow K(=SIMD width) consecutive elements to be compared simultaneously. However, their SIMD comparison in a node only splits the search into two ranges just as in binary search. Overall, the total computation complexity is still $\log(N/K)$. Recently, P-ary and K-ary search have been proposed to exploit SIMD execution on GPUs [21] and CPUs [28]. Using the GPU's memory gather capability, P-ary accelerates search on sorted lists. The SIMD comparison splits the tree into $(K+1)$ ranges, thus reducing the total number of comparisons by a larger number of $\log_K(N)$. To avoid non-contiguous memory accesses at each level, they linearize the K-ary tree in increasing order of levels by rearranging elements. However, when the tree is large, traversing the bottom part of the tree incurs TLB/cache misses at each level and search becomes latency bound. We explore latency hiding techniques for CPUs and GPUs to improve instruction throughput, resulting in better SIMD utilization.

Another architectural trend affecting search is that main memory bandwidth is becoming a critical resource that can limit performance scaling on future processors. Traditionally, the problem has been disk I/O bandwidth. Compression techniques have been used to overcome disk I/O bottleneck by increasing the effective memory capacity [15, 17, 20]. The transfer unit between memory and processor cores is a cache line. Compression allows each cache line to pack more data and increases the effective memory bandwidth. This increased memory bandwidth can improve query processing speed as long as decompression overhead is kept minimal [19, 33]. Recently, SIMD execution has been applied to further reduce decompression overhead in the context of scan operations on compressed data [30].

While there is much research on handling numerical data in index trees [9, 18, 21, 25, 26, 28], there are relatively few studies on handling variable length keys [5, 7, 8]. Compression techniques can be used to shrink longer keys into smaller keys. For tree structured indexes, compressing keys increases the fanout of the tree and decreases the tree height, thus improving search performance by reducing cache misses. Binnig et al. [7] apply the idea of buffered search [32] to handle variable size keys in a cache-friendly manner when the response time is of less concern as compared to throughput and lookups can be handled in bulk.

3. ARCHITECTURE OPTIMIZATIONS

Efficient utilization of compute resources depends on how to extract instruction-level parallelism (ILP), thread-level parallelism

(TLP), and data-level parallelism (DLP) while managing usage of external memory bandwidth judiciously.

3.1 ILP

Most CPUs and GPUs are capable of issuing one or more instructions per cycle. The latency of memory instructions can prevent execution of other instructions. Every memory access must go through a virtual-to-physical address translation, which is in the critical path of program execution. To improve translation speed, a translation look aside buffer (TLB) is used to cache translation of most frequently accessed pages. If the translation is not found in the TLB, processor pipeline stalls until the TLB miss is served. Both last-level cache (LLC) and TLB misses are difficult to hide because the miss penalty reaches more than a few hundred cycles. If the miss penalty cannot be hidden, a processor cannot fully utilize its compute resources and applications become memory latency bound. At this point, exploiting SIMD vector units is ineffective.

One way to reduce the memory access latency is prefetches. However, hardware prefetcher is not effective for irregular memory accesses like tree traversal. Software prefetch instructions are also hard to insert for tree structured indexes. Prefetching tree nodes close to the current node results in very short prefetch distance and most prefetches will be too late to be effective. Tree nodes far down from the current node can create a large fan out and prefetching all tree elements down wastes memory bandwidth significantly since only one of prefetches will be useful.

3.2 TLP/DLP

Once memory latency impact is minimized, we can exploit high density compute resources in modern processors, which have integrated many cores, each with wider vector (SIMD) units. Parallelization and vectorization are two key techniques to exploit compute resources. Parallelization in search can be done trivially by assigning threads to different queries. Each core executes different search queries independently.

As for exploiting SIMD, there are multiple approaches to performing search. We can use a SIMD unit to speed up a single query, assign different queries to each SIMD lane and execute multiple queries in parallel, or combine these two approaches by assigning a query to a subset of SIMD lanes. The best approach depends on the underlying hardware architectures such as the SIMD width and efficiency of gathering and scattering[2] data. In Section 5, we describe the best SIMD search mechanism for both CPUs and GPUs.

3.3 Memory Bandwidth

With the increased compute capability, the demand for data also increases proportionally. However, main memory bandwidth is growing at a lower rate than compute [27]. Therefore, performance will not scale up to the number of cores and SIMD width if applications become bandwidth bound. To bridge the enormous gap between bandwidth requirements and what the memory system can provide, most processors are equipped with several levels of on-chip memory storage (e.g., caches on CPUs and shared buffer on GPUs). If data structures being accessed fit in this storage, no bandwidth is utilized, thus amplifying the effective memory bandwidth.

However, if data structures are too big to fit in caches, we should ensure that a cache line brought from the memory be fully utilized before evicted out of caches (called "cache line blocking"). A cache line with a typical size of 64 bytes can pack multiple data elements in it (e.g., 16 elements of 4 byte integers). The cache line blocking technique basically rearranges data elements so that the subsequent elements to be used also reside within the same cache line.

Finally, data compression techniques can be used to pack more data elements into a cache line and prevent performance to be bandwidth bound. In Section 6, we show integrating data compression into our index tree framework to improve performance besides gaining more effective memory space.

[2]In this paper, we use the term "gather/scatter" to represent read/write from/to non-contiguous memory locations

4. ARCHITECTURE SENSITIVE TREE

We motivate and describe our index tree layout scheme. We also provide analytical models highlighting the compute and memory bandwidth requirements for traversing the resultant trees.

4.1 Motivation

Given a list of (key, rid) tuples sorted by the keys, a typical query involves searching for tuples containing a specific key ($\texttt{key}_q$) or a range of keys ([$\texttt{key}_{q1}$, $\texttt{key}_{q2}$]). Tree index structures are built using the underlying keys to facilitate fast search operations – with run-time proportional to the depth of the trees. Typically, these trees are laid out in a breadth first fashion, starting from the root of the tree. The search algorithm involves comparing the search key to the key stored at a specific node at every level of the tree, and traversing a child node based on the comparison results. Only one node at each level is actually accessed, resulting in ineffective cache line utilization, to the linear storage of the tree. Furthermore, as we traverse deeper into the tree, each access results in accessing elements stored in different pages of memory, thereby incurring TLB misses. Since the result of the comparison is required before loading the appropriate child node, cache line prefetches cannot be issued beforehand. On modern processors, a search operation typically involves a long-latency TLB/cache miss followed by small number of arithmetic operations at **each level of the tree**, leading to ineffective utilization of the processor resources.

Although blocking for disk/memory page size has been proposed in the past [13], the resultant trees may reduce the TLB miss latency, but do not necessarily optimize for effective cache line utilization, leading to higher bandwidth requirements. Cache line wide nodes [25] minimize the number of accessed cache lines, but cannot utilize SIMD instructions effectively. Recently, 3-ary trees [28] were proposed to exploit the 4-element wide SIMD of CPUs. They rearranged the tree nodes in order to avoid expensive gather/scatter operations. However, their tree structure does not naturally incorporate cache line/page blocking and their performance suffers for tree sizes larger than the last-level cache (LLC). In order to efficiently use the compute performance of processors, it is imperative to eliminate the latency stalls, and store/access trees in a SIMD friendly fashion to further speedup the run-time.

4.2 Hierarchical Blocking

We advocate building binary trees (using the keys of the tuple) as the index structure, with a layout optimized for the specific architectural features. For tree sizes larger than the LLC, the performance is dictated by the number of cache lines loaded from the memory, and the hardware features available to hide the latency due to potential TLB and LLC misses. In addition, the first few levels of the tree may be cache resident, in which case the search algorithm should be able to exploit the SIMD architecture to speed up the run-time. In order to optimize for all the architectural features, we rearrange the nodes of the binary index structure and **blocking** in a **hierarchical** fashion. Before explaining our hierarchical blocking scheme in detail, we first define the following notation.

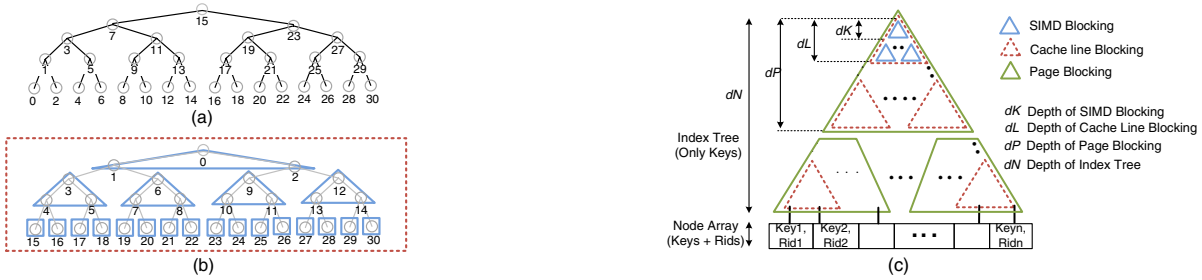

Figure 1: (a) Node indices (=memory locations) of the binary tree (b) Rearranged nodes with SIMD blocking (c) Index tree blocked in three-level hierarchy – first-level page blocking, second-level cache line blocking, third-level SIMD blocking.

E : Key size (in bytes).
K : SIMD width (in bytes).
L : Cache line size (in bytes).
C : Last level cache size (in bytes).
P : Memory page size (in bytes).
N : Total Number of input keys.
N_K : Number of keys that can fit into a SIMD register.
N_L : Number of keys that can fit into a cache line.
N_P : Number of keys that can fit into a memory page.
d_K : Tree depth of SIMD blocking.
d_L : Tree depth of cache line blocking.
d_P : Tree depth of page blocking.
d_N : Tree depth of Index Tree.

In order to simplify the computation, the parameters N_P, N_L and N_K are set to be equal to the number of nodes in complete binary sub-trees of appropriate depths[3]. For example, N_P is assigned to be equal to 2^{d_P}-1, such that $E(2^{d_P}\text{-}1) \leq P$ and $E(2^{d_P+1}\text{-}1) > P$. Similarly, $N_L = 2^{d_L}$-1 and $N_K = 2^{d_K}$-1. Consider Figure 1 where we let $N_L = 31$, $d_L = 5$ and $d_K = 2$. Figure 1(a) shows the indices of the nodes of the binary tree, with the root being the key corresponding to the 15^{th} tuple, and its two children being the keys corresponding to the 7^{th} and 23^{rd} tuples respectively, and so on for the remaining tree. Traditionally, the tree is laid out in a breadth-first fashion in memory, starting from the root node.

For our hierarchical blocking, we start with the root of the binary tree and consider the sub-tree with N_P elements. The first N_K elements are laid out in a breadth-first fashion. Thus, in Figure 1(b), the first three elements are laid out, starting from position 0. Each of the $(N_K + 1)$ children sub-trees (of depth d_K) are further laid out in the same fashion, one after another. This corresponds to the sub-trees (of depth 2) at positions 3, 6, 9 and 12 in Figure 1(b). This process is carried out for all sub-trees that are completely within the first d_L levels from the root. In case a sub-tree being considered does not completely lie within the first d_L levels (i.e. when d_L % d_K != 0), the appropriate number of levels (d_L % d_K) are chosen, and the elements laid out as described above. In Figure 1(b), since the 16 sub-trees at depth 4 can only accommodate depth one sub-trees within the first five (d_L) levels, we lay them out contiguously in memory, from positions 15 to 30.

After having considered the first d_L levels, each of the $(N_L + 1)$ children sub-trees are laid out as described above. This is represented with the red colored sub-trees in Figure 1(c). This process is carried out until the first d_P levels of the tree are rearranged and laid out in memory (the top green triangle). We continue the same rearrangement with the sub-trees at the next level and terminate when all the nodes in the tree have been rearranged to the appropriate positions. For e.g., Fig. 1(c) shows the rearranged binary tree, with the nodes corresponding to the keys stored in the sorted list of (key, rid) tuples. Our framework for architecture optimized tree layout preserves the structure of the binary tree, but lays it out in a fashion optimized for efficient searches, as explained in the next section.

[3] By definition, tree with *one* node has a depth of *one*.

4.3 Compute and Bandwidth Analysis

We first analyze the memory access pattern with our hierarchically blocked index tree structure, and then discuss the instruction overhead required for traversing the restructured tree. Let d_N denote the depth of the index tree. Consider Figure 1(c). Assuming a cold cache and TLB, the comparison to the root leads to a memory page access and a TLB miss, and say a latency of l_P cycles. The appropriate cache line is fetched from the main memory into the cache, incurring a further latency of say l_L cycles. We then access the necessary elements for the next d_L levels (elements within the top red triangle in the figure). The subsequent access incurs a cache miss, and a latency of l_L cycles. At an average, $\lceil d_P/d_L \rceil$ cache lines will be accessed within the first page (the top green triangle). Therefore, the total incurred latency for any memory page would be $(l_P + \lceil d_P/d_L \rceil l_L)$ cycles. Going to the bottom of the complete index tree would require $\lceil d_N/d_P \rceil$ page accesses, for an average incurred latency of $\lceil d_N/d_P \rceil (l_P + \lceil d_P/d_L \rceil l_L)$ cycles [4].

To take into account the caching and TLB effect, say d_C out of the d_N levels fit in the last level cache. Modern processors have a reasonable size TLB, but with a random query distribution, it is reasonable to assume just the entry for the top page to be in the page table during the execution of a random search. Therefore, the average incurred latency will be $(1\text{-}d_C/d_N)(\lceil d_N/d_P \rceil \lceil d_P/d_L \rceil l_L) + l_P(\lceil d_N/d_P \rceil\text{-}1)$ cycles (ignoring minimal latency of accessing cache lines from the cache). The resultant external memory bandwidth will be $L(1\text{-}d_C/d_N)(\lceil d_N/d_P \rceil \lceil d_P/d_L \rceil)$ bytes.

As for the computational overhead, our blocking structure involves computation of the starting address of the appropriate SIMD chunk, cache line, page block once we cross the appropriate boundary. For each crossed boundary, the computation is simply an accumulated scale-shift operation (multiply-add followed by add) due to the linear address translation scheme. For example, when crossing the cache line block, we need to multiply the relative child index from the cache line with the size of each cache line and add it to the starting address of that level of sub-trees.

For a given element key size (E), the number of accessed cache lines (and memory pages) is **provably minimized** by the hierarchical tree structure. However, while performing a single search per core, the compute units still do not perform any computation while waiting for the memory requests to return, thereby under utilizing the compute resource. In order to perform useful computation during the memory accesses, we advocate performing *multiple search queries* simultaneously on a single core/thread. We use software

[4] Assuming a depth of d_P for the bottom most page of the index tree. For a smaller depth, replace d_P/d_L with d'_P/d_L for the last page, where d'_P is the sub-tree depth for the last page.

pipelining, and interleave the memory access and computation for different queries. For example, while crossing cache line blocks, we issue a *prefetch* for the next cache line block to be accessed (for a specific query), and subsequently perform comparison operations for the other query(ies). After performing the comparison for all the remaining queries, we access the cache line (the same address we issued prefetch for earlier). With adequate amount of gap, the cache line would have been brought into the cache by the prefetch, thereby hiding memory latency. This process ensures complete utilization of the compute units. Although modern GPUs provide large number of threads to hide the memory access latency, the resultant memory access and instruction dependency still expose the latency, which is overcome using our layout and software pipelining schemes (Section 5.2).

In order to minimize the incurred latency during search operation, we want to increase d_P (in order to reduce the number of TLB misses) and increase d_L (to reduce the number of accessed cache lines). The only way to increase both is to reduce the element size (E). For in-memory databases, the number of tuples is typically less than the range of 32-bit numbers (2^{32}), and hence the keys can be represented using 32 bits (4-byte) integers. We assume 4-byte keys and 4-byte rid's for algorithm description in the next section, and provide algorithms for representing longer and variable length keys using 4-bytes in Section 6. 32-bit keys also map well to SIMD on modern architectures (CPU and GPU), with native instruction support for 32-bit elements in each SIMD lane.

5. CPU SEARCH VS. GPU SEARCH

We describe in detail the complete search algorithm on CPUs and GPUs We discuss various parameters used for our index tree layout and the SIMDified search code, along with a detailed analysis of performance and efficiency comparison on the two architectures.

5.1 CPU Implementation

Today's CPUs like the Intel Core i7 have multiple cores, each with a 128-bit SIMD (SSE) computational unit. Each SSE instruction can operate simultaneously on four 32-bit data elements. E equals four bytes for this section. As far as exploiting SIMD is concerned, there are multiple ways to perform searches:
(a) Searching *one* key, and using the SSE instructions to speedup the search.
(b) Searching *two* keys, using two SIMD lanes per search.
(c) Searching *four* keys, one per SIMD lane.

Both options (b) and (c) would require *gathering elements* from different locations. Since modern CPUs do not support an efficient implementation of gather, the overhead of implementing these instructions using the current set of instructions subsumes any benefit of using SIMD. Hence we choose option (a) for CPUs, and set d_K = 2 levels. The cache line size is 64 bytes, implying d_L = 4 levels. The page size used for our experimentation is 2MB (d_L = 19), although smaller pages (4KB, d_L = 10) are also available.

5.1.1 Building the Tree

Given a sorted input of tuples (T_i, i$\in$ (1..N), each having 4-byte (key, rid)), we layout the index tree ($T_\triangle$) by collecting the keys from the relevant tuples and laying them out next to each other. We set $d_N = \lceil \log_2(N) \rceil$. In case N is not a power of two, we still build the perfect binary tree, and assume keys for tuples at index greater than N to be equal to the largest key (or largest possible number), denoted as key$_L$. We iterate over nodes of the tree to be created (using index k initialized to 0). With current CPUs lacking gather support, we layout the tree by:
(a) computing the index (say j) of the next key to be loaded from the input tuples.
(b) loading in the key : key$' \leftarrow T_j$.key (if j>N , key$' \leftarrow$ key$_L$).
(c) $T_{\triangle_k}$ = key$'$, k++.

This process is repeated till the complete tree is constructed (i.e. k = (2^{d_N} -1)). The tree construction can be parallelized by dividing the output equally amongst the available cores, and each core computing/writing the relevant part of the output. We exploit SIMD for step (a) by computing the index for N_K (= 3) keys within the SIMD level block simultaneously. We use appropriate SSE instructions and achieve around 2X SIMD scaling as compared to the scalar code. Steps (b) and (c) are still performed using scalar instructions.

For input sizes that fit in the LLC, tree construction is compute bound, with around *20* ops[5] per element, for a total construction time of *20*·2^{d_N} ops per core. For N = 2M, the total time is around 40M cycles per core, assuming the CPU can execute 1 instruction per cycle. When the input is too large to fit into the LLC, the tree construction needs to read data from memory, with the initial loads reading complete cache lines but only extracting out the relevant 4 bytes. To compute the total bandwidth required, let's start with the leaf nodes of $T_\triangle$. There are a total of 2^{d_N-1} leaf nodes. The indices for the nodes would be the set of even indices – 0, 2, and so on. Each cache line holds eight (key, rid) tuples of which four tuples have even indices. Hence populating four of the leaf nodes of $T_\triangle$ requires reading one cache line, amounting to L/4 bytes per node. For the level above the leaf nodes, only two leaf nodes can be populated per cache line, leading to L/2 bytes per node. There are 2^{d_N-2} such nodes. For all the remaining nodes (2^{d_N-2}-1), a complete cache line per node is read. Since there is no reuse of the cache lines, the total amount of required bandwidth (analytically) would be $2^{d_N-1}L/4 + 2^{d_N-2}L/2 + (2^{d_N-2}-1)L \sim (L/2)2^{d_N}$ bytes, equal to *32*(2^{d_N} *)* bytes for CPUs. Depending on the available bandwidth, this may be compute/bandwidth bound. Assuming reasonable bandwidth (>1.6-bytes/cycle/core), our index tree construction is **compute bound**. For N **as large as 64M tuples**, the run-time is around 1.2 billion cycles (for a single core), which is **less than a 0.1 seconds** on the Core i7. With such fast build times, we can support updates to the tuples by buffering the updates and processing them in a batch followed by a rebuild of the index tree.

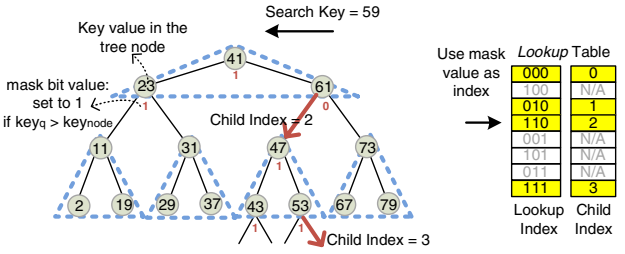

Figure 2: Example of SIMD(SSE) tree search and the lookup table.

5.1.2 Traversing the Tree

Given a search key (key$_q$), we now describe our SIMD friendly tree traversal algorithm. For a range query ([key$_{q1}$, key$_{q2}$]), key$_q$ $\leftarrow$ key$_{q1}$. We begin by splatting key$_q$ into a vector register (i.e., replicating key$_q$ for each SIMD lane), denoted by V$_{\text{keyq}}$. We start the search by loading 3 elements from the start of the tree into the register V$_{tree}$. At the start of a page, *page_offset* $\leftarrow$ 0.
Step 1: V$_{tree}$ $\leftarrow$ sse_load($T_\triangle$ + *page_offset*).
This is followed by the vector comparison of the two registers to set a mask register.

[5] 1 op implies 1 operation or 1 executed instruction.

```
/*
TΔ: starting address of a tree
page_address: starting address offset of a particular page blocking sub-tree
page_offset: starting address offset of a particular cache line blocking sub-tree
cache_offset: starting address offset of a particular SIMD blocking sub-tree
*/

__m128i xmm_key_q = _mm_load1_ps(key_q);
/* xmm_key_q : vector register Vkeyq, Splat a search key (keyq) in Vkeyq */

for (i=0; i<number_of_accessed_pages_within_tree; i++) {
    page_offset = 0;
    page_address = Compute_page_address(child_offset);
    for (j=0; j<number_of_accessed_cachelines_within_page; j++) {
        /* Handle the first SIMD blocking sub-tree (=2 levels of the tree)*/

        __m128i  xmm_tree = _mm_loadu_ps(TΔ + page_address + page_offset);
        /* xmm_tree: vector register Vtree. Load four tree nodes in Vtree*/

        __m128i xmm_mask = _mm_cmpgt_epi32(xmm_key_q, xmm_tree));
        /* xmm_mask: mask register Vmask. Set the mask register Vmask*/

        index = _mm_movemask_ps(_mm_castsi128_ps(xmm_mask));
        /* Convert mask register into index*/

        child_index = LookUp[index];

        /* Likewise, handle the second SIMD blocking sub-tree (=2 levels of the tree)*/
        xmm_tree = _mm_loadu_ps(TΔ + page_address + page_offset + NK*child_index);
        xmm_mask = _mm_cmpgt_epi32(xmm_key_q, xmm_tree));
        index = _mm_movemask_ps(_mm_castsi128_ps(xmm_mask));
        cache_offset = child_index*4 + Lookup[index];
        page_offset =  page_offset*16 + cache_offset;
    }
    child_offset = child_offset*(2^dp) + page_offset;
}

/* child_offset is the offset into the input (Key, Rid) tuple (T) */
While (T[child_offset].key <= keq_q2)
        child_offset++
```

Figure 3: SSE code snippet for index tree search.

Step 2: $V_{mask} \leftarrow$ sse_greater($V_{\texttt{keyq}}$, V_{tree}).
We then compute an integer value (termed *index*) from the mask register:
Step 3: *index* $\leftarrow$ sse_index_generation(V_{mask})
The *index* is then looked up into a *Lookup* table, that returns the local child index(*child_index*), and is used to compute the offset for the next set of load.
Step 4: *page_offset* $\leftarrow$ *page_offset* + N_K ·Lookup[*index*].
Since d_K = 2, there are two nodes on the last level of the SIMD block, that have a total of four child nodes, with local ids of 0, 1, 2 and 3. There are eight possible values of V_{mask}, which is used in deciding which of the four child nodes to traverse. Hence, the lookup table has 2^{N_K} (= 8) entries, with each entry returning a number $\in$ [0..3]. Even using four-bytes per entry, this lookup table occupies less than one cache line, and is always cache resident during the traversal algorithm.

In Figure 2, we depict an example of our SSE tree traversal algorithm. Consider the following scenario when $\texttt{key}_q$ equals 59 and ($\texttt{key}_q > V_{tree}[0]$), ($\texttt{key}_q > V_{tree}[1]$) and ($\texttt{key}_q < V_{tree}[2]$). In this case, the lookup table should return 2 (the left child of the second node on the second level, shown in the first red arrow in the figure). For this specific example, $V_{mask} \leftarrow$ [1, 1, 0] and hence *index* would be $1(2^0) + 1(2^1) + 0(2^2) = 3$. Hence *Lookup*[3] $\leftarrow$ 2. The other values in the lookup table are similarly filled up. Since the lookup table returns 2, *child_index* for the next SSE tree equals 2. Then, we compare three nodes in the next SSE tree and $V_{mask} \leftarrow$ [1, 1, 1], implying the right child of the node storing the value 53, as shown with the second red arrow in the figure.

We now continue with the load, compare, lookup and offset computation till the end of the tree is reached. After traversing through the index structure, we get the index of the tuple that has the *largest key* less than or equal to $\texttt{key}_q$. In case of range queries, we do a linear scan of the tuples, till we find the first key greater than $\texttt{key}_{q2}$.

Analysis: Figure 3 delineates our SSE tree traversal code. As

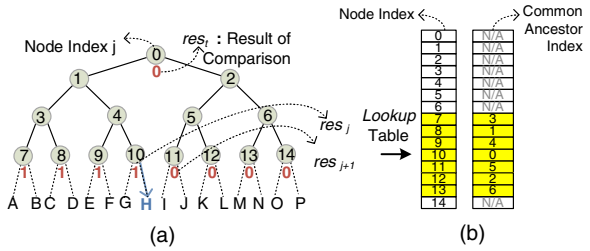

Figure 4: Example of GPU tree search and the lookup table.

compared to a scalar code, we resolve d_K (= $\log_2(N_K+1)$) levels simultaneously. Hence theoretically, a maximum of 2X (=d_K) speedup is possible (in terms of number of instructions). We first analyze the case where the index tree fits in the LLC. For each level of the index tree, the scalar code is:
child_offset $\leftarrow$ 2·child_offset + ($\texttt{key}_q > T_{\triangle}$[child_offset])

The above line of code performs 5 ops (load, compare, add, multiply and store). In comparison, the SSE code requires similar number of instructions for two levels. However, our blocking scheme introduces some overhead. For every 4 levels of execution, there are 2 ops (multiply-add for load address computation), another 2 ops (multiply-add for cache_offset computation), and 2 ops for multiply-add (for page_offset computation), for a total of 6 ops for 4 levels. Thus the net number of ops per level of the SSE code is around (((10+6)/4) = 4), for a speedup of 1.25X (=5/4). Since modern CPUs can execute multiple instructions simultaneously, the analytical speedup provides a high-level estimate of the expected speedup. As far as tree sizes larger than the LLC are concerned, for each cache line brought into memory, the total amount of instructions executed is around 16 ops. The net bandwidth required would be 64/16 = 4 bytes/cycle (assuming IPC=1). Even recent CPUs do not support such high bandwidths. Furthermore, the computation will be bandwidth bound for the last few levels of the tree, thereby making the actual SIMD benefit depend on the achieved bandwidth.

5.1.3 *Simultaneous Queries*

For tree sizes larger than the LLC, the latency of accessing a cache line (l_L) is the order of a few hundred cycles. The total amount of ops per cache line is around 16. To remove the dependency on latency, we execute S simultaneous queries, using the software pipelining technique. The value of S is set to be equal to **eight** for our experiments, since that covers up for the cache/TLB miss latency. In addition, for small tree sizes that fit in the LLC, our software pipelining scheme hides the latency caused by instruction dependency. The search code scales near linearly with multiple cores. The Core i7 supports the total of eight threads, with four cores and two SMT threads per each core. Therefore, we need a total of **64 concurrent queries** to achieve peak throughput.

5.2 GPU Implementation

The NVIDIA GPU architecture consists of multiple shared multiprocessors (or SMs). The GTX 280 has 30 such SMs. GPUs hide memory latency through multi-threading. Each GPU SM is capable of having more multiple threads of execution (up to 32 on the GTX 280) simultaneously active. Each such thread is called a thread *block* in CUDA [24].

Each GPU SM has multiple scalar processors that execute the same instruction in parallel. In this work, we view them as SIMD lanes. The GTX 280 has eight scalar processors per SM, and hence an 8-element wide SIMD. However, the logical SIMD width of the architecture is 32. Each GPU instruction works on 32 data ele-

ments (called a *thread warp*), which are executed in four cycles. GPUs provide hardware support for gather/scatter instructions at a half-warp granularity (16 lanes) [24], and hence we explored the complete spectrum of exploiting SIMD:
(a) Searching *32* keys, one per SIMD lane ($d_K = 1$).
(b) Searching *one* key, and exploiting the 32 element-wide SIMD ($d_K = 5$).
Since the GPUs do not explicitly expose caches, the cache line width (d_L) was set to be same as d_K. This reduces the overhead of computing the load address by the various SIMD lanes involved in searching a specific key. As far as the TLB size is concerned, NVIDIA reveals no information of page size and the existence of TLB in the official document. With various sizes of d_P, we did not see any change in run-time. Hence, d_P is assigned equal to d_N.

5.2.1 Building the Tree

Similar to the CPUs, we parallelize for the GPUs by dividing the output pages equally amongst the available SMs. On each SM, we run the scalar version of the tree creation algorithm on one of the threads within a half-warp (16 lanes). Only that one thread per half-warp executes the tree creation code, and computes the index, and updates the output page. This amounts to running two instances of tree creation per warp, with effective SIMD width of two. Running more than two instances within the same warp leads to *gather* (to read keys from multiple tuples in the SIMD operation), and *scatter* (to store the keys to different pages within the SIMD operation) from/to the global memory. In our experiments, we measured a slow down in run-time by enabling more than two threads per warp. We execute *eight* blocks on the same SM to hide the instruction/memory access latency. We assign one warp per block for a total of 30 SMs·8 (=240) warps.

As far as the run-time is concerned, the number of ops per tree element is similar to the CPU (∼20 ops), therefore reducing to 20/2 = 10 SIMD ops per element. Each of the executed warp takes 4 cycles of execution per warp. Hence, total number of cycles is equal to $10 \cdot 4 \cdot (2^{d_N})$ cycles (= $40 \cdot 2^{d_N}$ cycles) per SM. Since GPUs provide a very high memory bandwidth, our tree creation is **compute bound**. For N **as large as 64 million tuples**, the run-time is around 2.6 billion cycles, which is **less than 0.07 seconds** on GTX 280.

5.2.2 Traversing the Tree

d_K equal to 1 is a straightforward implementation, with each SIMD lane performing an independent search, and each memory access amounting to gather operations. Any value of d_K less than 4 leads to a gather operation within the half-warp, and the search implementation is latency bound. Hence we choose $d_K = 4$, since it avoids gather operations, and tree node elements are fetched using a load operation. Since GPUs have a logical SIMD width of 32, we issue *two independent* queries per warp, each with d_K = 4. Although the tree traversal code is similar to the CPUs, the current GPUs do not expose explicit masks and mask manipulation instructions. Hence Steps 2 and 3 (in Section 5.1.2) are modified to compute the local child index. We exploit the available *shared buffer* in the GTX 280 to facilitate inter-lane computation.

After comparing the key with the tree node element, each of the N_K SIMD lanes stores the result of the comparison (res_i = 0/1) in a pre-allocated space. Consider the *eight* leaf nodes in Figure 4(a) (assume $N_K = 15$). We need to find the largest index (say j), such that $res_j = 1$ and $res_{j+1} = 0$. For the figure, $j = 10$. Hence the child node is either `H` or `I`. The result depends on the comparison result of their common ancestor ($node_0$). In case $res_0 = 0$, key_q is *less than* the key stored at $node_0$, and hence the node belongs to its left sub-tree, thereby setting *index* ← `H` (shown in Figure 4). In case

```
/* In the GPU code, we process two independent queries within a warp*/
simd_lane = threadId.x %16; // 16 threads are devoted for each search query
query_id = threadId.x / 16;    // query_id, either 0 or 1
ancestor = Common_Ancester_Array [simd_lane];
base_index = 2*(simd_lane) – 13;
__shared__ int child_index [2]; // store the child index for two queries
__shared__ int shared_gt [32];

for (j=0; j<number_of_accessed_cachelines_within_page; j++) {
/* Handle the SIMD blocking sub-tree (=4 levels of the tree)*/
   page_address = (2^(4*j)-1) + page_offset*15;  // consume 2 ops

   int v_node = (Td + page_address + simd_lane)));  // consume 4 ops
   /* This is actually SIMD load. Our SIMD level blocking enables this instruction to be
   loading 16 consecutive values as opposed to loading 16 non-consecutive values */

   int gt = (keyq > v_node);                    // consume 2 ops
   shared_gt[threadIdx.x] = gt;                 // consume 2 ops
   __syncthreads();                             // consume 2 ops

   next_gt = shared_gt[threadIdx.x + 1];        // consume 2 ops
   if (threadIdx.x == 7) {                      // consume 2 ops
      child_index[query_id] = 0;                // consume 2 ops
   }
   if (threadIdx.x >=7) {                       // consume 2 ops
       if (gt & !next_gt) {                     // consume 2 ops
       /* res_j = 1 && res_j+1 = 0 */
          child_index[query_id] = base_index + shared_gt[ancestor];
                                                // consume 5 ops
       }
   }
   __syncthreads();                             // consume 2 ops
   page_offset = page_offset*16 + child_index[query_id]; // consume 3 ops
}
child_offset = page_offset;

/* child_offset is the offset into the input (Key, Rid) tuple (T) */
While (T[child_offset].key <= keq_q2)
        child_offset++
```

Figure 5: GPU code snippet for index tree search.

$res_0 = 1$, *index* ← `I`. We pre-compute the common ancestor node for each of the leaf nodes (with its successor node) into a lookup table, and load it into the *shared buffer* at the start of the algorithm. Figure 4(b) shows this lookup table for our specific example, and similar tables can be built for other values of d_K. The total size of the lookup table is N_K (=15 for our example).

Figure 5 shows the CUDA code for GPU search. __syncthreads() is required to ensure that the *shared buffer* is updated before being accessed in the subsequent line of code (for inter-lane communication). *child_index* is also stored in the *shared buffer* so that the relevant SIMD threads can access it for subsequent loads. This requires another __syncthreads() instruction. We also show the number of ops for each line of code in the figure. The total number of executed ops is 32. Since $d_K = 4$, the average number of executed ops per level is 8 ops for two queries within the 32-wide SIMD.

5.2.3 Simultaneous Queries

Although the GPU architecture is designed to hide latency, our implementation was still not completely compute bound. Therefore, we implemented our software pipelining technique by varying S from 1 to 2. This further reduced the latency, and our resultant run-time were within 5%–10% of the compute bound timings.

In order to exploit the GPU architecture, we execute independent queries on each SM. In order to hide the latency, we issue one warp per block, with eight blocks per SM, for a total of 240 blocks on the GTX 280 architecture. Since we execute 2 queries in SIMD, and S queries per warp, a total of $480S$ queries are required. With S being equal to 2, we operate the GPU at full throttle with **960 concurrent queries**.

5.3 Performance Evaluation

We now evaluate the performance of FAST on an quad-core Core i7 CPU and an GTX 280 GPU. Peak flops (computed as frequency · core count · SIMD width), peak bandwidth, and total frequency

Platform	Peak GFlops	Peak BW	Total Frequency
Core i7	103.0	30	12.8
GTX 280	933.3	141.7	39

Table 1: Peak compute (GFlops), bandwidth (GB/sec), and total frequency (Cores * GHz) on the Core i7 and the GTX 280.

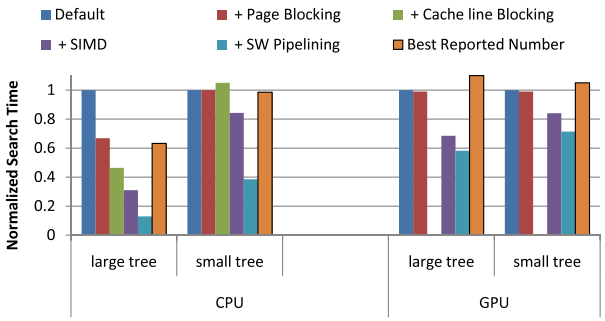

Figure 6: Normalized search time with various architectural optimization techniques (lower is faster). The fastest reported performance on CPUs [28] and GPUs [2] is also shown (for comparison).

(core count · frequency) of the two platforms are shown in Table 1. We generate 32-bit (key, rid) tuples, with both keys and rids generated randomly. The tuples are sorted based on the key value and we vary the number of tuples from 64K to 64M[6]. The search keys are also 32-bit wide, and generated uniformly at random. Random search keys exercise the worst case for index tree search with no coherence between tree traversals of subsequent queries.

We first show the impact of various architecture techniques on search performance for both CPUs and GPUs and compare search performance with the best reported number on each architecture. Then, we compare the throughput of CPU search and GPU search and analyze the performance bottlenecks for each architecture.

5.3.1 *Impact of Various Optimizations*

Figure 6 shows the normalized search time, measured in cycles per query on CPUs and GPUs by applying optimization techniques one by one. We first show the default search when no optimization technique is applied and a simple binary search is used. Then, we incrementally apply page blocking, cache line blocking, SIMD blocking, and software pipelining with prefetch. The label of "+SW Pipelining" shows the final relative search time when all optimization techniques are applied. We report our timings on the two extreme cases – small trees (with 64K keys) and large trees (with 64M keys). The relative performance for intermediate tree sizes fall in between the two analyzed cases, and are not reported.

For CPU search, the benefit of each architecture technique is more noticeable for large trees than small trees because large trees are more latency bound. First, we observe that search gets 33% faster with page blocking, which translates to around 1.5X speedup in throughput. Adding cache line blocking on top of page blocking results in an overall speedup of 2.2X. This reduction of search time comes from reducing the average TLB misses and LLC misses significantly – especially when traversing the lower levels of the tree. However, page blocking and cache line blocking do not help small trees because there are no TLB and cache misses in the first place; in fact, cache line blocking results in a slight increase of instructions with extra address computations. Once the impact of latency is reduced, SIMD blocking exploits data-level parallelism and provides an additional 20% – 30% gain for both small and large trees.

[6]64M is the max. number of tuples that fit in GTX 280 memory of 1GB.

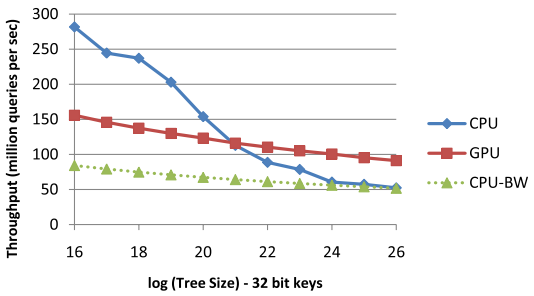

Figure 7: Comparison between the CPU search and the GPU search. "CPU-BW" shows the throughput projection when CPU search becomes memory bandwidth bound

Finally, the software pipelining technique with prefetch relaxes the impact of instruction dependency and further hides cache misses.

Our final search performance is **4.8X** faster for large trees and **2.5X** faster for small trees than the best reported numbers [28]. As shown in Figure 6, our **scalar** performance with page and cache line blocking *outperforms* the best reported **SIMD** search by around 1.6X. This emphasizes the fact that SIMD is only beneficial once the search algorithm is compute bound, and not bound by various other architectural latencies. Applications that are latency bound do not exploit the additional compute resources provided by SIMD instructions. Also note that our comparison numbers are based on a single-thread execution (for fair comparison with the best reported CPU number). When we execute independent search queries on multiple cores, we achieve *near-linear* speedup (3.9X on 4-cores). The default GPU search (Fig. 6) executes one independent binary search per SIMD lane, for a total of 32 searches for SIMD execution. Unlike CPU search, GPU search is less sensitive to blocking for latency. We do not report the number for cache line blocking since the cache line size is not disclosed. While the default GPU search suffers from gathering 32 tree elements, SIMD blocking allows reading data from contiguous memory locations thus removing the overhead of gather. Since the overhead of gather is more significant for large trees, our GPU search obtains 1.7X performance improvement for large trees, and 1.4X improvement for small trees with SIMD blocking. Our GPU implementation is compute bound.

5.3.2 *CPU search VS. GPU search*

We compare the performance of search optimized for CPU and GPU architectures. Figure 7 shows the throughput of search with various tree sizes from 64K keys to 64M keys. When the tree fits in the LLC, CPUs outperform GPUs by around 2X. This result matches well with analytically computed performance difference. As described in the previous subsections, our optimized search requires 4 ops per level per query for both CPUs and GPUs. Since GPUs take 4 cycles per op, they consume 4X more cycles per op as compared to the CPU. On the other hand, GPUs have 3X more total frequency than CPUs (Table 1). On small trees, CPUs are not bound by memory latency and can operate on the maximum instruction throughput rate. Unlike GPUs, CPUs can issue multiple instructions per cycle and we observe an IPC of around 1.5. Therefore, the total throughout ratio evaluates to around (1.5*4/3) ∼2X in the favor of CPUs.

As the tree size grows, CPUs suffer from TLB/LLC misses and get lower instruction throughput rate. The dotted line, labeled "CPU-BW" shows the throughput projection when CPU search becomes memory bandwidth bound. This projection shows that CPUs are compute bound on small trees and become closer to bandwidth bound on large trees. GPUs provide 4.6X higher memory bandwidth than CPUs and are far from bandwidth bound. In the next

	Throughput (million queries per sec)	
	Small Tree (64K keys)	Large Tree (16M keys)
CPU	280	60
GPU	150	100
MICA	667	183

Table 2: Measured performance comparison across three different platforms – CPU, GPU, and MICA.

section, we show how to further improve CPU search performance by easing bandwidth bottleneck with compression techniques.

Recent work [21] has shown that GPUs can improve search performance by an order of magnitude over CPUs and combining Fig. 6 and 7 confirms that unoptimized GPU search outperforms unoptimized CPU search by 8X for large trees. However, proper architecture optimizations reduced the gap and CPUs are only 1.7X slower on large trees, and in fact 2X faster on smaller trees. Note that GPUs provide much higher compute flops and bandwidth (Table 1). Thus optimized CPU search is *much better* than optimized GPU search in terms of **architecture efficiency**.

5.3.3 Search on MICA

We study how FAST would perform on the Intel Many-Core Architecture Platform (MICA), a Larrabee [29] based silicon platform for many-core research and software development.[7] Larrabee is an x86-based many-core processor architecture based on small in-order cores that uniquely combines full programmability of today's general-purpose CPU architecture with compute-throughput and memory bandwidth capabilities of modern GPU architectures. Each core is a general-purpose processor, which has a scalar unit based on the Pentium processor design, as well as a vector unit that supports 16 32-bit float or integer operations per clock. Larrabee has two levels of cache: low latency 32KB L1 data cache and larger globally coherent L2 cache that is partitioned among the cores. Each core has a 256KB partitioned L2 cache. To further hide latency, each core is augmented with 4-way multi-threading.

Since Larrabee features a 16-wide SIMD, we set $d_K = 4$ levels, enabling sub-tree traversal of four levels with SIMD operations. Unlike GPUs, Larrabee SIMD operation supports inter-lane computation. Therefore, SIMD tree traversal code is similar to SSE tree traversal code (Figure 3). The only difference is that $d_K = 4$ levels is used while SSE has $d_K = 2$ levels and no separate cache line blocking is necessary since we set d_L is assigned equal to d_K.

To compare search throughput with CPU/GPU search, we measure the performance on the MICA platform for small trees (with 64K keys) and large trees (with 16M keys). Table 2 shows search throughput on three different platforms – CPU, GPU, and MICA. As shown in the table, Larrabee takes the best of both architectures – a large cache in CPUs and high compute/bandwidth in GPUs. For small trees, Larrabee is able to store the entire tree within L2 cache, and is therefore compute bound. Compared to CPU and GPU, search on the MICA platform obtains a speed up of 2.4X and 4.4X respectively. The speedup numbers over CPUs are in line with the peak computational power of these devices, after accounting for the fact that search can only obtain a speedup of $\log(K)$ using K-wide SIMD. The high speedup over GPUs comes from the inefficiency of performing horizontal SIMD operations on GPUs. For large trees (16M entries – depth 24), the first 16 levels are cached in L2 and hence the first 16-level traversal is compute bound while the remaining 8-level traversal is bandwidth bound. We observe a speedup of 3X and 1.8X over CPUs and GPUs, respectively.

[7] All results reported in this section are based on Intel's internal research configurations of the Intel Many-Core Architecture Platform (MICA), which is designed by Intel for research and software development.

5.3.4 Limited Search Queries

Research on search generally assumes the availability of a large number of queries. This may be the case either when a large number of users concurrently submit searches or a database optimizer chooses multiple index probes for a join operation. However, in some cases, there are limited number of concurrent searches that are available to schedule. In addition, response time may become more important than throughput when search queries cannot be buffered beyond some threshold of response time. To operate on the maximum throughput, our CPU search requires 64 concurrent queries to be available while our GPU search requires 960 concurrent queries once the overheads of thread spawning and instruction cache miss have been amortized over a few thousand queries.

6. COMPRESSING KEYS

In the previous section, we presented run-times with key size (E) equal to 4 bytes. For some databases, the keys can be much larger in length, and also vary in length with the tuples. The larger the key size in the index tree, the smaller the number of levels per cache line and per memory page, leading to more cache lines and pages being read from main memory, and increased bandwidth/latency requirements per query. With the total number of tuples being less than the range of numbers represented by 4 bytes, it is possible (theoretically) to map the variable length keys to a fixed length of 4 bytes (for use in the index tree), and achieve maximum throughput.

In this section, we use compression techniques to ease memory bandwidth bottleneck and obtain further speedup for index tree search. Note that we focus on compressing the keys in the index tree[8] for CPUs. Since CPU search is memory bandwidth bound for last few levels of large trees, compressing the keys reduces the number of accessed cache lines, thereby reducing the latency /bandwidth, translating to improved performance. The GPU search is compute bound for 4-byte keys, and will continue to be compute bound for larger keys, with proportional decrease in throughput. Although the compression scheme developed in this section is applicable to GPUs too, we focus on CPUs and provide analysis and results for the same in the rest of the section.

6.1 Handling Variable Length Keys

We first present computationally simple *compression* scheme that maps variable input length data to an appropriate fixed length with small number of false positives, and incurs negligible construction and decompression overhead. Our scheme exploits SIMD for fast compression and supports order-preserving compression, leading to significant reduction in run-time for large keys.

6.1.1 Compression Algorithm

We compute a fixed length (E_p bits) partial key ($\texttt{pkey}_i$) for a given key ($\texttt{key}_i$). In order to support range queries, it is imperative to preserve the relative order of the keys, i.e. if ($\texttt{key}_j \leq \texttt{key}_k$), then ($\texttt{pkey}_j \leq \texttt{pkey}_k$). The cases where ($\texttt{pkey}_j = \texttt{pkey}_k$) but ($\texttt{key}_j < \texttt{key}_k$) are still admissible, but constitute the set of *false positives*, and should be minimal to reduce the overhead of redundant key comparisons during the search. Although there exist various hashing schemes to map the keys to a fixed length [6], the order preserving hash functions require $O(N \log_2 N)$ storage and compute time [14]. However, we need to compute and evaluate such functions in constant (and small) time.

We *extend* the prefix compression scheme [5] to obtain *order preserving* partial keys. These schemes compute a contiguous com-

[8] The original tuple data may be be independently compressed using other techniques [1] and used in conjunction with our scheme.

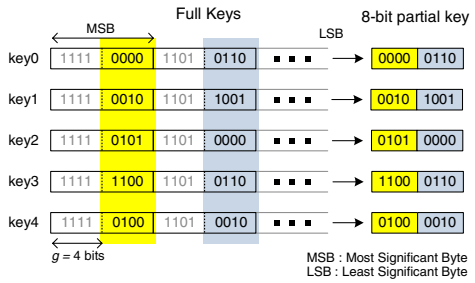

Figure 8: Example of extracting 8-bit partial keys in five keys.

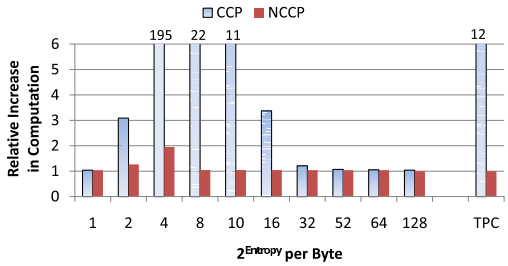

Figure 9: Relative increase in computation with respect to various alphabet sizes ($2^{entropy}$) per byte. For example, 52 means that we generate random keys for each byte among 52 values.

mon prefix of the set of keys, and store the subsequent part of the keys as the partial key. Instead, we compute a prefix such that the bits at which the keys differ constitute the partial key.

Let K represent the list of keys. We define a granularity factor (G), a number between 1 to 32. The partial key for all the keys is initialized to 0. We start by extracting and comparing the first G bits from all the keys. In case all the keys have the same G bits, we consider the next G bits, and repeat the process. In case they are different, we append these G bits to the partial key computed so far. We repeat this process and keep appending the set of G bits until a E_p bit partial key is obtained. Note that by construction, the prefix bits not appended to the partial key so far are the same for all the keys. Figure 8 shows an example for computing an 8-bit partial key for five keys with $G = 4$. Note that our scheme is a generalization of computing the common prefix – we extract a *non-contiguous common prefix*, with the discarded intermediate E_p bit positions forming the partial key. Our scheme is order preserving since we start with the most significant bit. It is indeed possible to get the same partial key for two different keys.

Setting the value of G: Databases storing variable length keys usually store low entropy values in each byte of the key. For example, keys representing the telephone numbers consist of only ten unique values in each byte. Similarly the name/address/url has the range of alphabets (52 values) in each byte. As far as setting G is concerned, it is affected by two competing factors: (a) the smaller the value of G, the higher the chances of forming partial keys with few false positives. (b) the larger the value of G, the lower the cost of computing the partial key from the original key. Furthermore, too small a G may require high cost of bit manipulation functions, thereby increasing the overhead of compressing/decompressing the keys. We choose $G = 4$ bits, which provide the right balance between the two factors described above.

Compression Cost: We exploit SSE instructions by loading 128-bits at a time for each key. We maintain a register V_{res} initialized to 0. We start by loading the first 128-bits of K_0 into register V_{K0}. For each key K_i, we load the relevant 128-bits (into V_{Ki}) and compute the bitwise exclusive-or operation, V_{xor} = _mm_xor_ps(V_{K0}, V_{Ki}). We then bitwise *or* the result with the V_{res} register, V_{res} = _mm_or_ps(V_{xor}, V_{res}). After iterating over all the keys, we analyze the V_{res} register by considering G bits at a time. If the G bits are all 0, then the result of all exclusive-or operations was 0, and hence all the keys have the relevant G bits the same. In case any of the bits is 1, then that relevant chunk becomes part of the partial key. We maintain a mask (termed as *pmask*) that stores a 0 or 1 for each G bit. We iterate over the remaining key (16-bytes at a time) to compute the bits contributing towards the partial key. As far as the total cost is concerned, we require 3 ops for every 16 bytes – for a total of $3\lceil E/16 \rceil$ ops (in the worst case). For 16-byte keys, this amounts to only 3 ops, and ∼21 ops per element for as long as 100-byte keys. All the keys now need to *packed* into their respective partial keys with the same *pmask*.

Consider the first 16-bytes of the key, loaded into the register V_{Ki}. This consists of 32 4-bit chunks, with a 32-bit *pmask*. $pmask_i$ = 1 implies the i^{th} chunk needs to be appended with the previously set 4-bit chunk. SSE provides for a general permute instruction (_mm_shuffle_eip8) that permutes the 16 1-byte values to any location within the register (using a control register as the second argument). However, no such instruction exists at 4-bit granularity. We however reduce our problem to permuting 8-bit data elements to achieve fast partial key computation. Each byte (say $B_{7..0}$) of data consists of two 4-bit elements ($B_{7..4}$ and $B_{3..0}$). Consider $B_{7..4}$. If chosen by the mask register, it can either end up as the 4 most significant bits in a certain byte of the output or in the 4 least significant bits of one of the bytes. Similar observation holds for $B_{3..0}$. Each of these 4-bit values within a byte can be 0 padded by 4 bits (at left or right) to create four 8-bit values: namely ($B_{3..0}$ 0000), (0000 $B_{3..0}$), ($B_{7..4}$ 0000) and (0000 $B_{7..4}$). We create four temporary registers for V_{Ki}, where each byte has been transformed as described. We can now permute each of the four registers (using the appropriate control registers), and bitwise *or* the result to obtain the desired result. Thus, the cost for *partial key* computation is 4 ops (for creating the temporary registers), 4 ops for permute, and 4 for *or*ing the result, for a total of $12\lceil E/16 \rceil$ ops per key. Including the cost to compute the *pmask*, the total cost for compression is $15\lceil E/16 \rceil$ ops. For a 512MB tree structure with 16-byte keys, our compression algorithm takes only 0.05 seconds on Core i7.

6.1.2 *Building the Compressed Tree*

We compute the partial keys for each page separately to increase the probability of forming effective partial keys with low false positives. As we traverse down the tree, it is important to have few (if any) false positives towards the top of the tree, since that increases the number of redundant searches expontentially. For example, say there are two leaf nodes (out of the first page) with the same partial key, and the actual search would have led through the second key. Since we only store the partial keys, the actual search leads through the first of these matches, and we traverse down that subtree and end up on a tuple that is $2^{(d_N - d_P)}$ tuples to the left of the actual match – leading to large slowdowns. Although Bohannon et al. [8] propose storing links to the actual keys, this leads to further TLB/LLC misses. Instead, we solve the problem by computing an E_p that leads to less than 0.01% false positives for the keys in the first page. We start with $E_p = 32$ bits, and compute the partial keys, and continue doubling E_p until our criterion for false positives is satisfied. In practice, with $E_p = 128$ bits (16-bytes), we saw no false postives for the first page of the tree (for $E \leq 100$).

For all subsequent pages, we use partial keys of length ($E_p = 32$ bits). The keys are compressed in the same fashion and each page stores the *pmask* that enables in fast extraction of the partial key. Since the time taken for building the index tree is around 20 ops per key (Section 5.1.1), the total time for building *compressed index tree* increases to $(20 + 15\lceil E/16 \rceil)$ ops per key. The compressed tree

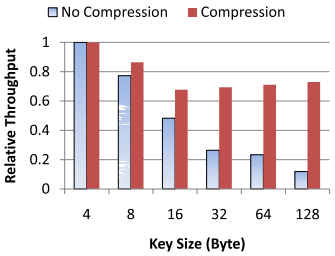

Figure 10: Throughput comparison between no compression and compression with key size from 4 to 128 bytes.

construction is still compute bound, and only around 75% slower than the uncompressed case (for $E \leq 16$).

6.1.3 Traversing the Compressed Tree

During the tree traversal, we do not decompress the stored partial keys for comparsion. Instead we compress $\texttt{key}_q$ (the query key) for each new page, and compare it with the partial keys, which enables search on compressed tuples. The cost of compressing the query key is around $12\lceil E/16 \rceil$ ops *per page*. The key length is a multiple of 16 bytes for the first page, and 4 bytes for all subsequent page accesses. For Steps 2 and 3 of the traversal algorithm (Section 5.1.2) on the first page, we need to compare in multiples of 16-byte keys and generate the appropriate index. We implement 16-byte key comparison in SSE using the native 8-byte comparison instruction (_mm_cmpgt_epi64), and use the corresponding mask-to-index generation instruction (_mm_movemask_pd) to generate index into a lookup table. The total cost of tree traversal for the first page increases to around 10 ops per level. For all subsequent pages, it is 4 ops per level (similar to analysis in Section 5.1).

As far as the bandwidth requirements are concerned, for tree sizes larger than the LLC, we now access one cache line for *four levels*. In comparison, for $E = 16$, we would have accessed one cache line for 2 levels, and hence the bandwidth requirement is reduced by $\sim$2X by storing the order-preserving partial keys. For $E \geq 32$, the bandwidth reduction is 4X and beyond. This translates to significant speedups in run-time over using long(er) keys.

6.1.4 Performance Evaluation

We implemented our compression algorithm on keys generated with varying entropy per byte, chosen from a set of 1 value (1-bit) to 128 values (7-bits), including cases like 10 (numeral keys) and 52 (range of alphabets). The key size varied from 4 to 128 bytes. The varying entropy per byte (especially low entropy) is the most challenging case for effective partial key computation. The distribution of the value within each byte does not affect the results, and we report data for values chosen uniformly at random (within the appropriate entropy). We also report results for 15-byte keys on phone number data collected from *CUSTOMER* table in TPC-H.

In Figure 9, we compare our compression scheme, non-contiguous common prefix (NCCP) with the previous scheme, contiguous common prefix (CCP) and report the relative increase in computaton for searching 16-byte keys with various entropies. The increase in computation is an indicative of the number of false positives, which increases the amount of redundant work done, and thereby increasing the run-time. A value of 1 imples *no false positives*. Even for very low entropy (choices per key ≤ 4), we perform relatively low extra computation, with total work less than 1.9X as compared to no false positives. For all other entropies, we measure less than 5% excess work, signifying the effectiveness of our low cost partial key computation. A similar overhead of around 5.2% is obesrved for the TPC-H data using our scheme, with the competing scheme reporting around 12X increase.

Figure 10 shows the relative throughput with varying key sizes (and fixed entropy per byte: $\log_2(10)$ bits). The number of keys for each case is varied so that the total tree size of the uncompressed keys is $\sim$1GB. All numbers are normalized to the throughput achieved using 4-byte keys. Without our compression scheme, the througput reduces to around 50% for 16-byte keys, and as low as 30% and 18% for key sizes 64 and 128 bytes respectively. This is due to the reduced effectiveness of cache lines read from main memory, and therefore increase in the latency/bandwidth. In contrast, our compression scheme utlizes cache lines more effectively by choosing 4-byte partial keys. The throughput drops marginally to around 80% for 16-byte keys (drop is due to the increase in cost for comparing 16-byte keys in the first page (Section 6.1.3)) and varies between 70%-80% for key sizes varying between 16 and 128 bytes. The overall run-time speedup is around 1.5X for 16-byte keys, and increases to around 6X for 128-byte keys.

6.2 Compressing Integer Keys

In the previous section, we described our algorithm for compressing large variable length keys. Apart from the first page, we store 4-byte partial keys for all other pages. Although the partial keys within each page are sorted, there does not exist a lot of coherence between the 32-bit data elements – as they are computed from longer original keys. However, for cases where the original key length is small (integer keys), there is coherence in the data that can be exploited, and the keys further compressed. This leads to further reduction in run-time – since run-time for the last few levels in trees is dependent on the available memory bandwidth. We now describe a light-weight compression scheme for such integer keys, widely used as primary keys for OLAP database tables.

6.2.1 Building Compressed Trees

We adopt the commonly used fixed-length delta based compression scheme. For each page, we find the minimum (K_{min}) and maximum key (K_{max}), and compute δ ($= \lceil \log_2(K_{max}\text{-}K_{min}+1) \rceil$) – the number of bits required to store the difference (termed as K_{δ_i}) between the key value and K_{min}. For each compressed page, we also store K_{min} and δ (using 32-bits each). Since each page may use different number of bits, we need an external Index Table, that stores the offset address of each page using 4-bytes. We use SSE to speed up computation of K_{min}, δ and the Index Table. To compute the minimum value, we use the instruction (_mm_min_epi32), and compute a vector of minimum values after iterating over all the keys. The actual minimum value is computed by computing the minimum of the four final values in the SSE register. The overall minimum value computation takes around 2 ops for 4 elements. Maximum value is computed similarly, and then delta for each key is computed with a total of around 2 ops per key. Packing the δ least significant bits from each key is performed using scalar code, for a total of around 6 ops per key. The total tree construction time is *increased* by around 30% over uncompressed tree building.

6.2.2 Traversing Compressed Trees

We read K_{min} and δ at the start of the compressed page, and compute ($\texttt{key}_q$-K_{min}) for the query key, termed as $\texttt{key}_{q_\delta}$. We directly compare $\texttt{key}_{q_\delta}$ with the compressed key values to compute the child index (Step 2 of the tree traversal in Section 5.1.2). The comparison is done using a SSE friendly implementation [30]. The resultant computational time only increases by 2 ops per level. On the other hand, our compression scheme increases the number of levels in each cache line, and reduces the number of cache lines. For example, for pages with 4X compression or more ($\delta \leq 8$), we can now fit d_K (≥ 6) levels, as opposed to 4 levels without com-

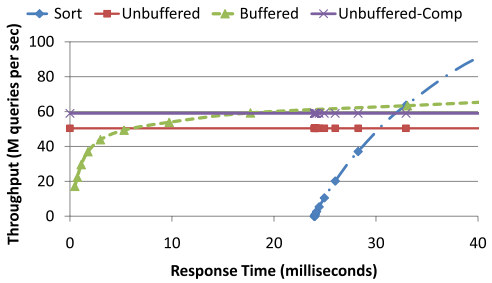

Figure 11: Throughput VS. Response time with various techniques.

pression. Since the performance for the uncompressed tree was bandwidth bound for the last few levels, this reduces the number of cache lines accessed by around 1.5X, which directly translates to run-time reduction. As far as the compression ratio is concerned, we achieve 2-4X compression on the randomly generated integer datasets, and the run-time reduces by 15%-20% as compared to the run-time without using our compression technique.

7. THROUGHPUT VS. RESPONSE TIME

In this section, we compare *four* different techniques for searching queries based on the user/application response time. This varies from stringent response time ($\leq$1 ms) to more tolerant ($\geq$ 50 ms).
(a) **Unbuffered Scheme:** FAST with *uncompressed index trees*.
(b) **Unbuffered-Compressed Scheme:** FAST with *compressed index trees*.
(c) **Buffered Scheme:** We implemented a variant of the buffered search algorithm [32]. We allocate a buffer for all the children nodes of the leaves of the sub-tree in the first page – 2^{d_p} buffers in total. Instead of searching a query through the complete tree, we traverse only the first page of the tree, and temporarily store the query in the appropriate buffer. As more queries are processed, each of these buffers start filling up. We trigger search for all the queries within a buffer *simultaneously* either when a pre-defined threshold of *batch of input queries* have been stored at these intermediate buffers, or when *any* of the buffers is filled up. This technique eliminates TLB misses that would have incurred with the unbuffered scheme (from one per query to one per group), and also exploit the caches better due to the coherence in the cache line accesses amongst the queries. This reduces the latency/bandwidth requirements, thereby speeding up the run-time.
(d) **Sort-Based Scheme:** Instead of traversing through the search tree, we sort the *input batch of queries*, and perform a linear pass of the input tuples and the sorted queries, comparing the individual queries and tuple keys and recording matches. The queries are sorted using an efficient implementation of merge-sort [11]. Although sort-based scheme performs a scan through all the input tuples, it provides better throughput when the *input batch* of queries is relatively large in number. For (c) and (d), the *response time* is defined as the time taken to process all queries in the input batch.

In Figure 11, we plot the throughput w.r.t. obtained response time. Both Unbuffered and Unbuffered-compressed scheme require only 64 simultaneous queries to achieve peak throughput, which corresponds to $\leq$1 ms of response time. The throughput for compressed index trees is around 20% greater than the traversal in the uncompressed trees (Section 6.2). The buffered scheme provides a peak throughput of around 5% larger than the compressed case, but requires a much larger *batch* ($\sim$640K) of input queries, that corresponds to response time of around 20-30 ms. Sort-based scheme also has a larger throughput, but exceeds our implementation for a response time of 33ms or larger with $\geq$ 2M queries.

8. CONCLUSIONS

We present FAST, an architecture sensitive layout of the index tree. We report fastest search performance on both CPUs and GPUs, with 5X and 1.7X faster than best reported numbers on the same platform. We also support efficient bulk updates by rebuilding index trees in less than 0.1 seconds for datasets as large as 64M keys. With future trend of limited memory bandwidth, FAST naturally integrates compression techniques with support for variable length keys and allows fast SIMD tree search on compressed index keys.

9. REFERENCES

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682, 2006.
[2] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, et al. Real-time parallel hashing on the GPU. *ACM Transactions on Graphics*, 28(5), Dec. 2009.
[3] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan. Software pipelining. *ACM Comput. Surv.*, 27(3):367–432, 1995.
[4] L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.
[5] R. Bayer and K. Unterauer. Prefix b-trees. *ACM Trans. Database Syst.*, 2(1):11–26, 1977.
[6] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Theory and practise of monotone minimal perfect hashing. In *ALENEX*, pages 132–144, 2009.
[7] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based order-preserving string compression for column stores. In *SIGMOD*, pages 283–296, 2009.
[8] P. Bohannon, P. Mcllroy, and R. Rastogi. Main-memory index structures with fixed-size partial keys. In *SIGMOD*, pages 163–174, 2001.
[9] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. *SIGMOD Record*, 30(2):235–246, 2001.
[10] S. Chen, P. B. Gibbons, T. C. Mowry, et al. Fractal prefetching b+-trees: optimizing both cache and disk performance. In *SIGMOD*, pages 157–168, '02.
[11] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, et al. Efficient implementation of sorting on multi-core SIMD CPU architecture. *PVLDB*, 1(2), 2008.
[12] J. Cieslewicz and K. A. Ross. Adaptive aggregation on chip multiprocessors. In *VLDB*, pages 339–350, 2007.
[13] D. Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.
[14] E. A. Fox, Q. F. Chen, A. M. Daoud, and L. S. Heath. Order-preserving minimal perfect hash functions. *ACM Trans. Inf. Syst.*, 9(3):281–308, 1991.
[15] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing relations and indexes. In *ICDE*, pages 370–379, 1998.
[16] G. Graefe and P.-A. Larson. B-tree indexes and cpu caches. In *ICDE*, pages 349–358, 2001.
[17] G. Graefe and L. Shapiro. Data compression and database performance. In *Applied Computing*, pages 22–27, Apr 1991.
[18] R. A. Hankins and J. M. Patel. Effect of node size on the performance of cache-conscious b+-trees. In *SIGMETRICS*, pages 283–294, 2003.
[19] A. L. Holloway, V. Raman, G. Swart, and D. J. DeWitt. How to barter bits for chronons: tradeoffs for database scans. In *SIGMOD*, pages 389–400, 2007.
[20] B. R. Iyer and D. Wilhite. Data compression support in databases. In *VLDB*, pages 695–704, 1994.
[21] T. Kaldewey, J. Hagen, A. D. Blas, and E. Sedlar. Parallel search on video cards. In *USENIX Workshop on Hot Topics in Parallelism*, 2009.
[22] C. Kim, E. Sedlar, J. Chhugani, T. Kaldewey, et al. Sort vs. hash revisited: Fast join implementation on multi-core CPUs. *PVLDB*, 2(2):1378–1389, 2009.
[23] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *VLDB*, pages 294–303, 1986.
[24] NVIDIA. *NVIDIA CUDA Programming Guide 2.3*. 2009.
[25] J. Rao and K. A. Ross. Cache conscious indexing for decision support in main memory. In *VLDB*, pages 78–89, 1999.
[26] J. Rao and K. A. Ross. Making b+- trees cache conscious in main memory. In *SIGMOD*, pages 475–486, 2000.
[27] M. Reilly. When multicore isn't enough: Trends and the future for multi-multicore systems. In *HPEC*, 2008.
[28] B. Schlegel, R. Gemulla, and W. Lehner. k-ary search on modern processors. In *DaMoN*, pages 52–60, 2009.
[29] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, et al. Larrabee: A Many-Core x86 Architecture for Visual Computing. *SIGGRAPH*, 27(3), 2008.
[30] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, et al. Simd-scan: Ultra fast in-memory scan using vector processing units. *PVLDB*, 2(1):385–394, 2009.
[31] J. Zhou and K. A. Ross. Implementing database operations using simd instructions. In *SIGMOD Conference*, pages 145–156, 2002.
[32] J. Zhou and K. A. Ross. Buffering accesses to memory resident index structures. In *VLDB*, pages 405–416, 2003.
[33] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *ICDE*, page 59, 2006.