Designing Fast Architecture-Sensitive Tree Search on Modern Multicore/Many-Core Processors

CHANGKYU KIM, JATIN CHHUGANI, and NADATHUR SATISH, Intel Corporation
ERIC SEDLAR, Oracle Corporation
ANTHONY D. NGUYEN, Intel Corporation
TIM KALDEWEY, Oracle Corporation
VICTOR W. LEE, Intel Corporation
SCOTT A. BRANDT, University of California, Santa Cruz
PRADEEP DUBEY, Intel Corporation

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 article, 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 Single Instruction Multiple Data (SIMD) width of the underlying hardware. FAST eliminates the 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 for large trees of 64M elements, with even better results on smaller trees. These are 5X (CPU) and 1.7X (GPU) faster than the best previously reported performance on the same architectures. We also evaluated FAST on the Intel® Many Integrated Core architecture (Intel® MIC), showing a speedup of 2.4X–3X over CPU and 1.8X–4.4X over GPU. 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, Measurement, Algorithms

Additional Key Words and Phrases: Tree search, compression, many-core, multicore, single instruction multiple data (SIMD), CPU, GPU

ACM Reference Format:
Kim, C., Chhugani, J., Satish, N., Sedlar, E., Nguyen, A. D., Kaldewey, T., Lee, V. W., Brandt, S. A., and Dubey, P. 2011. Designing fast architecture-sensitive tree search on modern multicore/many-core processors. ACM Trans. Datab. Syst. 36, 4, Article 22 (December 2011), 34 pages.
DOI = 10.1145/2043652.2043655 http://doi.acm.org/10.1145/2043652.2043655

Authors' addresses: C. Kim (corresponding author), J. Chhugani and N. Satish, Parallel Computing Lab, Intel Corporation; email: changkyu.kim@intel.com; E. Sedlar, Special Project Group, Oracle Corporation; A. D. Nguyen, Parallel Computing Lab, Intel Corporation; T. Kaldewey, Special Project Group, Oracle Corporation; V. W. Lee, Parallel Computing Lab, Intel Corporation; S. A. Brandt, University of Santa Cruz, Santa Cruz, CA; P. Dubey, Parallel Computing Lab, Intel Corporation.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2011 ACM 0362-5915/2011/12-ART22 $10.00
DOI 10.1145/2043652.2043655 http://doi.acm.org/10.1145/2043652.2043655

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. For instance, the latest CPU Intel® Core™ i7 processor is a quad-core processor with 128-bit SIMD (SSE), and NVIDIA® GeForce® GTX 280 GPU is a 30-core processor with 256-bit SIMD. Future processors will include a larger number of cores, with even wider SIMD units (e.g., 256-bit Advanced Vector Extensions (AVX) on CPUs, 512-bit on Intel® MIC [Seiler et al. 2008], and 512-bit on NVIDIA Fermi [Leischner et al. 2009]). Although memory bandwidth has also been increasing steadily, the bandwidth to computation ratio is declining, and eventually memory bandwidth will become the bottleneck for future scalable performance.

In the database community, there is growing interest in exploiting the increased computational power in modern processors. Recently, researchers have explored speeding up critical primitives like scan, sort, join, and aggregation [Willhalm et al. 2009; Chhugani et al. 2008; Kim et al. 2009; Cieslewicz and Ross 2007]. However, unlike these primitives, index tree search presents significant challenges in utilizing the high computing power and bandwidth resources. Database search typically involves long latency for main memory access followed by a small number of arithmetic operations, leading to ineffective utilization of a 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 article, we present the FAST (Fast Architecture-Sensitive Tree) search algorithm that exploits high computing power 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 a hierarchically blocked tree, software pipelining, and prefetches.

Having eliminated the 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 Intel Core i7 processor 975 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 OnLine Analytical Processing (OLAP), DSS, and data mining. However, our scheme is still restricted to static trees, and requires rebuild to reflect updated values. For such cases, we can rebuild our index trees in less than 0.1 seconds for 64M keys on both CPUs and GPUs, 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 computation and bandwidth requirements. When measured with various tree sizes from 64K elements

[1] 1M refers to 1 million.

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 trees where memory bandwidth limits the performance.

We also evaluate the FAST search algorithm on the new Intel MIC architecture. By exploiting wider SIMD and large caches, search on Intel's Knights Ferry implementation of the MIC architecture results in 2.4X–3X higher throughput than CPU search and 1.8X–4.4X higher than GPU search.

Research on search generally assumes the availability of a large number of search queries. However, in some cases, there is a limit to the number of concurrent searches that are available to schedule. GPUs are designed for high throughput, thus producing the peak throughput only when there are enough queries available. To maximize throughput, GPU search needs 15X more concurrent queries than CPUs. Therefore, CPU search gets close to its peak throughput quicker than GPU search.

Search on current GPUs is computation 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[2] on the Intel Core i7 processor. We integrate our fast SIMD compression technique into the FAST search framework on CPUs and achieve up to 6X further speedup 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 25ms, 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 30ms. Our architecture-sensitive tree search with efficient compression support lends itself well to exploiting the future trends of decreasing bandwidth to computation ratio and increasing computation resource with more cores and wider SIMD.

2. RELATED WORK

B+-trees [Comer 1979] were designed to accelerate searches on disk-based database systems. As main memory sizes become large enough to store databases, T-trees [Lehman and Carey 1986] have been proposed as a replacement, specifically tuned for main memory index structure. While T-trees have less storage overhead than B+-trees, Rao and Ross [1999] 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 [Rao and Ross 2000] to support efficient updates. Graefe and Larson [2001] summarized techniques of improving cache performance on B-tree indexes.

Hankins and Patel [2003] 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

[2] 1B refers to 1 billion.

that Translation Lookaside Buffer (TLB) misses are much higher than on trees with large node sizes, thus favoring large node sizes. Chen et al. [2001] 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 [Chen et al. 2002].

While using prefetches in pB+-trees reduces cache misses in the search within a node, all published tree structured indexes suffer a full cache miss latency when moving from each node to its child. Chen et 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 [Allan et al. 1995]. 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 and cache miss overheads is to amortize the penalties by processing data in batches [Arge 2003; Zhou and Ross 2003]. Buffers are attached to nodes or subtrees 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 computational density. Researchers have used SIMD instructions to improve search performance in tree structured index [Schlegel et al. 2009; Zhou and Ross 2002]. Zhou and Ross [2002] 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 [Kaldewey et al. 2009] and CPUs [Schlegel et al. 2009]. 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 noncontiguous 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 and 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.

This article is an extended version of the conference paper [Kim et al. 2010]. In this extended version, we present tree search on the new Intel MIC (Many Integrated Core) architecture. We include the algorithm, implementation, and analysis of FAST search on Intel's Knights Ferry processor (a silicon implementation of Intel MIC architecture with 32 cores at 1.2GHz). Intel MIC differs from GPUs since it provides a cache-coherent last-level cache to allow a large degree of tree blocking, thus filtering out accesses to external memory. MIC differs from CPUs by providing much higher computational power and bandwidth. Thus, as opposed to the previous paper [Kim et al. 2010], this article provides detailed optimization and analysis techniques of in-memory tree search on a spectrum of throughput architectures: CPUs, GPUs, and MICs. In Section 6.3, we extend FAST search to two latest high-throughput platforms, 6-core Intel Core i7 processor X980 (Intel microarchitecture code name Westmere) and NVIDIA®

Tesla™ C2050, respectively, and show how FAST search can exploit increased computational power and memory bandwidth of modern processors effectively. Finally, in Section 8, we provide a detailed description and new analysis for comparing unbuffered FAST on uncompressed and compressed trees, buffered FAST scheme, and sort-based search scheme with respect to response time(s) and achieved peak throughput.

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 [Goldstein et al. 1998; Graefe and Shapiro 1991; Iyer and Wilhite 1994]. The transfer unit between memory and processor cores is a cache line. Compression allows each cache line to contain 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 [Holloway et al. 2007; Zukowski et al. 2006]. Recently, SIMD execution has been applied to further reduce decompression overhead in the context of scan operations on compressed data [Willhalm et al. 2009].

While there is much research on handling numerical data in index trees [Rao and Ross 1999; 2000; Hankins and Patel 2003; Chen et al. 2001; Kaldewey et al. 2009; Schlegel et al. 2009], there are relatively few studies on handling variable-length keys [Bayer and Unterauer 1977; Graefe and Larson 2001; Bohannon et al. 2001; Binnig et al. 2009]. Compression techniques can be used to shrink longer keys into smaller (variable- [Bayer and Unterauer 1977] or fixed-length [Bohannon et al. 2001]) 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. [2009] apply the idea of buffered search [Zhou and Ross 2003] to handle variable-size keys in a cache-friendly manner when the response time is of less concern as compared to throughput and look-ups can be handled in bulk.

3. ARCHITECTURE OPTIMIZATIONS

Efficient utilization of computational 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. Instruction-Level Parallelism (ILP)

Most modern processors with superscalar architectures can execute multiple instructions simultaneously on different functional units, resulting in Instruction Level Parallelism (ILP). Moreover, modern processors with pipelining can issue a new instruction to the same functional unit per cycle. However, this requires that a sufficient number of independent instructions are available to the functional units.

An important factor that can limit the number of independent instructions is the impact of memory latency. Instructions that are waiting for the result of a memory operation are unavailable to schedule, resulting in lower ILP. Memory operations such as loads and stores can suffer high latency due to a number of reasons. First, the memory operation may miss the Last-Level Cache (LLC) on the processor. This means that the memory access must access the DRAM that is off-chip, resulting in a latency of hundreds of cycles. Second, 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 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 computational resources and applications become memory latency bound. For example, operations such as tree traversal inherently generate irregular and nonpredictable memory access patterns, especially when the tree size is too big to fit in TLBs or caches. The resulting long memory latency can render database operations like index search latency bound, thus causing low instruction throughput.

One way to reduce the memory access latency is prefetches. CPUs have a hardware prefetcher that can predict memory access patterns and fetch the data in advance. However, this hardware prefetcher is not effective for irregular memory accesses like tree traversal. An alternative is to issue explicit prefetch instructions in software. Software prefetch instructions are also often ineffective 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 fanout and prefetching all tree elements down wastes memory bandwidth significantly since only one of the prefetches will be useful. In later sections, we describe in detail how to make index search nonlatency bound. We also compare two different hardware architectures (CPUs and GPUs) and show how each architecture hides long memory latency to improve instruction throughput.

3.2. Thread-Level Parallelism (TLP) and Data-Level Parallelism (DLP)

Once the impact of memory latency is minimized, we can exploit the high-density computational resources available in modern processors. Modern processors have integrated an increasing number of cores. For example, current state-of-the-art Intel Xeon processor X5670 (Westmere) has 6 cores. In order to exploit multiple cores, the program is parallelized using threads and each thread is assigned to a core. Parallelization (i.e., exploiting Thread-Level Parallelism or TLP) in search can be done trivially by assigning threads to different queries. Each core executes different search queries independently.

The cores in modern processors have functional units that perform the same operation on multiple data items simultaneously. These units are called Single Instruction Multiple Data (SIMD) units. Today's processors have 128-bit wide SIMD instructions that can, for instance, perform operations on four 32-bit elements simultaneously. Such operations have also traditionally been called vector operations. In this article, we use the terms "exploiting Data-Level Parallelism (DLP)", "vectorization," and "exploiting SIMD" interchangeably. There are multiple approaches to performing tree search using SIMD units. 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[3] data. In Section 5, we describe the best SIMD search mechanism for both CPUs and GPUs.

3.3. Memory Bandwidth

With the increased computational capability, the demand for data also increases proportionally. However, main memory bandwidth is growing at a lower rate than computational power [Reilly 2008]. 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

[3]In this article, we use the term "gather/scatter" to represent read/write from/to noncontiguous memory locations.

CPUs and shared buffers 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 being 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 7, we show integrating data compression into our index tree framework to improve performance, besides gaining more effective memory space.

4. ARCHITECTURE SENSITIVE TREE

We motivate and describe our index tree layout scheme. We also provide analytical models highlighting the computational 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 runtime 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. 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 miss and cache miss followed by a 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 [Comer 1979], 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 [Rao and Ross 1999] minimize the number of accessed cache lines, but cannot utilize SIMD instructions effectively. Recently, 3-ary trees [Schlegel et al. 2009] 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 performance of processors, it is imperative to eliminate the latency stalls, and store/access trees in a SIMD friendly fashion to further speed up the runtime.

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 runtime. We also use SIMD for other levels that are not cache resident, with a resulting small benefit. This benefit is lower than that for the first few levels because the execution time on the lower levels of the tree is dominated by the memory bandwidth. 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.

$\mathcal{E}$: Key size (in bytes).
$\mathcal{K}$: SIMD width (in bytes).
$\mathcal{L}$: Cache line size (in bytes).
$\mathcal{C}$: Last-level cache size (in bytes).
$\mathcal{P}$: Memory page size (in bytes).
$\mathcal{N}$: Total number of input keys.
$\mathcal{N}_\mathcal{K}$: Number of keys that are assigned to an SIMD register.
$\mathcal{N}_\mathcal{L}$: Number of keys that are assigned to a cache line.
$\mathcal{N}_\mathcal{P}$: Number of keys that are assigned to a memory page.
$d_\mathcal{K}$: Tree depth of SIMD blocking.
$d_\mathcal{L}$: Tree depth of cache line blocking.
$d_\mathcal{P}$: Tree depth of page blocking.
$d_\mathcal{N}$: Tree depth of index tree.

In order to simplify the computation, the parameters $\mathcal{N}_\mathcal{P}$, $\mathcal{N}_\mathcal{L}$, and $\mathcal{N}_\mathcal{K}$ are set to be equal to the number of nodes in complete binary subtrees of appropriate depths.[4] For example, $\mathcal{N}_\mathcal{P}$ is assigned to be equal to $2^{d_\mathcal{P}} - 1$, such that $\mathcal{E}(2^{d_\mathcal{P}} - 1) \leq \mathcal{P}$ and $\mathcal{E}(2^{d_\mathcal{P}+1} - 1) > \mathcal{P}$. Similarly, $\mathcal{N}_\mathcal{L} = 2^{d_\mathcal{L}} - 1$ and $\mathcal{N}_\mathcal{K} = 2^{d_\mathcal{K}} - 1$. Consider Figure 1 where we let $\mathcal{N}_\mathcal{L} = 31$, $d_\mathcal{L} = 5$ and $d_\mathcal{K} = 2$. Figure 1(a) shows the indices of the nodes of the binary tree, with the root being the key corresponding to the 15th tuple, and its two children being the keys corresponding to the 7th and 23rd 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 subtree with $\mathcal{N}_\mathcal{P}$ elements. The first $\mathcal{N}_\mathcal{K}$ 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 $(\mathcal{N}_\mathcal{K} + 1)$ children subtrees (of depth $d_\mathcal{K}$) are further laid out in the same fashion, one after another. This corresponds to the subtrees (of depth 2) at positions 3, 6, 9, and 12 in Figure 1(b). This process is carried out for all subtrees that are completely within the first $d_\mathcal{L}$ levels from the root. In case a subtree being considered does not completely lie within the first $d_\mathcal{L}$ levels (i.e., when $d_\mathcal{L}$ % $d_\mathcal{K}$!= 0), the appropriate number of remaining levels ($d_\mathcal{L}$ % $d_\mathcal{K}$) are chosen, and the elements laid out as described earlier. In Figure 1(b), since the 16 subtrees at depth 4 can only accommodate depth one subtrees within the first five ($d_\mathcal{L}$) levels, we lay them out contiguously in memory, from positions 15 to 30.

After having considered the first $d_\mathcal{L}$ levels, each of the $(\mathcal{N}_\mathcal{L} + 1)$ children subtrees are laid out as described before. This is represented with the red colored subtrees in Figure 1(c). This process is carried out until the first $d_\mathcal{P}$ levels of the tree are rearranged and laid out in memory (the top green triangle). We continue the same rearrangement with the subtrees at the next level and terminate when all the nodes in the tree

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

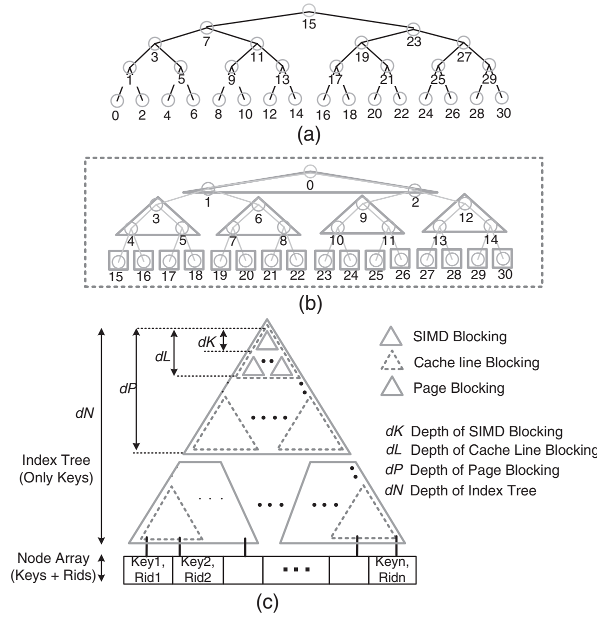

Fig. 1. (a) Node indices (=memory locations) of the binary tree; (b) rearranged nodes with SIMD blocking; (c) index tree blocked in three-level hierarchy: first-level page blocking, second-level cache line blocking, third-level SIMD blocking. Note that we only show the SIMD blocking for the top triangle for clarity, while it is actually present at all levels.

have been rearranged to the appropriate positions. For example, Figure 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.

4.3. Latency and Memory Traffic 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 $d_{\mathcal{N}}$ 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 $l_{\mathcal{P}}$ cycles. The appropriate cache line is fetched from the main memory into the cache, incurring a further latency of say $l_{\mathcal{L}}$ cycles. We then access the necessary elements for the next $d_{\mathcal{L}}$ levels (elements within the top red triangle in the figure). The subsequent access incurs a cache miss, and a latency of $l_{\mathcal{L}}$ cycles. At an average, $\lceil d_{\mathcal{P}}/d_{\mathcal{L}} \rceil$ cache lines will be accessed within the first page (the top green triangle). Therefore, the total incurred latency for any memory page would be $(l_{\mathcal{P}} + \lceil d_{\mathcal{P}}/d_{\mathcal{L}} \rceil l_{\mathcal{L}})$ cycles. Going to the bottom of the complete index tree would require $\lceil d_{\mathcal{N}}/d_{\mathcal{P}} \rceil$ page accesses, for an average incurred latency of $\lceil d_{\mathcal{N}}/d_{\mathcal{P}} \rceil (l_{\mathcal{P}} + \lceil d_{\mathcal{P}}/d_{\mathcal{L}} \rceil l_{\mathcal{L}})$ cycles. [5]

To take into account the caching and TLB effect, say $d_{\mathcal{C}}$ out of the $d_{\mathcal{N}}$ 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

[5]Assuming a depth of $d_{\mathcal{P}}$ for the bottommost page of the index tree. For a smaller depth, replace $d_{\mathcal{P}}/d_{\mathcal{L}}$ with $d'_{\mathcal{P}}/d_{\mathcal{L}}$ for the last page, where $d'_{\mathcal{P}}$ is the subtree depth for the last page.

page table during the execution of a random search. Therefore, the average incurred latency will be $(1 - d_{\mathcal{C}}/d_{\mathcal{N}})(\lceil d_{\mathcal{N}}/d_{\mathcal{P}}\rceil \lceil d_{\mathcal{P}}/d_{\mathcal{L}}\rceil l_{\mathcal{L}}) + l_{\mathcal{P}}(\lceil d_{\mathcal{N}}/d_{\mathcal{P}}\rceil - 1)$ cycles (ignoring minimal latency of accessing cache lines from the cache). The resultant external memory traffic will be $\mathcal{L}(1 - d_{\mathcal{C}}/d_{\mathcal{N}})(\lceil d_{\mathcal{N}}/d_{\mathcal{P}}\rceil \lceil d_{\mathcal{P}}/d_{\mathcal{L}}\rceil)$ 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 subtrees.

For a given element key size ($\mathcal{E}$), the number of accessed cache lines (and memory pages) is minimized by the hierarchical tree structure. However, while performing a single search per core, the computational units still do not perform any computation while waiting for the memory requests to return, thereby underutilizing the computational 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 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 computational units. Although modern GPUs provide a 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 $d_{\mathcal{P}}$ (in order to reduce the number of TLB misses) and increase $d_{\mathcal{L}}$ (to reduce the number of accessed cache lines). The only way to increase both is to reduce the element size ($\mathcal{E}$). 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 7. Section 7 accounts for the extra computation requirement for mapping longer keys to 4-byte keys. 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 processor have multiple cores, each with a 128-bit SIMD (SSE) computational unit. Four 32-bit execution units (called *SIMD lanes*) are laid down in a horizontal fashion to form the 128-bit SIMD unit. Each SSE instruction can operate simultaneously on four 32-bit data elements. $\mathcal{E}$ equals four bytes for this

section. As far as exploiting SIMD is concerned, there are multiple ways to perform searches:

(a) Searching *one* key, using the SSE instructions to speed up 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 $d_{\mathcal{K}} = 2$ levels. The cache line size is 64 bytes, implying $d_{\mathcal{L}} = 4$ levels. The page size used for our experimentation is 2MB ($d_{\mathcal{L}} = 19$)[6], although smaller pages (4KB, $d_{\mathcal{L}} = 10$) are also available.

5.1.1. Building the Tree. Given a sorted input of tuples ($\mathcal{T}_i$, i$\in$ (1..$\mathcal{N}$), each having 4 byte (key, rid)), we lay out the index tree ($\mathcal{T}_\triangle$) by collecting the keys from the relevant tuples and laying them out next to each other. We set $d_{\mathcal{N}} = \lceil \log_2(\mathcal{N}) \rceil$. In case $\mathcal{N}$ is not a power of two, we still build the perfect binary tree, and assume keys for tuples at index greater than $\mathcal{N}$ 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 lay out 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$' \leftarrow \mathcal{T}_j$.key (if $j > \mathcal{N}$, key$' \leftarrow$ key$_L$).
(c) $\mathcal{T}_{\triangle_k}$ = key$'$, k++. Store key$'$ in $\mathcal{T}_{\triangle_k}$, defined as the kth location in the rearranged index tree ($\mathcal{T}_\triangle$) and increment k.

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

Computation and Bandwidth Analysis. We now describe the computation and bandwidth analysis for tree building. Our general strategy is as follows. For computation analysis, we calculate the number of operations (or ops) required. We divide the ops by the expected Instructions Per Cycle (IPC) of the architecture to obtain an execution cycle bound based on computation. For bandwidth analysis, we first calculate the external memory traffic, and divide this by the effective memory bandwidth of the architecture to get the bandwidth bound execution cycles. These bounds give theoretical limits on performance. We measure the execution cycles of the actual implementation in order to determine if the program is close to computation or bandwidth bound limits.

For input sizes that fit in the LLC, tree construction is computation bound, with around *20* ops[7] per element, for a total construction time of $20 \cdot 2^{d_{\mathcal{N}}}$ ops per core. For $\mathcal{N}$ = 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 memory traffic

[6] 2MB page is the largest available page in our system.
[7] 1 op implies 1 operation or 1 executed instruction.

required, let's start with the leaf nodes of $\mathcal{T}_\triangle$. There are a total of $2^{d_\mathcal{N}-1}$ 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 $\mathcal{T}_\triangle$ requires reading one cache line, amounting to $\mathcal{L}/4$ bytes per node. For the level above the leaf nodes, only two leaf nodes can be populated per cache line, leading to $\mathcal{L}/2$ bytes per node. There are $2^{d_\mathcal{N}-2}$ such nodes. For all the remaining nodes ($2^{d_\mathcal{N}-2} - 1$), a complete cache line per node is read. Since there is no reuse of the cache lines, the total amount of required memory traffic (analytically) would be $2^{d_\mathcal{N}-1}\mathcal{L}/4 + 2^{d_\mathcal{N}-2}\mathcal{L}/2 + (2^{d_\mathcal{N}-2} - 1)\mathcal{L} \sim (\mathcal{L}/2)2^{d_\mathcal{N}}$ bytes, equal to *32*($2^{d_\mathcal{N}}$) bytes for CPUs. Depending on the available bandwidth, this may be computation or bandwidth bound. Assuming reasonable bandwidth (>1.6-bytes/cycle/core), our index tree construction is computation bound. For $\mathcal{N}$ as large as 64M tuples, the runtime is around 1.2 billion cycles (0.4 seconds on a single core of a 3.3 GHz Intel Core i7 processor), which is less than a 0.1 seconds on the 4-core Intel 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.

5.1.2. Traversing the Tree. Given a search key ($\texttt{key}_q$), we now describe our SIMD-friendly tree traversal algorithm. For a range query ([$\texttt{key}_{q1}$, $\texttt{key}_{q2}$]), $\texttt{key}_q \leftarrow \texttt{key}_{q1}$. We begin by splatting $\texttt{key}_q$ into a vector register (i.e., replicating $\texttt{key}_q$ for each SIMD lane), denoted by $V_{\texttt{keyq}}$. We start the search by loading 3 elements from the top 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((Base Address of $\mathcal{T}_\triangle$) + *page_offset*).
This is followed by the vector comparison of the two registers to set a mask register.
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* + $\mathcal{N}_\mathcal{K}\cdot$Lookup[*index*].

Since $d_\mathcal{K} = 2$, 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[8] has $2^{\mathcal{N}_\mathcal{K}}$ (= 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 until the end of the tree is reached. After traversing through the index structure, we get the

[8]The popcount instruction can also be used if available.

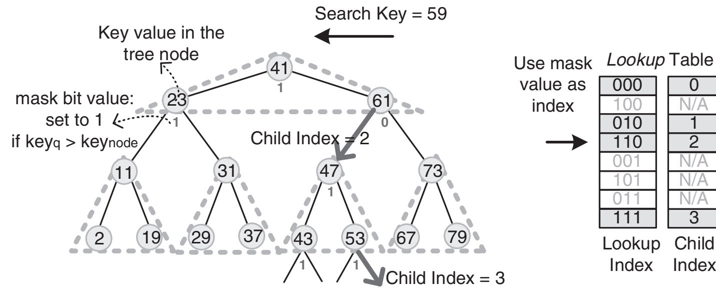

Fig. 2. Example of SIMD(SSE) tree search and the lookup table.

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, until we find the first key greater than $\texttt{key}_{q2}$.

Effect of Data Skew. As far as the output for the search queries is concerned, three scenarios exist: no matching keys are found, one matching key is found, or multiple matches are found (typical of range queries). If none of the input keys matches the search key, we still traverse the complete index tree to one of the leaf nodes where we find no matching tuples. In case of a successful match, we perform a linear scan of the tuples until the tuple with key larger than the search key is found. Hence, for all these scenarios, we pay a similar cost of tree traversal, while the case of multiple matches includes the overhead of linear scan of the input tuples to find all successful matches.

As for skewed data, it is possible to find lots of successful matches, in which case a significant amount of time will be spent in the linear scan of the leaf nodes. Note that in such cases, each scanned tuple would still contribute to the output, and hence no redundant computation is performed. In case of heavily skewed data, a large portion of the total time may be spent in such scan operations, in which cases optimization of the index tree has a smaller impact on the overall runtime.

Computation and Bandwidth Analysis. Figure 3 delineates our SSE tree traversal code. As compared to a scalar code, we resolve $d_{\mathcal{K}}$ $(= \log_2(\mathcal{N}_{\mathcal{K}} + 1))$ levels simultaneously. Hence theoretically, a maximum of 2X $(=d_{\mathcal{K}})$ 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:

$$\text{child_offset} \leftarrow 2 \cdot \text{child_offset} + (\texttt{key}_q > \mathcal{T}_{\triangle}[\text{child_offset}])$$

The preceding 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. Thus, total 10 SSE ops are required for four 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.

```
/*
TΔ: starting address of a tree
page_address: starting address of a particular page blocking sub-tree
page_offset: offset (# of tree nodes before the current node) within a particular page blocking
        sub-tree
level_offset: offset (# of tree nodes before the current node) at the current tree level
cache_offset: offset (# of tree nodes before the current node) within a particular cache line
        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];

        /* multiply by 16 because there are 16 leaf nodes in a cache line blocking sub-tree */
        level_offset =  level_offset*16 + cache_offset;
       /* 2^(4j) − 1 : # of tree nodes above the current cache line blocking sub-tree
        level_offset is multipled by 15 because each tree node has 15 tree nodes in a cache
          blocking sub-tree */
        page_offset = (2^(4j) − 1) + level_offset*15

    }
    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++
```

Fig. 3. SSE code snippet for index tree search.

5.1.3. Simultaneous Queries. For tree sizes larger than the LLC, the latency of accessing a cache line ($l_{\mathcal{L}}$) 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 $\mathcal{S}$ simultaneous queries per hardware thread, using the software pipelining technique.

Software pipelining is a technique used to optimize loops in a manner that parallels hardware pipelining. The query loop is unrolled, and the operations for the unrolled queries are interleaved. In tree search, the traversal is done independently for different queries. Each thread handles multiple queries organized as a loop. We call this loop the query loop. As an example, let $\{AB\}^n$ represent the query loop with operations A and B that is executed over n independent queries. The operations A and B for a single query are dependent on each other and cannot be simultaneously executed. However,

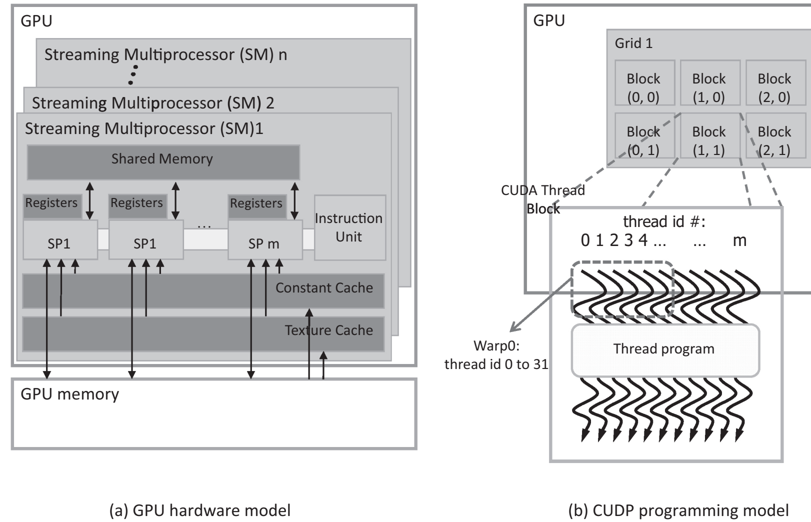

Fig. 4. Block diagram for (a) NVIDIA GPU hardware model and (b) GPU CUDA programming model. These are borrowed from the NVIDIA CUDA Programming Guide [NVIDIA 2010].

we can rewrite this loop as $A\{BA\}^{n-1}B$. In this new loop, the operations B and A correspond to different queries (A corresponds to the next loop iteration than B) and can be executed simultaneously. This example is simplified and the loop is not unrolled; unrolling the loop can expose even more instruction-level parallelism.

The value of $\mathcal{S}$ is set to be equal to eight for our experiments, since that covers up for the cache miss and 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 a parallel array of Streaming Multiprocessors (or SMs). Figure 4(a) shows the GPU hardware model with multiple SMs. Each GPU SM has multiple Scalar Processors (SPs) that execute the same instruction in parallel. In this work, we view SPs as equivalent to SIMD lanes. The GTX 280 has eight scalar processors per SM, and hence an 8-element wide SIMD (the later NVIDIA GeForce GTX 480 (GTX 480) has a 16-element wide SIMD). All the SPs within a single SM communicate via an on-chip memory called shared memory.

Each GPU SM is capable of supporting multiple simultaneous threads of execution in order to hide memory latencies. Only one (two on the latest GTX 400 series) thread of execution can be executing on one SM at any one point of time; however if any thread stalls due to memory latency or other events, another active thread can start executing. The process of switching in and out threads of execution is hardware assisted and is inexpensive. The NVIDIA GTX 280 has 30 SMs, and each SM can have 32 simultaneously active contexts. Each such thread of execution is called a thread *block* in CUDA [NVIDIA 2010]. Figure 4(b) shows the GPU CUDA programming model. The GPU execution is broken down into thread blocks. Each thread block consists of multiple threads that execute the same thread program. The CUDA programming model groups these threads into *thread warps* of 32 threads each. All threads within a thread

warp execute a single instruction on different data elements; this is essentially the Single Instruction Multiple Data (SIMD) execution model. Since the hardware has only 8 or 16 execution units (SPs), each GPU instruction (corresponding to 32 data elements) executes in multiple cycles (4 cycles on the GTX 280 with 8 execution units, and 2 cycles on the GTX 480 with 16 execution units). For more details on GPU architecture and programming, please refer to the CUDA programming guide [NVIDIA 2010].

GPUs provide hardware support for gather/scatter instructions at a half-warp granularity (16 lanes) [NVIDIA 2010], and hence we explored the complete spectrum of exploiting SIMD:

(a) Searching *32* keys, one per SIMD lane ($d_{\mathcal{K}} = 1$),
(b) Searching *one* key, and exploiting the 32-element wide SIMD ($d_{\mathcal{K}} = 5$),
(c) Searching between one and 32 keys using all SIMD lanes, with $d_{\mathcal{K}} = 2$, 3, or 4.

Since the GPUs do not explicitly expose caches, the cache line width ($d_{\mathcal{L}}$) was set to be same as $d_{\mathcal{K}}$. 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 $d_{\mathcal{P}}$, we did not see any change in runtime. Hence, $d_{\mathcal{P}}$ is assigned equal to $d_{\mathcal{N}}$.

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, computes the index, and updates the output page. This amounts to running two instances of tree creation per warp, while using two of the SIMD lanes. 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 slowdown in runtime 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 runtime 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 $10 \cdot 4 \cdot (2^{d_{\mathcal{N}}})$ cycles ($= 40 \cdot 2^{d_{\mathcal{N}}}$ cycles) per SM. Since GPUs provide a very high memory bandwidth, our tree creation is computation bound. For $\mathcal{N}$ as large as 64 million tuples, the runtime is around 2.6 billion cycles, which is less than 0.07 seconds on GTX 280.

5.2.2. Traversing the Tree. $d_{\mathcal{K}}$ 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 $d_{\mathcal{K}}$ less than 4 leads to a gather operation within the half-warp, and the search implementation is latency bound. Hence we choose $d_{\mathcal{K}} = 4$, 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 $d_{\mathcal{K}} = 4$. 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-SIMD-lane computation.

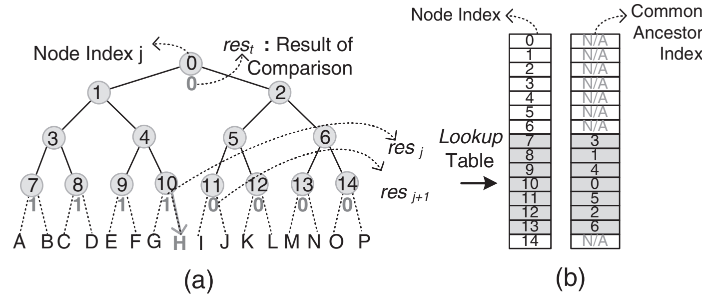

Fig. 5. Example of GPU tree search and the lookup table.

After comparing the key with the tree node element, each of the $\mathcal{N}_{\mathcal{K}}$ SIMD lanes stores the result of the comparison (res_i = 0/1) in a preallocated space. Consider the *eight* leaf nodes in Figure 5(a) (assume $\mathcal{N}_{\mathcal{K}} = 15$). 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 $res_0 = 0$, $\texttt{key}_q$ is *less than* the key stored at node$_0$, and hence the node belongs to its left subtree, thereby setting $index \leftarrow$ `H` (shown in Figure 5). In case res_0 is equal to 1, *index* is set to `I`. We precompute 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 5(b) shows this lookup table for our specific example, and similar tables can be built for other values of $d_{\mathcal{K}}$. The total size of the lookup table is $\mathcal{N}_{\mathcal{K}}$ (=15 for our example).

Figure 6 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-SIMD-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 $d_{\mathcal{K}} = 4$, 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 computation bound. Therefore, we implemented our software pipelining technique by evaluating multiple $\mathcal{S}$ values of 1 and 2. We found the best value of $\mathcal{S}$ to be 2. This further reduced the latency, and our resultant runtimes were within 5%–10% of the computation 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 $\mathcal{S}$ queries per warp, a total of 480$\mathcal{S}$ queries are required. With $\mathcal{S}$ being equal to 2, we operate the GPU at full throttle with 960 concurrent queries.

5.3. Intel MIC Implementation

The Intel MIC (Many Integrated Core) architecture is an Aubrey Isle [Seiler et al. 2008]-based silicon platform and Knights Ferry (KNF) [Skaugen 2010] is its first implementation with 32 cores running at 1.2GHz. It 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 computational 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

```
/* 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++
```

Fig. 6. GPU code snippet for index tree search.

well as a vector unit that supports sixteen 32-bit float or integer operations per clock. Vector instructions can be predicated by a mask register that controls which vector lanes are written. To improve SIMD efficiency, Knights Ferry also includes hardware support for gather and scatter operations, which allow loading from and storing to 16 noncontiguous memory addresses. Knights Ferry 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 multithreading. Since Knights Ferry has fully coherent caches, standard programming techniques such as pthreads or OpenMP can be used for ease of programming.

Since Knights Ferry features a 16-wide SIMD and hardware support for gather and scatter operations, there are multiple ways to perform searches between:

```
/*
TΔ: starting address of a tree
page_address: starting address of a particular page blocking sub-tree
page_offset: offset (# of tree nodes before the current node) within a particular page blocking
        sub-tree
level_offset: offset (# of tree nodes before the current node) at the current tree level
cache_offset: offset (# of tree nodes before the current node) within a particular cache line
        blocking sub-tree
*/

_M512 v_key_q, v_tree;
 v_key_q = _mm512_loadd(&key_q, _MM_FULLUPC_NONE, _MM_BROADCAST_1x16, 0);
/* v_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 SIMD blocking sub-tree (=4 levels of the tree)*/

        v_tree = _mm512_vloadunpackld(TΔ + page_address + page_offset);
        v_tree = _mm512_vloadunpackhd(TΔ + page_address + page_offset);
        /* v_tree: vector register Vtree. Load 16 tree nodes in Vtree*/

        __mmask v_mask = _mm512_cmpgt_pi(v_key_q, v_tree));
        /* v_mask: mask register Vmask. Set the mask register Vmask*/

        child_index = LookUp[v_mask];
        /* KNF supports inter-lane SIMD computation. Thus generating mask and using it to index
           the LookUp array greatly reduce the instruction overhead to select the next child node */

        /* SIMD width is 16 (512bit, the same as a cache line). Therefore, no separate cache line
           blocking is necessary */

        /* The below two lines are ncessary for proper address computation for traversing the next
           children node. The address computation overhead can be further reduced by vectorizing
           computation over multiple search queries */

        level_offset =  level_offset*16 + cache_offset;
        /* multiply by 16 because there are 16 leaf nodes in a cache blocking sub-tree */

        page_offset = (2^(4j) − 1) + level_offset*15
         /* 2^(4j) − 1 : # of tree nodes above the current cache line blocking sub-tree
        level_offset is multipled by 15 because each tree node has 15 tree nodes in a cache line
          blocking sub-tree */
    }
    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++
```

Fig. 7. Knights Ferry code snippet for index tree search.

(a) searching 16 keys, one per SIMD lane ($d_{\mathcal{K}} = 1$) and
(b) searching one key, and using the SIMD instruction ($d_{\mathcal{K}} = 4$) to speed up the search.

We choose $d_{\mathcal{K}} = 4$ since it performs best by removing the gather overhead just like SSE search on CPU and SIMD search on GPU. Setting $d_{\mathcal{K}} = 4$ levels enables subtree traversal of four levels with SIMD operations.

Figure 7 shows 16-wide SIMD traversal code for Knights Ferry. Unlike GPUs, Knights Ferry SIMD operations support inter-lane computation. Therefore, SIMD tree traversal code is very similar to SSE tree traversal code (Figure 3). The 512-bit SIMD comparison instruction generates a mask, which is used to index the LookUp array. Note that this 16-bit mask does not require a working set size of $2^{16} = 64$KB for the LookUp array. While we have the LookUp array with the size of 64KB for convenience,

Table I. Peak Computational Power (GFlops) and Bandwidth (GB/sec) on the Core i7 and the GTX 280

Platform	Peak GFlops	Peak BW
Intel Core i7 processor 975 (Nehalem)	103.0	30
NVIDIA GTX 280	933.3	141.7

note that there are only 16 distinct possibilities, hence the actual working set size only needs at most 16 separate cache lines, resulting in less than 1KB of the array being cache resident.

The only difference between the Knights Ferry code and the SSE code on CPUs is that $d_{\mathcal{K}} = 4$ levels are used for Knights Ferry while SSE has $d_{\mathcal{K}} = 2$ levels. No separate cache line blocking is necessary for Knights Ferry since $d_{\mathcal{L}}$ is assigned equal to $d_{\mathcal{K}}$. Setting $d_{\mathcal{L}}$ equal to $d_{\mathcal{K}}$ decreases the address computation overhead because there is no need to compute the address for a child SIMD subtree.

To reduce the latency impact in tree traversal, we use the software pipelining technique just like SSE search on CPUs. We perform $\mathcal{S}$ simultaneous queries by unrolling the loop over different queries. The value of $\mathcal{S}$ is set to be equal to four for our experiments. Therefore, we need a total of 512 (=128(threads) * 4(simultaneous queries)) concurrent queries to achieve peak throughput. To further reduce the address computation overhead, we actually use the SIMD code to perform address computation over $\mathcal{S}$ simultaneous queries.

6. PERFORMANCE EVALUATION

We now evaluate the performance of FAST on an quad-core Core i7 CPU and a GTX 280 GPU. Peak flops (computed as frequency · core count · SIMD width) and peak bandwidth of the two platforms are shown in Table I. 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[9]. 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.

6.1. Impact of Various Optimizations

Figure 8 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 falls in between the two analyzed cases and is 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

[9]64M is the max. number of tuples that fit in GTX 280 memory of 1GB.

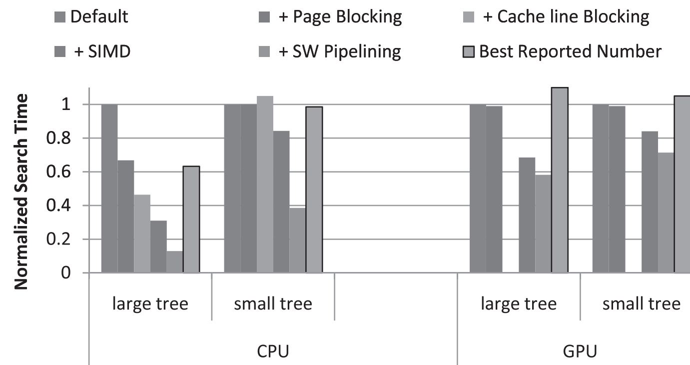

Fig. 8. Normalized search time with various architectural optimization techniques (lower is faster). The fastest reported performance on CPUs [Schlegel et al. 2009] and GPUs [Alcantara et al. 2009] is also shown (for comparison).

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. 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 [Schlegel et al. 2009]. As shown in Figure 8, 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 computation bound, and not bound by various other architectural latencies. Applications that are latency bound do not exploit the additional computational 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 (Figure 8) 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 computation bound.

6.2. CPU Search VS. GPU Search

We compare the performance of search optimized for CPU and GPU architectures. Figure 9 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

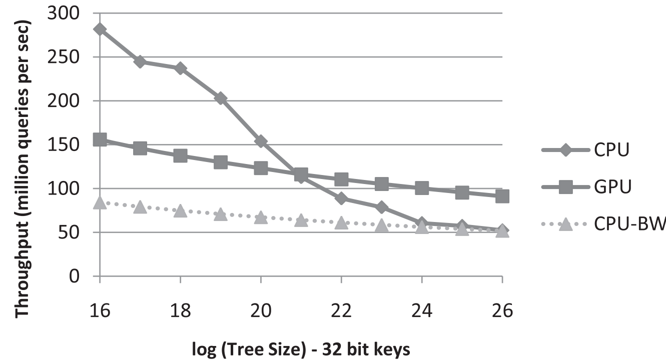

Fig. 9. Comparison between the CPU search and the GPU search. "CPU-BW" shows the throughput projection when CPU search becomes memory bandwidth bound.

Table II. Peak Computational Power (GFlops) and Bandwidth (GB/sec) on the Intel Xeon processor X5670 (Westmere) and the NVIDIA Tesla C2050

Platform	Peak GFlops	Peak BW
Intel Core i7 processor 975 (Nehalem)	103.0	30
Intel Xeon processor X5670 (Westmere)	278	60
NVIDIA GTX 280	933.3	141.7
NVIDIA Tesla C2050	1030	144

as compared to the CPU. On the other hand, GPUs have 30 Streaming Multiprocessors (SMs) each with 1.3 GHz while CPUs have 4 cores each with 3.2 GHz, resulting in roughly 3X more total ops/second than CPUs. 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) $\sim$2X in favor of CPUs.

As the tree size grows, CPUs suffer from TLB misses and 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 computation 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 section, we show how to further improve CPU search performance by easing bandwidth bottleneck with compression techniques.

Recent work [Kaldewey et al. 2009] has shown that GPUs can improve search performance by an order of magnitude over CPUs and combining Figures 8 and 9 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 computational flops and bandwidth (Table I). Thus optimized CPU search is much better than optimized GPU search in terms of architecture efficiency, defined as achieved performance as compared to peak computational power.

6.3. Search on State-of-Art Processors

In this section, we measure the performance of FAST on recent CPU and GPU platforms, and also on the Many Integrated Core (MIC) platform. The CPU platform we use is the 12-core (two sockets) Intel Xeon X5670 (Westmere) processor running at 2.9 GHz. The GPU platform is the NVIDIA Tesla C2050 that has 15 Simultaneous Multiprocessors (SMs), each with 2 groups of 16 scalar processors (hence 2 sets of 16-element wide SIMD units). The peak GFlops and peak BW are shown in Table II. To

Table III. Measured Performance Comparison across Five Different Platforms – Nehalem (CPU), Westmere (CPU), GTX 280 (GPU), Tesla C2050 (GPU)

	Throughput (million queries per sec)	
	Small Tree (64K keys)	Large Tree (16M keys)
Nehalem (CPU - single-socket)	280	60
Westmere (CPU - dual-socket)	603	117
GTX 280 (GPU)	150	100
Tesla C2050 (GPU)	272	149

compare search throughput with CPU/GPU search, we also measure the performance on the MIC platform-based Knights Ferry processor.

Table III shows search throughput on the Westmere CPU, Tesla GPU, and Knights Ferry for small trees (with 64K keys) and large trees (with 16M keys). As shown in the table, the small tree search on Westmere is about 2.1X better than the Core i7 Nehalem processor. This is because tree search on small trees is bound by computational resources available on Westmere (as is the case for Nehalem). Since the two processors have similar microarchitecture, performance scales in accordance with 2.7X additional computational resources available on Westmere (as shown in Table III). For large trees, search performance is bound by the available memory bandwidth. Since the memory bandwidth on dual-socket Westmere is about 2X higher than on Nehalem, the search performance on large trees improves by 2X.

On the GPU platforms, the Tesla C2050 architecture has about 1.7X more computational power[10] than the GTX 280. While the physical SIMD width on Tesla is twice that on the GTX 280, the logical SIMD width (warp width) remains 32. To avoid gathers/scatters, we still perform searches at the granularity of half-warps (2 queries per warp). The performance scales 1.8X for small trees and about 1.5X for large trees over the GTX 280, in line with the ratio of the computational resources.

Knights Ferry takes the best of both architectures: a large cache in CPUs and high computing power and high bandwidth in GPUs. For small trees, Knights Ferry is able to store the entire tree within L2 cache, and is therefore computation bound. Compared to the latest CPU (Westmere) and GPU (Tesla) platforms, search on Knights Ferry processors results in 667 million queries per sec, providing a speedup of 1.1X and 2.4X respectively. The speedup numbers over CPUs are in line with the peak computational power of these devices, after accounting for the lack of multiply-add operations and the fact that search can only obtain a speedup of $\log(\mathcal{K})$ using $\mathcal{K}$-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 computation bound while the remaining 8-level traversal is bandwidth bound. Large tree search on Knights Ferry processors results in 183 million queries per sec, providing a speedup of 1.6X over CPUs and 1.2X over GPUs.

6.3.1. 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 at least 64 concurrent queries to be available while our GPU search requires, at minimum, 960 concurrent queries once the overheads of thread spawning and

[10]Without counting SFU flops on GTX280, the actual flops on GTX280 is 622 GFlops.

instruction cache miss have been amortized over a few thousand queries. The absolute minimum number of concurrent queries on the CPU is 8 (since there are 8 hardware threads) while that on the GPU is 240 (since there are 240 total Scalar Processors). On the CPU, running with the minimum of 8 concurrent queries results in 1.4X performance drop (this is about 1.9X on GPUs). Increasing the number of concurrent queries at the expense of response time may further improve performance by minimizing the effect of TLB and cache misses and the results are discussed in Section 8. The best buffering size and the implications on query processing models are discussed in the papers Boncz et al. [2005] and Cieslewicz et al. [2009].

7. COMPRESSING KEYS

In the previous section, we presented runtimes with key size ($\mathcal{E}$) 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[11] for CPUs. Since CPU search is memory bandwidth bound for the 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 computation bound for 4-byte keys, and will continue to be computation 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.

7.1. Handling Variable-Length Keys

We first present computationally simple compression scheme that maps variable input length data to an appropriate fixed length with a 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 runtime for large keys.

7.1.1. Compression Algorithm. We compute a fixed-length ($\mathcal{E}_p$ 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, that is, 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 [Belazzougui et al. 2009], the order-preserving hash functions require O($\mathcal{N} \log_2 \mathcal{N}$) storage and computation time [Fox et al. 1991]. However, we need to compute and evaluate such functions in constant (and small) time.

We extend the prefix compression scheme [Bayer and Unterauer 1977] to obtain order-preserving partial keys. These schemes compute a contiguous common prefix of the set of keys, and store the subsequent part of the keys as the partial key. Instead,

[11]The original tuple data may be independently compressed using other techniques [Abadi et al. 2006] and used in conjunction with our scheme.

Fig. 10. Example of extracting 8-bit partial keys in five keys.

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 ($\mathcal{G}$), 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 $\mathcal{G}$ bits from all the keys. In case all the keys have the same $\mathcal{G}$ bits, we consider the next $\mathcal{G}$ bits, and repeat the process. In case they are different, we append these $\mathcal{G}$ bits to the partial key computed so far. We repeat this process and keep appending the set of $\mathcal{G}$ bits until a $\mathcal{E}_p$ 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 10 shows an example for computing an 8-bit partial key for five keys with $\mathcal{G} = 4$. Note that our scheme is a generalization of computing the common prefix; we extract a noncontiguous common prefix, with the discarded intermediate $\mathcal{E}_p$ 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 $\mathcal{G}$. 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 $\mathcal{G}$ is concerned, it is affected by two competing factors: (a) the smaller the value of $\mathcal{G}$, the higher the chances of forming partial keys with few false positives; (b) the larger the value of $\mathcal{G}$, the lower the cost of computing the partial key from the original key. Furthermore, too small a $\mathcal{G}$ may require high cost of bit manipulation functions, thereby increasing the overhead of compressing/decompressing the keys. Also, $\mathcal{G}$ should be smaller than size of a character (8 bits) to be able to compress string-based keys. We choose $\mathcal{G} = 4$ bits, which provides the right balance between the various factors described before. Also, choosing $\mathcal{G} = 4$ bits eliminates the need to look across multiple architecture registers (of width 32 bits (a multiple of 4 bits) for each SIMD lane) while computing the partial key, and hence simplifies the computation.

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 $\mathcal{G}$ bits at a time. If the $\mathcal{G}$ bits are all 0, then the result of all exclusive-or operations was 0, and hence all the keys have the relevant $\mathcal{G}$ 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 $\mathcal{G}$ 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 $3\lceil\mathcal{E}/16\rceil$ 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 be 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 thirty-two 4-bit chunks, with a 32-bit *pmask*. $pmask_i = 1$ implies the ith 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 sixteen 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 $\mathcal{B}_{7..0}$) of data consists of two 4-bit elements ($\mathcal{B}_{7..4}$ and $\mathcal{B}_{3..0}$). Consider $\mathcal{B}_{7..4}$. 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 $\mathcal{B}_{3..0}$. 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 ($\mathcal{B}_{3..0}$ 0000), (0000 $\mathcal{B}_{3..0}$), ($\mathcal{B}_{7..4}$ 0000) and (0000 $\mathcal{B}_{7..4}$). 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 $12\lceil\mathcal{E}/16\rceil$ ops per key. Including the cost to compute the *pmask*, the total cost for compression is $15\lceil\mathcal{E}/16\rceil$ ops. For a 512MB tree structure with 16-byte keys, our compression algorithm takes only 0.05 seconds on Core i7.

7.1.2. 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 exponentially. 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 $2^{(d_{\mathcal{N}}-d_{\mathcal{P}})}$ tuples to the left of the actual match, leading to large slowdowns. Although Bohannon et al. [2001] propose storing links to the actual keys, this leads to further TLB/LLC misses. Instead, we solve the problem by computing an $\mathcal{E}_p$ that leads to less than 0.01% false positives for the keys in the first page. We start with $\mathcal{E}_p = 32$ bits, and compute the partial keys, and continue doubling $\mathcal{E}_p$ until our criterion for false positives is satisfied. In practice, with $\mathcal{E}_p = 128$ bits (16 bytes), we saw no false positives for the first page of the tree (for $\mathcal{E} \leq 100$).

For all subsequent pages, we use partial keys of length ($\mathcal{E}_p = 32$ 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 the compressed index tree increases to $(20 + 15\lceil\mathcal{E}/16\rceil)$ ops per key. The compressed tree construction is still computation bound, and only around 75% slower than the uncompressed case (for $\mathcal{E} \leq 16$).

7.1.3. Traversing the Compressed Tree. During the tree traversal, we do not decompress the stored partial keys for comparison. Instead we compress $\texttt{key}_q$ (the query key)

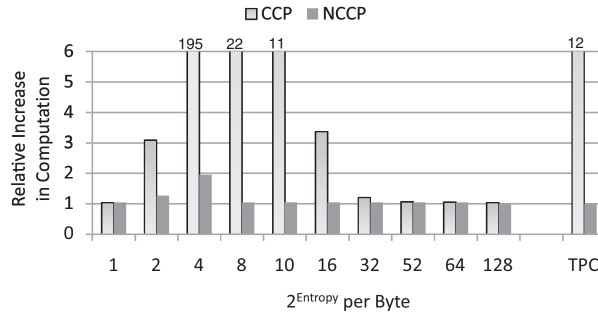

Fig. 11. Relative increase in computation with respect to various alphabet sizes ($2^{entropy}$) per byte. For example, 52 means that we generate random keys for each byte among 52 values.

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 $12\lceil \mathcal{E}/16 \rceil$ 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 $\mathcal{E} = 16$, we would have accessed one cache line for 2 levels, and hence the bandwidth requirement is reduced by ~2X by storing the order-preserving partial keys. For $\mathcal{E} \geq 32$, the bandwidth reduction is 4X and beyond. This translates to significant speedups in runtime over using long(er) keys.

7.1.4. 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 11, we compare our compression scheme, NonContiguous Common Prefix (NCCP) with the previous scheme, Contiguous Common Prefix (CCP) and report the relative increase in computation for searching 16-byte keys with various entropies. The increase in computation is indicative of the number of false positives, which increases the amount of redundant work done, thereby increasing the runtime. A value of 1 implies 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 observed for the TPC-H data using our scheme, with the competing scheme reporting around 12X increase.

Figure 12 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

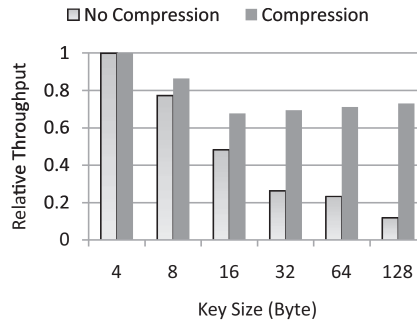

Fig. 12. Throughput comparison between no compression and compression with key size from 4 to 128 bytes.

size of the uncompressed keys is ∼1GB. All numbers are normalized to the throughput achieved using 4-byte keys. Without our compression scheme, the throughput 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 utilizes 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 7.1.3) and varies between 70%–80% for key sizes varying between 16 and 128 bytes. The overall runtime speedup is around 1.5X for 16-byte keys, and increases to around 6X for 128-byte keys.

Thus, for input consisting of variable-length keys, the throughput of a naively constructed index tree may be up to 6X slower than our scheme of compressing such variable-length keys into partial keys of length of 4 bytes each, and using FAST-based scheme to build and traverse the resultant index tree.

7.2. 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 runtime, since runtime for the last few levels in trees is dependent on the available memory bandwidth. We now describe a lightweight compression scheme for such integer keys, widely used as primary keys for OLAP database tables.

7.2.1. 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 (= $\lceil \log_2(K_{max} - K_{min} + 1) \rceil$), 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.

7.2.2. Traversing Compressed Trees. We read K_{min} and δ at the start of the compressed page, and compute $(\mathtt{key}_q - K_{min})$ for the query key, termed as $\mathtt{key}_{q_\delta}$. We directly compare $\mathtt{key}_{q_\delta}$ 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 an SSE-friendly implementation [Willhalm et al. 2009]. 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 $d_{\mathcal{K}}$ ($\geq$6) levels, as opposed to 4 levels without compression. 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 runtime reduction. As far as the compression ratio is concerned, we achieve 2–4X compression on the randomly generated integer datasets, and the runtime reduces by 15%–20% as compared to the runtime without using our compression technique.

8. THROUGHPUT VS. RESPONSE TIME

In this section, we compare four different techniques for search queries based on the user/application response time. Specifically, we compare the FAST search algorithm on uncompressed index tree (with key sizes of 4 bytes), FAST on compressed index trees (using the compression algorithm described in Section 7.2), FAST using a buffering scheme, and a sort-based scheme for performing search queries. We first describe the algorithms and later analyze and compare the respective performances achieved on CPUs (Core i7 975). These four schemes provide a spectrum of schemes with response times varying from very stringent ($\leq$1ms) to more tolerant ($\geq$50ms). We analyze how these schemes differ in the amount of buffering required to temporarily store the queries before executing them to achieve peak throughput with the various schemes.

(a) *Unbuffered Scheme Using FAST.* We execute the FAST search algorithm with uncompressed index trees. The key sizes are each 4 bytes. The search queries are executed similar to the algorithm described in Section 5.1.2. The number of simultaneous queries required to achieve close to peak throughput is around 64 (Section 5.1.3).

(b) *Unbuffered-Compressed Scheme Using FAST.* We assume 4-byte keys with low entropy, and hence can compress them using our novel order-preserving partial key computation scheme described in Section 7.2. Since compression reduces the effective size of each key (in the tree), we can now pack more than 16 keys in each cache line, thereby reducing the total number of cache lines read to perform the search. Similar to (a), the number of simultaneous queries required to achieve close to peak throughput is 64.

Both (a) and (b) schemes given before describe algorithms that require small (around 64) buffer length to queue up the queries and perform search operations at close to peak throughput using our FAST layout and traversal algorithm. We now describe two other schemes which require a comparatively larger buffer space (and higher response time), but can potentially increase the achieved throughput (processed queries per unit of time).

(c) *Buffered Scheme Using FAST.* We implemented a variant of the buffered search algorithm [Zhou and Ross 2003]. We allocate a buffer for all the children nodes of the

leaves of the subtree 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. The main advantage of buffering queries is the fact that the queries in each buffer need to traverse down a certain subtree, and therefore exhibit coherence in data access, and can amortize the cost of TLB and cache misses. A query is searched in the subtree corresponding to the first page of the layout, and enqueued in the buffer corresponding to the appropriate leaf node (and hence needs to traverse the subtree rooted at it) emanating from the subtree.

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:

(i) a predefined threshold of batch of input queries have been stored at these intermediate buffers, or
(ii) when any of the buffers is filled up.

The buffering technique reduces TLB misses that would have incurred with the unbuffered scheme (from one per query to one per group), and also exploits 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 runtime. Note that an increase in the size of the buffer(s) also proportionately increases the number of queries that need to be queued up, thereby increasing the response time. In contrast, larger sized buffers also increase the coherence in the queries that need to be searched in the respective subtrees, thereby leading the increase in throughput. Computing the appropriate size of buffers reduces to an optimization problem, and is a function of the overall tree depth and the distribution of input queries. For our experiments, we used a buffer size of 64–128 entries for input tree(s) varying between 16K and 64M keys.

(d) *Sort-Based Scheme.* Sort-based techniques represent one of the extremes of the trade-off between response time and the achieved throughput. 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. Such techniques have become attractive due to the recent advancement in sorting algorithms that have sped up the sorting throughput by orders of magnitude [Chhugani et al. 2008; Satish et al. 2009].

For our implementation, we sort the queries using an efficient implementation of merge-sort [Chhugani et al. 2008]. The linear scan and comparison is also performed using an SIMD-friendly merge pass. 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.

Performance Comparison. In Figure 13, we plot the throughput with respect to obtained response time for Core i7 975. Both unbuffered FAST and unbuffered-compressed FAST schemes require only 64 simultaneous queries to achieve peak throughput. The number of simultaneous queries required is computed as the product of the number of threads executing on the CPU and the number of simultaneous queries per thread (around 8). Even with the increase in the number of cores on CPUs and KNF, we expect the number of simultaneous queries required to be around a few hundred simultaneous queries. Such a small number of queries corresponds to ≤1ms of response time. For the Core i7 975, the throughput for compressed index trees is around 20% greater than the throughput on uncompressed trees (Section 7.2). Even for modern

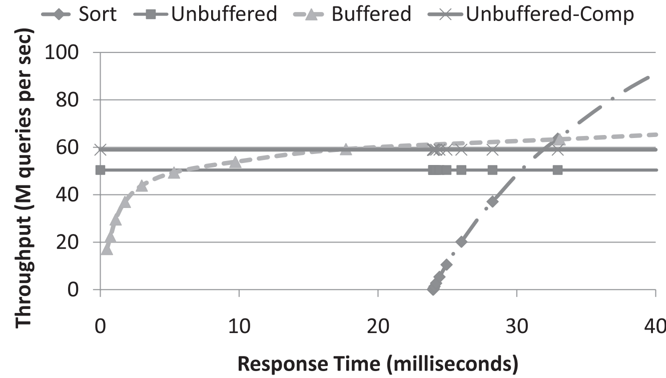

Fig. 13. Throughput vs. Response time with various techniques.

GPUs with thousands of threads, the number of simultaneous queries would be of the order of a few thousands, with the resultant response time still being ≤1ms.

In contrast, the buffered scheme using FAST provides a peak throughput of around 5% larger than the compressed case, but requires a much larger batch (∼640K) of input queries, that corresponds to response time of around 20–30ms. For a response time in the range of 1–10ms, the throughput of buffered scheme using FAST increases from around 20 million queries per second to around 40–50M queries per second. The throughput exceeds the throughput of unbuffered FAST scheme at around 8ms response time, and the corresponding cross-over of unbuffered-compressed scheme occurs for a response time of around 15ms. Furthermore, for response times of around 40–50ms, the throughput is only 5% larger than the unbuffered-compressed case.

As far as the sort-based scheme is concerned, the peak throughput is almost 1.5X greater than the compressed FAST trees. However, this peak performance is only achieved when the application can tolerate latencies of 33ms and larger (corresponding to ≥2M queries). An interesting observation is that the sort-based schemes have a very small throughput for up to 25ms response time. This is due to the fact that we need to scan through the complete input tree before producing any output, which corresponds to a large overhead even for a relatively small number of queries. The throughtput for sort-based schemes exceeds the throughput of FAST-based buffered schemes at a response time of around 33ms.

Facilitating the DBMS query optimizer. For a given real-world database application and use-case scenario, the query optimizer can exploit the preceding described algorithms to choose/predict the best algorithm that maximizes the throughput for a given response time. For example, in case of index nested loop joins, a join with an outer-table size of ≤2M elements should use a FAST-based scheme to perform efficient join operations and achieve throuput of around 50–60M elements/second. For cases with a larger outer-table size (≥2M elements), a sort-based scheme should be used to achieve higher throughputs (up to 90M elements/second). Note that the sort-based scheme in case of nested join corresponds to the sort-merge join scheme [Graefe et al. 1994].

To summarize, for scenarios requiring response times of ≤1ms, the query optimizer should choose either unbuffered FAST and unbuffered-compressed FAST. For more tolerant applications (in terms of response-time), sort-based schemes become an attractive alternative for the query optimizer, and can provide up to 1.5X larger throughput at 33–50ms response time for the queries.

9. DISCUSSIONS

A complete discussion of database use cases for FAST is out of scope of this article, but a brief set of ideas for the ways in which FAST could be utilized is discussed here. Clearly, FAST will have the most impact on in-memory database use-cases. Some in-memory databases (e.g., Oracle TimesTen) have used variants of binary search trees such as T-Trees directly to manage indexes on user data, and FAST could be used in place of T-Trees. Another application of FAST would be to use FAST to speed up the searches of sorted lists in B-Tree index leaf nodes. The low-latency search by FAST can also be applied to speed up multiple concurrent select queries used in highly parallel databases like staged database system [Harizopoulos and Ailamaki 2003] or STEPS [Harizopoulos and Ailamaki 2006] to combine various OLTP queries.

Faster index traversals using FAST can be used to push out the join selectivity crossover-point for the query optimizer (i.e., for example, how low does the selectivity need to be before hash join is less expensive than tree-based index join). Note that using hash join has a significant impact on response time (to get the first result row) since it is a blocking operation on the inner side (while the hash table is built on the smaller relation). For join operations with small outer table sizes, FAST-based index join is more efficient than hash-join. For larger outer table sizes, the lower computational complexity of hash-join will result in better performance for equi-joins where hash-based techniques have been shown efficient. However, for band-joins [Lu and Tan 1995], where the join predicate involves range searches, it is difficult to extend hash-based methods. For such cases, a FAST-based tree search method is a good candidate.

10. CONCLUSIONS AND FUTURE WORK

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.

In terms of future work, further work is needed to evaluate the trade-offs between in-place updates and bulk updates with respect to how well they can exploit future trends towards increasing cores and SIMD width.

ACKNOWLEDGMENTS

The authors would like to thank Erin Richards for helping to revise the article. Also, the authors thank the anonymous reviewers for their thoughtful comments.

REFERENCES

Abadi, D., Madden, S., and Ferreira, M. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 671–682.

Alcantara, D. A., Sharf, A., Abbasinejad, F., Sengupta, S., et al. 2009. Real-Time parallel hashing on the GPU. *ACM Trans. Graph. 28,* 5.

Allan, V. H., Jones, R. B., Lee, R. M., and Allan, S. J. 1995. Software pipelining. *ACM Comput. Surv. 27,* 3, 367–432.

Arge, L. 2003. The buffer tree: A technique for designing batched external data structures. *Algorithmica 37,* 1, 1–24.

Bayer, R. and Unterauer, K. 1977. Prefix b-trees. *ACM Trans. Datab. Syst. 2,* 1, 11–26.

BELAZZOUGUI, D., BOLDI, P., PAGH, R., AND VIGNA, S. 2009. Theory and practice of monotone minimal perfect hashing. In *Proceedings of the 12th Workshop on Algorithm Engineering and Experiments (ALENEX)*. 132–144.

BINNIG, C., HILDENBRAND, S., AND FÄRBER, F. 2009. Dictionary-Based order-preserving string compression for column stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 283–296.

BOHANNON, P., MCLLROY, P., AND RASTOGI, R. 2001. Main-Memory index structures with fixed-size partial keys. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 163–174.

BONCZ, P. A., ZUKOWSKI, M., AND NES, N. 2005. Monetdb/x100: Hyper-pipelining query execution. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

CHEN, S., GIBBONS, P. B., AND MOWRY, T. C. 2001. Improving index performance through prefetching. *SIGMOD Rec. 30,* 2, 235–246.

CHEN, S., GIBBONS, P. B., MOWRY, T. C., ET AL. 2002. Fractal prefetching b+-trees: optimizing both cache and disk performance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 157–168.

CHHUGANI, J., NGUYEN, A. D., LEE, V. W., MACY, W., HAGOG, M., CHEN, Y.-K., BARANSI, A., KUMAR, S., AND DUBEY, P. 2008. Efficient implementation of sorting on multi-core SIMD CPU architecture. In *Proceedings of the International Conference on Very Large Data Bases (VLDB) 1,* 2.

CIESLEWICZ, J., MEE, W., AND ROSS, K. A. 2009. Cache-Conscious buffering for database operators with state. In *Proceedings of the 5th International Workshop on Data Management on New Hardware*.

CIESLEWICZ, J. AND ROSS, K. A. 2007. Adaptive aggregation on chip multiprocessors. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 339–350.

COMER, D. 1979. Ubiquitous b-tree. *ACM Comput. Surv. 11,* 2, 121–137.

FOX, E. A., CHEN, Q. F., DAOUD, A. M., AND HEATH, L. S. 1991. Order-Preserving minimal perfect hash functions. *ACM Trans. Inf. Syst. 9,* 3, 281–308.

GOLDSTEIN, J., RAMAKRISHNAN, R., AND SHAFT, U. 1998. Compressing relations and indexes. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 370–379.

GRAEFE, G. AND LARSON, P.-A. 2001. B-tree indexes and cpu caches. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 349–358.

GRAEFE, G., LINVILLE, A., AND SHAPIRO, L. D. 1994. Sort versus hash revisited. *IEEE Trans. Knowl. Data Engin. 6,* 6, 934–944.

GRAEFE, G. AND SHAPIRO, L. 1991. Data compression and database performance. In *Applied Computing*, 22–27.

HANKINS, R. A. AND PATEL, J. M. 2003. Effect of node size on the performance of cache-conscious b+-trees. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*. 283–294.

HARIZOPOULOS, S. AND AILAMAKI, A. 2003. A case for staged database systems. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

HARIZOPOULOS, S. AND AILAMAKI, A. 2006. Improving instruction cache performance in oltp. *ACM Trans. Database Syst. 31,* 3, 887–920.

HOLLOWAY, A. L., RAMAN, V., SWART, G., AND DEWITT, D. J. 2007. How to barter bits for chronons: Tradeoffs for database scans. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 389–400.

IYER, B. R. AND WILHITE, D. 1994. Data compression support in databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 695–704.

KALDEWEY, T., HAGEN, J., BLAS, A. D., AND SEDLAR, E. 2009. Parallel search on video cards. In *Proceedings of the USENIX Workshop on Hot Topics in Parallelism*.

KIM, C., CHHUGANI, J., SATISH, N., SEDLAR, E., NGUYEN, A. D., KALDEWEY, T., LEE, V. W., BRANDT, S. A., AND DUBEY, P. 2010. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 339–350.

KIM, C., SEDLAR, E., CHHUGANI, J., KALDEWEY, T., NGUYEN, A. D., BLAS, A. D., LEE, V. W., SATISH, N., AND DUBEY, P. 2009. Sort vs. hash revisited: Fast join implementation on multi-core CPUs. In *Proceedings of the International Conference on Very Large Data Bases (VLDB) 2,* 2, 1378–1389.

LEHMAN, T. J. AND CAREY, M. J. 1986. A study of index structures for main memory database management systems. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 294–303.

LEISCHNER, N., OSIPOV, V., AND SANDERS, P. 2009. Fermi Architecture White paper.

LU, H. AND TAN, K.-L. 1995. On sort-merge algorithm for band joins. *IEEE Trans. Knowl. Data Engin. 7,* 3, 508–510.

NVIDIA. 2010. *NVIDIA CUDA Programming Guide 3.2*. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming-Guide.pdf.

RAO, J. AND ROSS, K. A. 1999. Cache conscious indexing for decision support in main memory. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 78–89.

RAO, J. AND ROSS, K. A. 2000. Making b+- trees cache conscious in main memory. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 475–486.

REILLY, M. 2008. When multicore isn't enough: Trends and the future for multi-multicore systems. In *Proceedings of the Annual High-Performance Embedded Computing Workshop (HPEC)*.

SATISH, N., HARRIS, M., AND GARLAND, M. 2009. Designing efficient sorting algorithms for manycore GPUs. In *International Symposium on Parallel and Distributed Processing*. 1–10.

SCHLEGEL, B., GEMULLA, R., AND LEHNER, W. 2009. k-ary search on modern processors. In *Proceedings of the Workshop on Data Management on New Hardware (DaMoN)*. 52–60.

SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., AND HANRAHAN, P. 2008. Larrabee: A many-core x86 architecture for visual computing. *Trans. Graph. 27,* 3.

SKAUGEN, K. B. 2010. Keynote at international supercomputing conference. In *Proceedings of the Annual High-Performance Embedded Computing Workshop (HPEC)*

WILLHALM, T., POPOVICI, N., BOSHMAF, Y., PLATTNER, H., ET AL. 2009. Simd-scan: Ultra fast in-memory scan using vector processing units. In *Proceedings of the International Conference on Very Large Data Bases (VLDB) 2,* 1, 385–394.

ZHOU, J. AND ROSS, K. A. 2002. Implementing database operations using simd instructions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 145–156.

ZHOU, J. AND ROSS, K. A. 2003. Buffering accesses to memory resident index structures. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 405–416.

ZUKOWSKI, M., HÉMAN, S., NES, N., AND BONCZ, P. A. 2006. Super-Scalar RAM-CPU cache compression. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 59.

Received October 2010; revised April 2011; accepted July 2011