IIU: Specialized Architecture for Inverted Index Search

Jun Heo* Jaeyeon Won* Yejin Lee* Shivam Bharuka†§

Jaeyoung Jang‡ Tae Jun Ham* Jae W. Lee*

*Seoul National University, †Facebook, Inc., ‡Sungkyunkwan University

{j.heo, jerrywon, yejinlee, taejunham, jaewlee}@snu.ac.kr, shivamb@fb.com, jaey86@skku.edu

Abstract

Inverted index serves as a fundamental data structure for efficient search across various applications such as full-text search engine, document analytics, and other information retrieval systems. The storage requirement and query load for these structures have been growing at a rapid rate. Thus, an ideal indexing system should maintain a small index size with a low query processing time. Previous works have mainly focused on using CPUs and GPUs to exploit query parallelism while utilizing state-of-the-art compression schemes to fit the index in memory. However, scaling parallelism to maximally utilize memory bandwidth on these architectures is still challenging. In this work, we present IIU[1], a novel **i**nverted **i**ndex processing **u**nit, to optimize the query performance while maintaining a low memory overhead for index storage. To this end, we co-design the indexing scheme and hardware accelerator so that the accelerator can process highly compressed inverted index at high throughput. In addition, IIU provides flexible interconnects between modules to take advantage of both intra- and inter-query parallelism. Our evaluation using a cycle-level simulator demonstrates that IIU provides an average of 13.8× query latency reduction and 5.4× throughput improvement across different query types while reducing the average energy consumption by 18.6×, compared to Apache Lucene, a production-grade full-text search framework.

CCS Concepts • **Computer systems organization** → *Special purpose systems*; *Data flow architectures.*

Keywords full-text search; inverted index; domain-specific architecture; accelerator; hardware/software co-design

ACM Reference Format:
Jun Heo, Jaeyeon Won, Yejin Lee, Shivam Bharuka, Jaeyoung Jang, Tae Jun Ham, and Jae W. Lee. 2020. IIU: Specialized Architecture for Inverted Index Search. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3373376.3378521

1 Introduction

Billions of people use computers to perform different tasks. Among all those different uses of computers, the search is arguably one of the most popular tasks. While there exist many different ways to implement a search engine, the most common way is to utilize an inverted index [1]. Inverted index is a key-value data structure, where a term (key) is associated with a sorted list of documents that contain the term (value). When a user of a search engine wants to retrieve a set of documents related to a particular term, the inverted index data structure enables the engine to quickly retrieve the list of documents containing that term instead of going through all documents and checking if the document contains the term. Once the list of documents is retrieved, the search engine often scores them using a specific metric [2,3] to return the list of top-scoring documents to the user.

In many real-world text search engines [4,5,6], inverted index is often compressed to reduce the size of the index so that (most of) it can fit in memory. For example, a relatively common term such as *business* appears in more than 12 billion documents in Google search, and thus the list of document identifiers (docIDs) containing this term can account for more than 48 GB memory space (assuming a 4B docID). Without compression, considering that there exist more than 133,248,235 unique words in popular document sets such as

[§]This work was done while Shivam Bharuka was a visiting researcher at Seoul National University.

[1]Pronunciation note: IIU is pronounced as "*to-you*" by reading the first two characters ("II") like Roman numeral *two*.

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378521

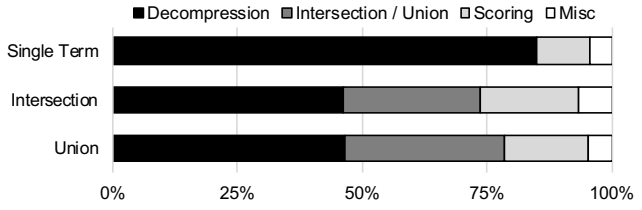

Figure 1. Breakdown of query processing time in Lucene

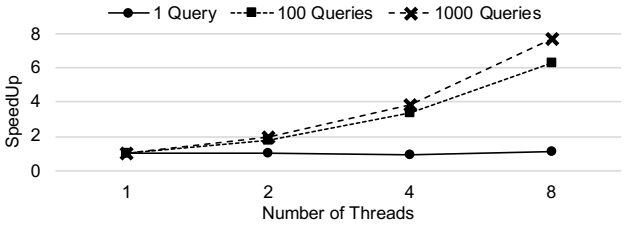

Figure 2. Performance scalability of Lucene with a varying number of queries

ClueWeb12 [7], it is highly likely that the size of the inverted index will exceed the amount of available physical memory and incur significant performance overhead.

A natural downside of the inverted index compression is that decompression needs to be performed every time the system needs to process a query with a specific term. As a result, it adds a substantial amount of latency to time-critical tasks. To demonstrate the importance of the decompression operation in search engines, we choose Apache Lucene as our baseline. It is an open-source search engine library powering many popular online services, such as Twitter, Netflix, Instagram, and Ebay [8,9]. Figure 1 shows a breakdown of the search time in Apache Lucene for three popular query types: single-term (e.g., "business"), intersection (e.g., "business" AND "cameo"), and union (e.g., "business" OR "cameo"). The result demonstrates that decompression accounts for over 40% of the total query response time over all three query types. Moreover, the rest of query time is mostly spent on set operations (intersection/union) and scoring the documents.

Unfortunately, conventional CPUs are not an ideal choice for text search engines. Inverted index operations often involve a series of simple, repetitive integer manipulation for which CPUs are not a cost-effective solution. According to our profiling using Intel VTune [10], it takes 70-100 instructions *per* docID for Lucene to perform these simple operations. In addition, as shown by Figure 2, most of the popular search engines (like Lucene) only exploit inter-query parallelism for throughput, but not intra-query parallelism, and hence their performance scales poorly when the number of backlogged queries is small.

Alternatively, there are proposals to use GPUs to accelerate index search, exploiting both inter- and intra-query parallelism [11, 12, 13]. However, there are other limitations in GPU-based index search implementations. In such systems, the limited capacity of GPU memory becomes a source of inefficiency. Moreover, they may underutilize GPU computation resources due to an insufficient amount of work at some stages of query processing [11].

To address these limitations, we propose IIU, a specialized hardware accelerator for inverted index search. Specifically, we co-design the indexing scheme (i.e., compression scheme and partitioning scheme) and the hardware accelerator together so that the accelerator can process highly compressed inverted index at high throughput. To achieve even higher throughput, our accelerator pipelines (i) the decompression of a sorted list, (ii) intersection/union of multiple sorted lists, and (iii) scoring the selected documents, so that all three operations can be executed in parallel. Finally, IIU leverages intra-query parallelism as well as inter-query parallelism to reduce the response time of a single query. In summary, we make the following contributions:

- We design and implement IIU, a specialized architecture for inverted index search, which accelerates both decompression and set operations for high-throughput low-latency query processing. We also extend IIU with scoring units to support full-text search engines.
- We devise a storage scheme for inverted index customized for our hardware implementation, which achieves both high compression ratio and scalability with additional hardware.
- We provide a detailed evaluation of IIU using a cycle-level simulator. Also, we synthesize our RTL design to estimate area and power. Compared to Apache Lucene IIU reduces the single-query latency by 13.8× and improves throughput by 5.4× on average across different query types, while saving energy consumption by 18.6×.

2 Background

Search engines utilize an inverted index structure to maintain information about billions of documents effectively. This index structure is often pre-constructed offline, and when a query arrives, the engine looks up this structure to efficiently serve the query within a few milliseconds. Here, we briefly explain common index compression schemes and a sequence of steps that a search engine takes to process a query.

2.1 Index Construction

Inverted Index. An inverted index (Figure 3) is a collection of sorted integer lists (called posting lists), one for each term appearing in the document corpus. The posting list consists of document identifiers of all documents containing that term. Each element in the posting list is defined as *posting*. The postings can also be used to contain other information such as term frequency, positional information, and document length.

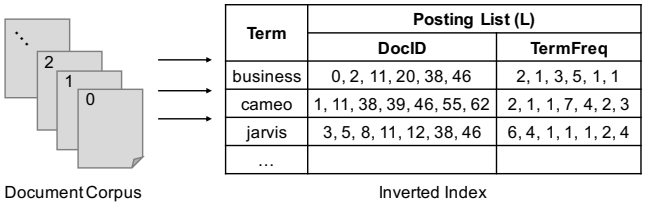

Figure 3. Inverted index example

Index Compression. Since the size of an inverted index is large, search engines compress it to minimize the storage overhead. There are many modern compression schemes such as VByte [14], Elias-Fano [15], PforDelta [16], NewPforDelta [17], OptPforDelta [17], SIMDPforDelta [18], and MILC [19]. These approaches partition the list into blocks and then compress the elements within each block. They maintain information about the range of values in a block for quick membership testing (i.e., checking if an element exists in a compressed block). For compression, they (except MILC) compute the deltas (d-gaps) between two consecutive document identifiers to minimize the bits to represent them. For example, L_Δ is a list of d-gaps, derived from L:

$$L\texttt{(business) = [0, 2, 11, 20, 38, 46]}$$

$$L_\Delta\texttt{(business) = [0, 2, 9, 9, 18, 8]}$$

PforDelta and Variants. PforDelta is a widely used compression algorithm which compresses data blocks of 128 d-gaps. It chooses the smallest number of bits b required to represent a majority (e.g., 90%) of elements called *regular values*. The remaining elements, called *exceptions*, are represented using 32 bits. It allocates 128 b-bit slots to store both regular values and position of the next exception while storing the actual exceptions at the end of the whole compressed sequence. There are many variants of PforDelta, which aim to reduce its storage overhead and increase the decompression speed. NewPforDelta stores the offset of exception and parts of the exception value in two additional arrays, which can be further compressed. OptPforDelta formulates the selection of b-bit for each block as an optimization problem. SIMDPforDelta reorders data elements so that a single CPU SIMD instruction can process multiple elements.

Memory Inverted List Compression (MILC). MILC [19] is a state-of-the-art compression scheme that reduces the query processing time of a compressed inverted list. It employs an offset-based encoding instead of d-gaps; for every element in a block, it stores the difference from the first element of the block instead of the previous element. It also utilizes dynamic partitioning with in-block compression to reduce the space overhead of compressed data. Our partitioning scheme (Section 3.2) is based on MILC. MILC further uses cache-aware and SIMD-aware policies to minimize the overhead of membership testing and improve the decompression performance on CPU. Note that IIU can easily be extended to query data compressed entirely using MILC.

2.2 Query Processing for Full-text Search

Query Types. Search engines support queries using terms and operators. The simplest query type is a single term query, which is a single-word search. These single term queries can be further combined using primitive set operators. The intersection ($\cap$) operator is used to return the common documents across the corresponding terms, whereas the union ($\cup$) operator is used to output all documents containing at least one of the terms. Assuming the inverted index in Figure 3, here are examples:

$$L\texttt{(business)} \cap L\texttt{(cameo) = [11, 38, 46]}$$

$$L\texttt{(business)} \cup L\texttt{(cameo) = [0, 1, 2, 11, 20, 38, 39, 46, 55, 62]}$$

Primitive set operations and single term query can be effectively used to create any complex query like a phrase or a prefix query. A phrase query is used to match documents that contain a list of single terms in a particular order. In this case, the positional information of each term is maintained in the postings, and an intersection query between their posting lists is used to obtain the candidate documents. Similarly, a prefix query, which matches documents containing terms with a specified prefix, is created using a union between the posting lists of all the terms with the prefix.

Workflow. To process a query, a search engine performs four key operations: (i) loads the posting list of each associated term into memory, (ii) decompresses them, (iii) performs set operation(s), and (iv) scores them for relevance and top-k selection. Below we describe each operation in details.

Query Load and List Decompression. The input is the query and the entire inverted index. The terms in the query are extracted, and the posting list of each term is loaded into memory. All the postings are decompressed for single term and union queries, whereas intersection queries use special data structures to reduce the number of decompression operations.

List Intersection. Fast processing of intersection between two posting lists is required to satisfy a quality-of-service (QoS) constraint. Two posting lists, L_0 and L_1, of similar lengths can be intersected efficiently using a linear merge algorithm due to their sorted order. They can be scanned in parallel to yield an overall runtime proportional to the length of the longer list. But this approach is wasteful when the lists have a low intersection rate or their sizes differ significantly. Thus, modern systems often employ Small-versus-Small (SvS) intersection algorithm [20, 21]. SvS takes advantage of the asymmetry of two lists to reduce the total number of comparisons. It orders the posting lists by their size and performs intersection starting from the two smallest lists. It uses a simple binary search algorithm to determine if

L	DocID	7	10	15	54	72	134	170	221	294	417	500	542
	TermFreq	11	2	1	1	5	3	1	2	4	1	3	7

L_Δ	d-gap	7	3	5	39	18	62	36	51	73	123	83	42
	TermFreq	11	2	1	1	5	3	1	2	4	1	3	7

Figure 4. A posting list for a term and its delta-encoding

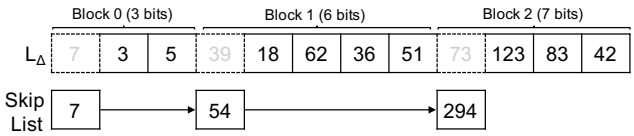

Figure 5. Example block structure of a posting list

a candidate posting from the first list appears in the second list. The overall runtime now depends on the length of the shorter list of the two. Furthermore, to avoid decompression of an entire posting list, the list is partitioned into *blocks*, and a *skip list* is maintained, which keeps the uncompressed value of the first element from each block. Binary search for a candidate posting can look up the skip values of the second list to decide which blocks to decompress and which to skip.

Scoring. Since there can be many documents relevant to a query, search engines use a scoring system to rank candidate documents and output the top-k documents. We follow the Okapi BM25 (Best Match 25) ranking model, which is used by popular search engines and TREC competitions [3, 22]. It computes a relevance score for each candidate document of a query by normalizing the frequency of query terms using the length of the document.

For example, our intersection query with terms q = {business, cameo} appears in N = 3 documents whose identifiers are D = {11, 38, 46}. Then, the BM25 score of each document $d \in D$ can be represented as follows.

$$\text{Score}(d) = \sum_{q_i \in q} \text{IDF}(q_i) \cdot \frac{tf(q_i, d) \cdot (k_1 + 1)}{tf(q_i, d) + k_1 \cdot (1 - b + b \cdot \frac{|d|}{avgdl})}$$

The term frequency $tf(q_i, d)$ counts the number of times a query term q_i appears in document d. The more times a term appears, the higher its contribution to the overall score will be. The term $\frac{|d|}{avgdl}$ measures the length of the document normalized to the average length of all documents in the corpus. The more terms in the document, the lower the contribution of each term to the score will be. The constant k_1 is used to limit the term frequency scaling, and the constant b is used to control the effect of the document length normalization.

$$\text{IDF}(q_i) = \log \frac{N - n(q_i) + 0.5}{n(q_i) + 0.5}$$

Moreover, the inverse document frequency IDF(q_i) measures the rareness of a term across the entire document corpus. It uses the total number of documents that contain the term $n(q_i)$ to penalize the common terms. The rarer terms in the query contribute more to the overall score. For example, if a query contains the terms q = {the, business}, the term "*business*" is more important than the term "*the*", which is likely to appear in almost every document.

3 Inverted Index Scheme for IIU

3.1 Index Structure

Posting List Structure. As explained in Section 2, we build an inverted index which consists of posting lists for all terms. In addition to document identifiers (docIDs) we also include the term frequency in the posting list to perform scoring. For example, Figure 4 shows that the posting list L for a term, say, *Lausanne*, is a list of 2-tuples composed of docID and term frequency. It also shows the delta-compressed posting list L_Δ. Compared to the uncompressed list L, the range of the docID field becomes much smaller. With this compression scheme, we can narrow down the bitwidth of this field.

Bit-packing and Partitioning. Bit-packing is an effective way to save storage space when the range of value is limited. Instead of utilizing the common bitwidths for integer values (e.g., 8, 16, 32 bits), bit-packing utilizes the minimum number of bits that can represent the largest value in the range. However, when a posting list is large, using the maximum bits (required to represent the largest value) for smaller values is inefficient. For example, the list L_Δ for the term *Lausanne* in Figure 4 contains elements that require $3 - 7$ bits for representation, but the entire list needs to be compressed using the maximum bits (=7). To avoid this, our scheme partitions the posting list into multiple blocks, where each block has a disjoint set of contiguous elements. Then, for each block, we inspect the required bitwidths for the delta-compressed docIDs and term frequencies and compress them together on a per-block basis.

Per-Block Metadata. Since we perform bit-packing at a block granularity, we need to store some metadata for each block. Specifically, the metadata includes the (i) number of bits (5 bits) required to encode the docIDs in the block, (ii) number of bits (5 bits) required to encode the term frequencies, (iii) number of elements (11 bits) stored in the block, and (iv) offset to the starting location of the compressed block (43 bits). Thus, 64 bits of metadata are required per block. Our scheme also maintains the raw value of the first docID within each block, called *skip value*, to let the search engine skip a block if the engine does not need to look up any value between its skip value and that of the next block. Figure 5 shows the first three blocks of an example posting list. This block structure also creates opportunities for inter-query parallelism as multiple data blocks can be decompressed simultaneously. The skip value is added to a d-gap to obtain the uncompressed docID.

3.2 Dynamic Block Partitioning

A naive way to partition a list into multiple blocks is to use a fixed block size. While easy to implement, this static partitioning scheme is suboptimal in terms of compression ratio. An outlier value within a block may prevent it from using a narrower bitwidth. On the other hand, a dynamic partitioning scheme uses variable block sizes. It gives the partitioner better chances to group elements of similar bitwidths together, thus reducing the overall storage cost of a posting list. Thus, we adopt a dynamic partitioning scheme for IIU.

Partitioning Scheme. Like previous proposals [19, 23, 24, 25], we also formulate block partitioning as an optimization problem. Let $L_\Delta = \{l_1, l_2, ..., l_n\}$ be a delta-compressed posting list. Each posting l_i is a tuple of the delta-compressed docID and term frequency, denoted by (l_i^{dn}, l_i^{tf}). Suppose $B = \{B_1, B_2, ..., B_k\}$ represents a k-way partitioning over L_Δ so that $L_\Delta = \{B_i | B_i \in B\}$ and $B_i \cap B_j = \emptyset$ for any $i \neq j$. Let $\bar{l}(B_i)$ be the set of all the postings in B_i such that $\bar{l}^{dn}(B_i) = \{l_j^{dn} | l_j \in B_i\}$ and $\bar{l}^{tf}(B_i) = \{l_j^{tf} | l_j \in B_i\}$.

Given a block B_i, the cost function $C(B_i)$ is defined as the storage cost of the block. Each block B_i requires the following information to be stored: (i) 64-bit metadata, (ii) 32-bit skip value, and (iii) bit-packed postings $\bar{l}(B_i)$. Let b_i^{dn} and b_i^{tf} represent the bitwidth of docID and term frequency after bit-packing, respectively. Then, the total cost for L_Δ using the partitioning scheme is the sum of $C(B_i)$ over all $B_i \in B$, which is represented as follows:

$$dn_{max} = max(\bar{l}^{dn}(B_i)) \qquad tf_{max} = max(\bar{l}^{tf}(B_i)) \quad (1)$$

$$b_i^{dn} = \lceil \log(dn_{max} + 1) \rceil \qquad b_i^{tf} = \lceil \log(tf_{max} + 1) \rceil \quad (2)$$

$$C(B_i) = (b_i^{dn} + b_i^{tf}) \cdot |B_i| + 96 \quad (3)$$

Thus, our scheme should find an optimal partition of L_Δ such that the overall storage cost $C(B)$ is minimized, i.e., $min\ C(B) = \sum_{B_i \in B} C(B_i)$ with a constraint that $\{|B_i| < 2^{11}\ \forall\ B_i \in B\}$ as 11 bits are used in the per-block metadata to store the number of elements present in the block. We use a dynamic programming algorithm to calculate the storage overhead of creating a block B_i at different positions i such that the overall cost $C(B)$ is minimized. We use a hyper-parameter *maxSize* to limit the size of each block, $|B_i| < maxSize$.

Note that this partitioning scheme controls space-time tradeoffs. On the one hand, a large block is preferred to minimize the storage overhead. On the other hand, large blocks limit the amount of block-level parallelism, which eventually limits intra-query parallelism. Based on an empirical analysis (shown in Figure 14), we set *maxSize* to 256 to balance the amount of intra-query parallelism and storage overhead.

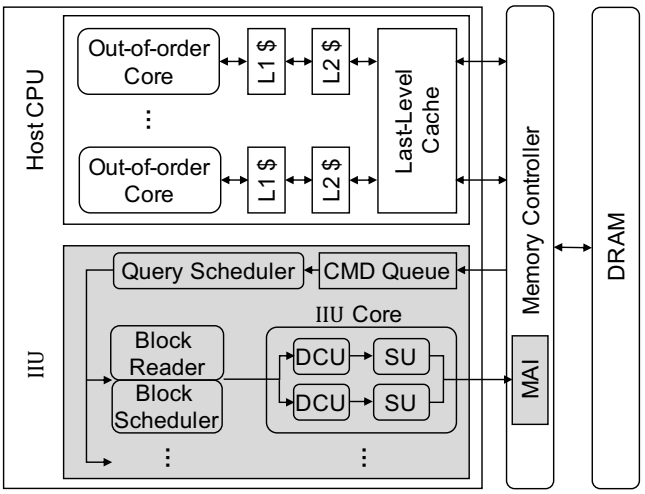

Figure 6. Overview of IIU

4 IIU Hardware Architecture

4.1 Overview

Figure 6 shows an overview of the IIU design. IIU is designed to offload key operations involved in a search query. The offloaded operations include decompression, set operations (intersection/union), and scoring. The host sends search queries to IIU via a command queue. Then IIU processes each query in two phases: (i) *scheduling phase* and (ii) *computation phase*. During the *scheduling phase*, the query scheduler allocates a query to a block reader-scheduler pair. During the *computation phase*, the block reader loads the compressed posting list, while the block scheduler loads per-block metadata and skip values. Each compressed block is allocated to an IIU Core to perform decompression, set operations, and finally scoring.

Host CPU-IIU Interface. IIU provides two primitive functions for the host to communicate with the IIU .

```
void init(file invFile)
```

The `init` function is used to configure an IIU. It sets up a communication pipe between the host and IIU using a memory-mapped register. It also loads the inverted index data structure from file `invFile` to a non-cacheable memory region, which obviates the need for managing cache coherence in IIU.

```
val search(val qtype, addr list0, size_t
    length0, addr list1, size_t length1, addr
    result, val numCores)
```

The `search` function is invoked to send a query to IIU. The type of the query is specified by the `qtype` argument. The start address and size of an input posting list is given by `list0` and `length0`, respectively. In the case of intersection and union queries, optional arguments `list1` and `length1` can be used to provide information about the second posting list. The return values are written to memory at the address

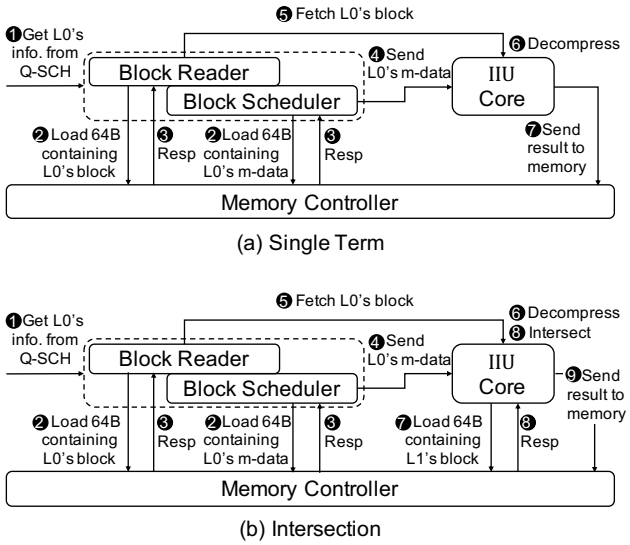

Figure 7. The execution flow of each IIU operation

`result`. This function also provides an argument `numCores` to specify the number of IIU Cores to be assigned to this query. This allows a programmer to specify a priority to the query. For example, multiple cores can be assigned to a single query requiring low latency. More details are discussed in Section 4.4.

Memory Access from IIU. All memory requests from the IIU are handled by the Memory Address Interface (MAI) located at the memory controller. MAI maintains a table of 128 entries to store the requested address with a unique requestor ID. This table plays the same role as a miss status handling register (MSHR) in the host CPU. Then the memory controller issues the pending requests to DRAM. When a response arrives at MAI, it relays the response data to the corresponding requestor using its unique ID. Also, since the host CPU uses the virtual address for accessing the inverted index, duplicate TLB entries are maintained within IIU.

4.2 IIU Query Processing Flow

IIU supports three major query types: single term, intersection, and union. We describe below the execution flow for each query type once a query is assigned to a block reader-scheduler pair and enters the *computation phase*.

Single Term Query. For a single term query, all elements are decompressed from the posting list of the queried term. Figure 7(a) shows the execution flow of decompressing a posting list $L0$. ❶ The query scheduler sends the starting address of the compressed list $L0$ to the block reader and the corresponding address of its metadata to the block scheduler. As described in Section 3.2, the compressed posting list is partitioned into blocks, and per-block metadata maintains their sizes. ❷ The block reader and scheduler request the compressed list and metadata, respectively, and ❸ store them into their internal buffers at a 64-byte granularity. ❹ Then, the block scheduler finds an available IIU Core and schedules the block for decompression. Each IIU Core contains two decompression units (DCU) and two scoring units (SU). The block scheduler assigns a disjoint set of blocks to each DCU. The number of blocks that can be decompressed in parallel is limited by the number of DCUs available to the block scheduler. ❺ The DCU fetches the compressed block from the block reader buffer. ❻ It decompresses all docIDs and their term frequency (tf). Then, the scoring unit (SU) calculates their BM25 score and ❼ writes the result to memory.

Intersection Query. An intersection query retrieves the common documents between two input posting lists. Figure 7(b) shows the execution flow of an intersection operation between two posting lists $L0$ and $L1$ ($|L0| < |L1|$). We adopt the SvS intersection scheme, described in Section 2.2. Since this scheme uses binary search on the longer list to reduce the number of decompressed blocks, IIU performs membership testing using $L1$ skip list. It identifies which blocks in $L1$ may contain elements from $L0$ and decompresses only those blocks. Similar to the single-term query ❶ - ❺, IIU loads and schedules the blocks in $L0$ for decompression using an available IIU Core. An IIU Core contains two DCUs, but the block scheduler uses only one of them (DCU0) for scheduling $L0$ blocks. ❻ After DCU0 decompresses a docID in $L0$, the binary search unit (BSU) inside the IIU Core is used to find a candidate $L1$ block. ❼ DCU1 loads the candidate $L1$ block from memory and decompresses it. Once both DCUs have decompressed docIDs, ❽ the IIU Core performs intersection by discarding the smaller of the two docIDs at the head of each DCU. If both docIDs are matched, they are sent to the two scoring units (SUs), and their scores are added. Since multiple $L0$ docIDs can fall in the same $L1$ block during the binary search, the IIU Core should process all $L0$ elements before fetching a new $L1$ block. ❾ Finally, the result of the intersection is written to memory.

Union Query. A union query finds the set of documents that are present in at least one of the two input posting lists. Like the other types, IIU uses the block reader and scheduler to load and schedule an input list to an IIU Core. But, instead of loading a single list, the block reader and scheduler load both lists simultaneously. The two DCUs and SUs in an IIU Core are utilized to decompress and score each list in parallel. Similarly to a 2-way merge sort, the IIU Core performs union such that the smaller docIDs are written to the memory first. Matched docIDs are identified, and their scores are merged by addition. Then the final score is written to memory. Once a DCU finishes processing the entire list first, the remaining postings from the other DCU are flushed to memory.

4.3 IIU Hardware Building Blocks

In this section, we describe the IIU hardware modules in detail. In particular, we divide them into two major parts: (i)

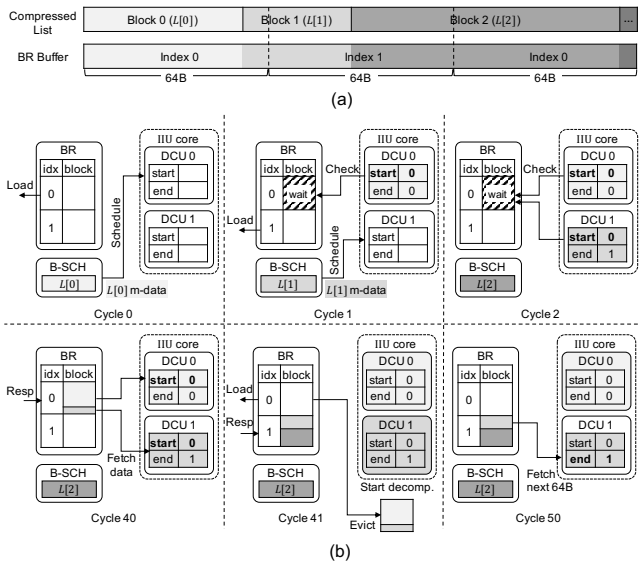

Figure 8. (a) Allocation of BR buffers to compressed list L; (b) Illustration of BR and B-SCH operations

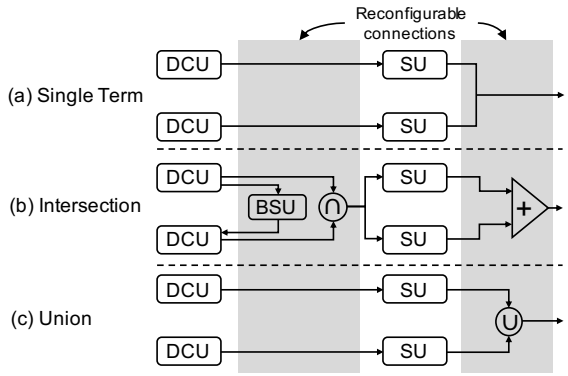

Figure 9. Datapath setup based on the query type

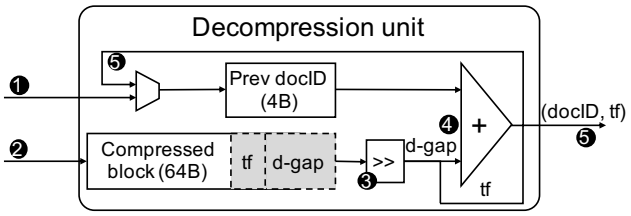

Figure 10. Decompression Unit (DCU) structure

block reader and block scheduler at the frontend, and (ii) IIU Core and its sub-modules at the backend.

Block Reader (BR) and Block Scheduler (B-SCH). The BR module reads the compressed posting list for a query term to store it in local stream buffers. Likewise, the B-SCH module reads the corresponding per-block metadata and skip list into its local buffers. Once the metadata is fetched, it finds an available IIU Core and schedules the corresponding block for computation. If there are no free IIU Cores and the B-SCH buffer is full, future reads are stalled. An allocated IIU Core, in turn, fetches the compressed block data from the BR buffer. Since a single buffer entry may contain compressed postings for multiple blocks, the BR maintains a fetch counter per buffer entry to track which block data is actually consumed. Once the entire entry is fetched by IIU Cores, it is cleared, and the next 64-byte chunk is eagerly prefetched from memory.

Running Example. Figure 8 illustrates a running example of how BR and B-SCH modules fetch compressed data and schedule blocks for computation. During the first two cycles, B-SCH fetches the metadata for Block 0 ($L[0]$) and 1 ($L[1]$) and schedules them to DCU 0 and 1, respectively. At the same time, BR sends a request to fetch the compressed block data. In this example, BR has two stream buffer entries with 64 bytes per each. DCUs calculate the start and end index of the BR buffer containing the block data they are processing. At Cycle 1 and 2, both DCUs send a request to fetch data from the first BR buffer entry (index 0) and waits for its response. BR receives the data for the first buffer entry after 40 cycles and then services the pending DCU requests. At Cycle 41, since all the data in the first entry are read by the DCUs, the entry is evicted, and BR sends a new block prefetch request to memory. Meanwhile, it receives a response for the second buffer entry, which contains the remaining data of Block 1 and the first part of Block 2. At Cycle 50, DCU 1 finishes computation on the first part of Block 1 data read from the BR buffer index 0 and fetches the remaining data from the next buffer entry (index 1). This process continues for all the remaining blocks.

IIU Core: Overview. The IIU Core supports the three query types: single term, intersection, and union. Figure 9 shows a simplified datapath of IIU Core showing three major components: (i) decompression unit (DCU), which decompresses d-gap-term frequency pairs (d-gap, tf) in the block; (ii) scoring unit (SU), which calculates the BM25 score for each decompressed (docID, tf) pair; and (iii) binary search unit (BSU) which finds a candidate block for fast intersection operation. Apart from these components, IIU Core employs reconfigurable connections (i.e., MUXes) to set up the datapath depending on the query type. In what follows, we describe each component in detail.

IIU Core: Decompression Unit (DCU). Figure 10 shows the structure of a DCU. ❶ The DCU first receives the skip value and metadata (containing bitwidth of d-gap and tf) of the scheduled block from the block scheduler. ❷ It fetches data from the block reader at a 64-byte granularity and stores it in a local buffer. The stored data is a list of (d-gap, tf) pairs, which are bitpacked together as described in Section 3. ❸ A (d-gap, tf) pair is extracted and decoded sequentially every cycle using shifting and bit-masking based on their bitwidth. ❹ DCU adds the previous (raw) docID to the newly

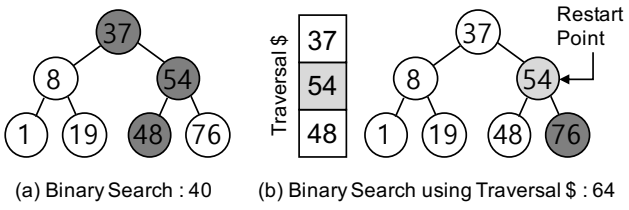

Figure 11. Binary search example using traversal cache

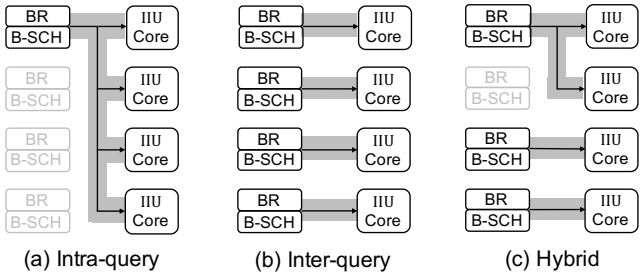

Figure 12. Intra-query, inter-query, and hybrid parallelism in IIU

decoded d-gap, and zero-extends tf to 32 bits. ❺ The final uncompressed (docID, tf) pair is sent to the next stage in the IIU Core pipeline, and the previous docID gets updated.

IIU Core: Scoring Unit (SU). The SU calculates the BM25 score for each input (docID, tf) pair. The BM25 scoring function (in Section 2.2) has multiple arithmetic operations. To reduce the strength of computation, IIU pre-computes a few sub-expressions at index time, which are reused at query time. For each possible query term q_t, we pre-calculate its inverse document frequency $idf(q_t)$ and store $\overline{idf}(q_t) = idf(q_t) * (k_1 + 1)$ in metadata. Also, for each document d, we pre-calculate its relative length and store $\overline{dl}(d) = k_1 * (1 - b + b * \frac{|d|}{avgdl})$ as well. At query time SU loads $\overline{idf}(q_t)$ at the beginning of query processing and performs a memory read for each per-document constant $\overline{dl}(docID)$ as it receives (docID, tf) pairs. It uses a single adder and a pipelined fixed-point divider to compute a partial score $\overline{s} = \frac{1}{tf+\overline{dl}(docID)}$. Finally, SU computes the final BM25 score, $s = \overline{idf}(q_t) \cdot \overline{s} \cdot tf$, and sends the (docID, s) pair to the next stage in the IIU Core pipeline. SU takes 18 cycles to output a (docID, s) pair, but it is fully-pipelined; i.e., 18 inputs can be simultaneously in flight for processing.

IIU Core: Binary Search Unit (BSU). An IIU Core uses BSU during an intersection query. It identifies the candidate blocks in a longer posting list, which may contain docIDs from a shorter posting list. To this end, BSU performs a binary search over the *skip list* of the longer list (not over postings inside a block), which is fetched from memory. However, we find there is a substantial amount of reuse (locality) in skip list traversal as we always search for docIDs in sorted order. Thus, we propose to use a small *traversal cache* with 32 entries to reduce memory traffic. The traversal cache stores the skip values on the traversal path from the previous search. In this way, we can go without accessing memory for the common prefix of the search with the last traversal.

Figure 11 shows an example of how the skip list is traversed where the skip values are {1, 8, 19, 37, 48, 54, 76} and the docIDs being searched are 40 and then 64. It traverses the skip list as follows: (1) For docID 40, BSU performs a full binary search on the skip list to find the candidate block whose skip value is 37. As a result, the traversal path (37 → 54 → 48) is stored in the traversal cache. (2) For docID 64, BSU first uses the traversal cache to perform traversal until the path diverges. It identifies a node with skip value 54 is the point of the restart. From there, BSU fetches the skip values from memory and finishes the traversal. (3) The block with skip value 54 becomes the candidate block as docID 64 is smaller than the next skip value 76. (4) BSU updates the traversal cache with the new path by replacing the skip value 48 with 76.

4.4 Query Parallelism

IIU can flexibly allocate IIU Cores to improve query latency or throughput to serve a variety of deployment scenarios. This is realized by supporting both intra- and inter-query parallelism in IIU. Each IIU consists of multiple BR and B-SCH pairs as well as IIU Cores. It also includes reconfigurable interconnects between them. During the *scheduling phase*, the query scheduler configures this interconnects based on a QoS requirement from the user.

Figure 12 shows three different configurations where (a) has minimum query latency by allocating all the available IIU Cores for a single query, hence maximizing intra-query parallelism; (b) has maximum throughput by using all the available BR and B-SCH pairs with each one connected to a single IIU Core, hence maximizing inter-query parallelism; and (c) takes a hybrid approach by using multiple BR and B-SCH pairs and IIU Cores to run both low-latency and high-throughput queries together.

4.5 Handling Complex Queries and Top-*k* Selection

Complex Queries. IIU can process complex queries with multiple terms and set operators like $(L0 \cup L1) \cap (L2 \cup L3)$. These queries are represented using a binary expression tree, where the leaf nodes are query terms, and non-leaf nodes are set operations. IIU can recursively evaluate the left and right subtrees and then apply the root set operation to the output. Inter-query parallelism can be exploited when evaluating multiple subtrees in parallel. Since the evaluation of a subtree requires operations on an uncompressed list, IIU can be trivially extended to accelerate only set operations while bypassing the DCU.

```
1 vector<docID> top-k(vector<pair<docID, score>>
    candidate, int k)
2   MinHeap<key=docID, value=score> pq
3   for curr in candidate
4     if (pq.size() < k)
5       pq.push(curr)
6     else if (pq.top().value < curr.score)
7       pq.pop()
8       pq.push(curr)
9   return pq.keys()
```

Figure 13. Pseudocode of top-k Selection

Top-k Selection. The last step in query processing is using the computed scores to retrieve the documents with top-k scores. Though IIU offloads scoring from the host CPU, we run the top-k selection process on it. Figure 13 shows the pseudocode of our top-k selection algorithm. The host CPU maintains a priority queue of size k and reads the (docID, score) pairs written by IIU to memory and updates the priority queue.

5 Evaluation

5.1 Methodology

Evaluation Model. We evaluate IIU using a custom cycle-level simulator integrated with DRAMSim2 [26]. Table 1 summarizes the system parameters we use to evaluate Lucene and IIU. Lucene is a search engine library powering some of the most popular enterprise search engines such as Elasticsearch [6] and Apache Solr [5], representing the state-of-the-art. We also implement IIU modules in RTL using Chisel3 [27], which have been extensively verified using realistic test cases. We synthesize IIU RTL using Synopsys Design Compiler with TSMC 40nm standard cell library for area and power estimation. We also use ARM Artisan memory compiler to generate buffer/queue structures. We use FastPFor [28] C++ library, which contains multiple integer compression schemes to compare the performance of our compression scheme with other existing schemes.

Workloads. We evaluate IIU performance using two real web datasets. One is CC-News [29], news articles that have been crawled from news sites around the world through CommonCrawl. This data is publicly available, and we only parse English articles among crawled news from January through December 2017 by following the methodology [30]. The second dataset, ClueWeb12 [7], is provided by CMU for research purposes, crawled by Hetrix web crawler. Pisa library [31] is used to turn ClueWeb12 into inverted index form. CC-News contains 29,948,077 documents and 84,940,183 unique terms, whereas ClueWeb12 contains 52,343,021 documents and 133,248,235 unique terms. As for query, we uniformly sample 100 single-term (for single term) and double-term (for intersection and union) queries from TREC 2006 Terabyte Track [32] with only those terms present in each dataset.

Host Processor	
Core	Intel(R) Core(TM) i7-7820X CPU @ 3.60GHz
L1 $	32KB I-cache, 32KB D-cache
L2 $	1MB (Private, Unified)
L3 $	11MB (Shared, Unified)
DDR4 Memory System	
Organization	DDR4-2400, 4 channels, 128GB
Timing	tCK = 0.833ns, tRAS = 32ns, tRCD = 14.16ns, tCAS = 14.16ns, tWR = 15ns, tRP = 14.16ns
Bandwidth	76.8 GB/s (19.2 GB/s per channel)
IIU Configuration	
IIU	8 (Block Reader, Block Scheduler), 8 IIU Core @ 1.0GHz
IIU Core	2 Decompression unit, 2 Scoring unit, 1 BinarySearch unit

Table 1. Architectural parameters for evaluation

Dataset	Lucene	Pfor	OptPfor	SIMDPfor	VByte	**IIU**
CC-News	7.58×	4.41×	18.57×	5.69×	3.99×	**13.39**×
ClueWeb12	3.77×	4.40×	7.91×	5.08×	3.99×	**5.54**×

Table 2. Compression ratio (higher is better)

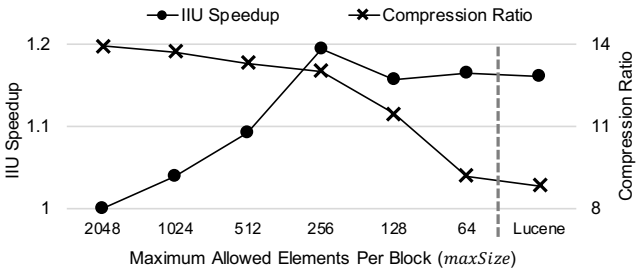

Figure 14. Speedup vs. compression ratio tradeoff

5.2 Compression Efficiency

Table 2 summarizes the compression ratio, i.e., the size of an uncompressed list over the size of a compressed list, for our scheme, Lucene, and other existing compression schemes. Compared to Lucene, which is our baseline, IIU reduces the compression ratio (posting) by 47% (ClueWeb12) and 76% (CC-News). These savings are mainly attributed to the dynamic partitioning scheme of IIU outperforming the static partitioning scheme of Lucene with a fixed block size of 128 elements. Moreover, the size of the metadata block is smaller in IIU than Lucene, which maintains additional per-block metadata to accelerate query processing. Compared to other schemes, IIU achieves a higher compression ratio except for OptPfor. One thing to note is that Pfor and its variants are not suitable for compressing unsorted data, they require a separate scheme for compressing term frequency, which is not sorted.

Figure 14 quantifies time-space tradeoffs in selecting the *maxSize* parameter used for the partitioning scheme in IIU. This parameter puts a limit on the maximum number of

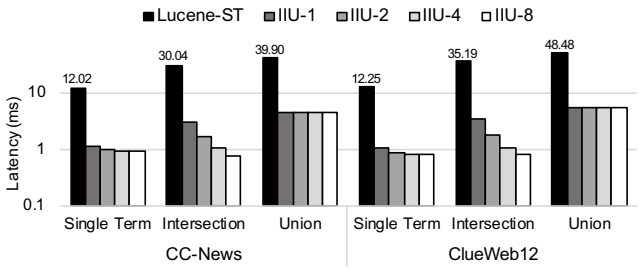

Figure 15. Query latency (in log scale)

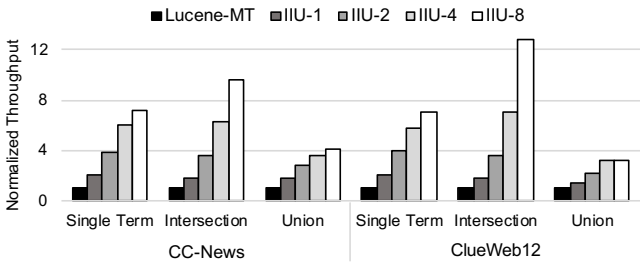

Figure 16. Query throughput for 100 queries

postings within a block. We overlay IIU speedups when processing 100 queries with the compression ratio for a range of *maxSize* values. For this experiment, we exploit intra-query parallelism to process one query at a time. Generally, the speedup increases and the compression ratio decreases as *maxSize* decreases. The increase in speedup is due to a higher degree of intra-query parallelism in IIU, whereas the decrease in the compression ratio is due to an increase in block count, which in turn increases the metadata size. We determine that 256 is the optimal value for *maxSize* as there is no increase in speedup for a smaller value, and a decrease in compression ratio is negligible (<1%) compared to the best case (*maxSize*=2048). Furthermore, while shown in the figure, we evaluate IIU using Lucene's static compression scheme to see a comparable speedup number but a 46% lower compression ratio than our scheme.

5.3 Performance Results

Query Latency. Figure 15 compares the normalized average latency of IIU and Lucene on a low-load system. Thus, we assume IIU is configured to optimize query latency by exploiting intra-query parallelism (as in Figure 12(a)). Note that Lucene does not exploit any intra-query parallelism and thus utilizes only one core to process a single query. We evaluate IIU with a varying number of IIU Cores, where IIU-*X* represents *X* IIU Cores connected to a single BR and B-SCH pair. The results demonstrate that IIU has much lower latency than Lucene by effectively exploiting intra-query parallelism. IIU-8 achieves over 13.1× latency reduction on single term queries, over 40.4× reduction on intersection, and around

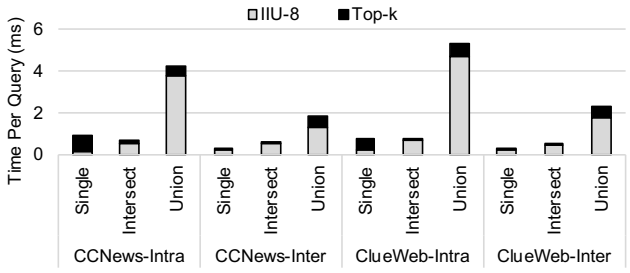

Figure 17. Performance breakdown of IIU-8

10× reduction on union. Single term queries do not observe a noticeable extra speedup with additional IIU Cores because the top-*k* selection time (performed on CPU) becomes the more dominant portion of the runtime. For union queries, IIU shows the same latency regardless of the number of IIU Cores allocated as the merge unit becomes the bottleneck beyond a single IIU Core.

Query Throughput. Figure 16 compares the normalized average throughput of IIU and Lucene. For this experiment, IIU is configured to exploit inter-query parallelism (as in Figure 12(b)), and Lucene also utilizes multiple CPU cores to process the queries in parallel. We evaluate IIU with a varying number of IIU Cores, where IIU-*X* represents the number of BR and B-SCH pairs and the same number of IIU Cores in a 1-1 mapping. In the IIU-8 configuration, the average speedup of IIU over Lucene is 7.1× on single terms, 11.3× on intersection, and 3.7× on union. As we keep increasing the number of IIU Cores, the speedup is eventually limited by the available memory bandwidth. Among the three types of queries, intersection queries demonstrate the best speedup as they benefit from the specialized intersection and decompression hardware. Union queries show relatively low speedups (i.e., a half of decompression speedups) as the merge unit, which merges two decompressed blocks from two DCUs, becomes the pipeline bottleneck. Overall, IIU achieves robust performance over the two datasets used for evaluation.

The throughput gains are attributed to two major sources: specialization and parallelism. For query throughput IIU-1 achieves a 14.6× speedup over the single-threaded Lucene (not shown in the Figure 16), which quantifies the benefits of specialization. Utilizing eight IIU cores achieves an extra 3.6× speedup on top of it, quantifying the benefits of parallelism. In Figure 14 the IIU compression scheme gives only a 1.2× speedup over Lucene; however, it saves a substantial amount of memory space for a higher compression ratio.

Runtime Breakdown. As stated in Section 4.1, the last step of query processing, top-*k* selection, is performed on the host CPU. Thus, when intra-query parallelism is exploited, the portion of the top-*k* selection operation gets larger due to Amdahl's Law as we increase the IIU Core count. Figure 17 shows the portion of top-*k* selection for IIU-8. For single term queries the query latency is already dominated by the

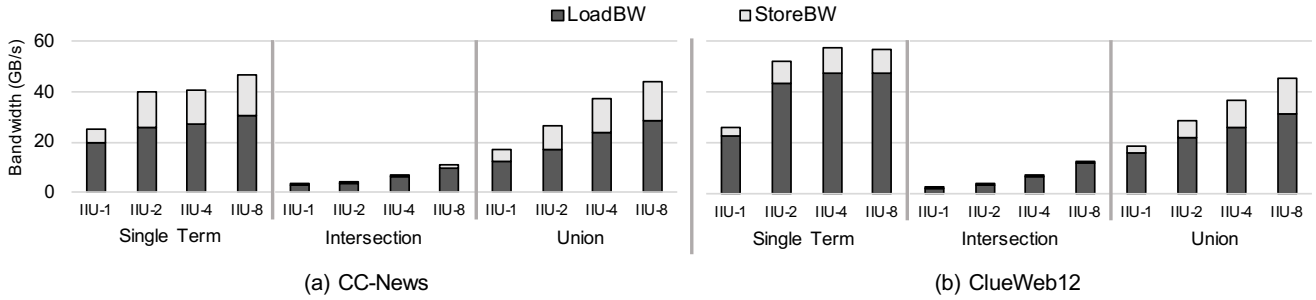

Figure 18. Bandwidth utilization of IIU-8 with two datasets

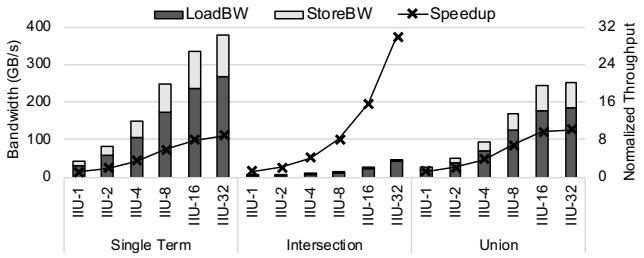

Figure 19. Scalability analysis of IIU on high bandwidth memory

Component	Area (mm^2)	Power (mW)	# of component	Total Area (mm^2)	Total Power (mW)
Block Reader	0.020	13.9	8	0.160	111.7
Block Scheduler	0.017	11.0	8	0.143	88.3
IIU Core	0.335	115.6	8	2.687	925.4
Command Queue	0.004	2.7	1	0.004	2.7
Query Scheduler	0.009	6.4	1	0.009	6.4
MAI	0.101	9.6	1	0.101	9.6
Total Area : 3.106mm^2 / Average Power : 1.144W					

Table 3. Total area/power usage of IIU

top-k selection operation; for intersection queries, there is some room for further scaling down the portion of IIU with additional cores, but query time will be eventually limited by the top-k operation as well.

Still, inter-query parallelism is free from this limitation, and IIU can achieve scalable throughput to the number of IIU Cores. In this setup, memory bandwidth will eventually become the limiting factor of throughput scaling. Figure 18 shows the bandwidth utilization of IIU for both datasets when it is configured to exploit inter-query parallelism. Except for the intersection query, the performance of IIU becomes memory bandwidth-bound as the number of IIU Cores increases.

Scalability. The main limiting factor of scaling throughput is memory bandwidth. Fortunately, emerging 3D stacked DRAM technologies [33] offer much higher bandwidth (with higher latency) than the conventional DDR4 DRAM system. Figure 19 presents the bandwidth utilization of IIU on CC-News with inter-query parallelism assuming an HBM-like memory system with higher memory bandwidth. Intersection queries do not fully utilize bandwidth as the result set often becomes small after the intersection. However, for single term and union queries, the bandwidth usage scales as we increase the number of IIU Cores. In turn, this leads to a significantly higher speedup than on the original DRAM system. Increased DRAM bandwidth can also benefit CPUs; however, it is much more difficult to saturate memory bandwidth with CPUs as they have much lower throughput density in terms of area and power than custom accelerators.

5.4 Area, Power and Energy Results

Area. Table 3 shows the area breakdown with individual components of IIU. Among them, the IIU Core accounts for the dominating portion with its many processing units and buffers. A single IIU Core occupies a $0.34mm^2$ of area, and the total area amounts to $2.69mm^2$ with 8 IIU Cores. Following the IIU Cores, block readers and block schedulers take $0.30mm^2$ area, and system-wide components (e.g., MAI, Command Queues) $0.11mm^2$ of area, thus making the total area of 8-core IIU $3.11mm^2$.

Power. Table 3 also shows an average power breakdown of IIU. Compared to the CPU platform used for evaluation, whose TDP is 140W, IIU consumes 122.4× less power. This large power gap is attributed to the flexibility-efficiency tradeoff. Operations on the inverted index are composed of relatively simple integer instructions for which general-purpose CPUs are not cost-effective. Instead, IIU benefits greatly from specialization and achieves both performance speedups and power savings at the same time.

Energy. Figure 20 shows normalized energy consumption that Lucene and IIU spend on processing a query. As shown in the figure, IIU itself consumes very little energy, which is not surprising as its power consumption is orders of magnitude smaller than that of CPUs. However, since IIU still performs top-k selection on CPUs, the total amount of energy spent is dominated by CPUs for IIU. Even with this consideration, IIU still improves the energy efficiency of full-text search by 18.6× on average across all cases.

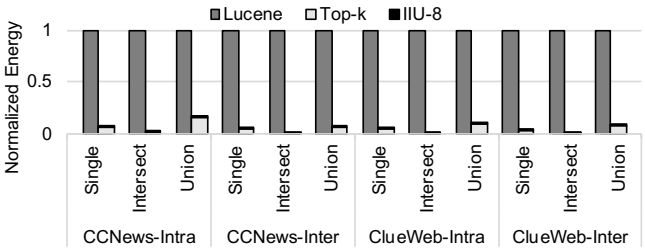

Figure 20. Normalized energy consumption

6 Related Work

Inverted Index Compression. Inverted index has been used over decades, and many compression schemes have been proposed for the inverted index [14, 16, 17, 18, 19, 34]. VByte [14, 34] is a classic byte-aligned compression scheme, which features a simple structure and fast sequential decoding speed. PforDelta [16] is a well-known compression scheme that compresses most of the values in a block and maintains a separate array for exception values that cannot be compressed with a smaller bitwidth. There are many variants of this scheme such as OptPforDelta [17], NewPforDelta [17], SIMDPforDelta [18] to reduce its space overhead and exploit modern SIMD instructions. We choose bitpacking compression [18] for our hardware accelerator as it requires no additional data structure (e.g., an exception list), which is complex to implement efficiently in hardware.

Also, there are several proposals for optimizing the partitioning scheme to reduce the space overhead [19, 23, 35, 36]. They employ various methods such as dynamic programming and approximation to find an optimal partition to minimize space overhead. However, these partitioning schemes are better suited for CPU architectures with multilevel caches. In contrast, our partitioning scheme is specialized for IIU hardware. Also, it flexibly controls tradeoffs between the compression ratio and the amount of block-level parallelism to balance storage requirement and performance.

Software Optimization for Inverted Index Operations. There are various software works for accelerating operations on inverted index [19, 20, 37, 38, 39, 40]. MILC [19] is a novel compression scheme that is highly optimized for fast membership testing and reducing space overhead of inverted lists. They leverage CPU features and cache hierarchy for SIMD acceleration and cache-aware compression. Even though they get impressive performance improvement using these optimizations, they still cannot fully exploit memory-level parallelism because of the limited size of load/store queues and instruction window in general-purpose CPUs. Culpepper et al. [20] propose an efficient intersection scheme and a simple additional data structure for supporting fast intersection. Lemire et al. [37] propose a SIMD galloping algorithm for fast intersection. Zhang et al. [38] propose direct document analytics on compressed data to reduce the space overhead and processing time. However, these works handle only part of key operations in a search process.

Also, there are proposals to exploit the high degree of parallelism on GPUs [11, 12, 13, 41, 42, 43, 44]. Griffin [11] exploits fine-grained scheduling between CPU and GPU depending on the characteristic of each query. They propose a parallel decompression and intersection scheme for query processing. GENIE [41] is a generic inverted index framework on GPU, which supports parallel similarity search. Wang et al. [42] propose an efficient GPU caching method to improve list intersection and top-k ranking. Ao et al. [44] propose a novel technique for optimized parallel intersection and efficient decompression on GPUs. However, all these works focus on improving either query latency or throughput, but not both. Instead, IIU flexibly exploits both intra- and inter-query parallelism depending on deployment scenarios, so it can increase the query throughput and reduce the latency.

Hardware support for inverted index. There are proposals for hardware support for inverted index [45, 46, 47]. Gunther et al. [45] propose custom hardware for supporting text search and scoring documents for relevance. Yan et al. [46] propose an index serving accelerator with a new compression scheme combining Huffman coding and VByte [14, 34]. Pinaka [47] is FPGA-based custom hardware for supporting essential operations of web search engines such as decompression, boolean operations, and ranking, which shares the same spirit with IIU. However, all these hardware systems cannot fully utilize the abundant DRAM memory bandwidth due to limited parallelism and/or do not flexibly support both intra- and inter-query parallelism.

7 Conclusion

This paper presents IIU to accelerate query processing on an inverted index with low storage overhead. To achieve both low latency and high throughput, we co-design an indexing scheme and a hardware accelerator. For the indexing scheme, we propose a hardware-friendly dynamic partitioning scheme to achieve high compression ratio. IIU is designed to fully exploit memory-level parallelism for decompression, set operations, and scoring. Our evaluation with a cycle-level simulator with two real datasets demonstrates an average of 13.8× latency reduction, 5.4× throughput increase, and 18.6× reduction in energy consumption over Apache Lucene, the state-of-the-art full-text search framework.

Acknowledgments

This paper was the result of a research project supported by SK Hynix. It was also supported in part by a National Research Foundation of Korea (NRF) grant funded by Korea Government (MSIT) (NRF-2016M3C4A7952587), an Institute for Information & communications Technology Promotion (IITP) grant (2014-0-00035, Research on High Performance and Scalable Manycore Operating System), and IDEC (EDA tools). Jae W. Lee is the corresponding author.

References

[1] W. B. Croft, D. Metzler, and T. Strohman, "Search engines: Information retrieval in practice.," 2010.

[2] "Okapi bm25." https://en.wikipedia.org/wiki/Okapi_BM25.

[3] S. Robertson and H. Zaragoza, "The probabilistic relevance framework: Bm25 and beyond," *Foundations and Trends in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009.

[4] "Apache lucene." https://lucene.apache.org/.

[5] "Apache solr." https://lucene.apache.org/solr/.

[6] "Elasticsearch." https://www.elastic.co/.

[7] "The clueweb12 dataset." https://lemurproject.org/clueweb12/.

[8] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin, "Earlybird: Real-time search at twitter," in *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, pp. 1360–1369, 2012.

[9] "Apache solr wiki." https://cwiki.apache.org/confluence/display/solr/PublicServers#PublicServers-PublicWebsitesusingSolr.

[10] "Intel VTune Amplifier." https://software.intel.com/en-us/vtune.

[11] Y. Liu, J. Wang, and S. Swanson, "Griffin: Uniting cpu and gpu in information retrieval systems for intra-query parallelism," *ACM SIGPLAN Notices*, vol. 53, no. 1, pp. 327–337, 2018.

[12] S. Ding, J. He, H. Yan, and T. Suel, "Using graphics processors for high performance ir query processing," in *Proceedings of the 18th International Conference on World Wide Web*, pp. 421–430, 2009.

[13] D. Wu, F. Zhang, N. Ao, F. Wang, X. Liu, and G. Wang, "A batched gpu algorithm for set intersection," in *Proceedings of the 10th International Symposium on Pervasive Systems, Algorithms, and Networks*, 2009.

[14] D. Cutting and J. Pedersen, "Optimizations for dynamic inverted index maintenance," in *Proceedings of the 13th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 405–411, 1989.

[15] S. Vigna, "Quasi-succinct indices," in *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, 2013.

[16] M. Zukowski, S. Heman, N. Nes, and P. Boncz, "Super-scalar ram-cpu cache compression," in *Proceedings of the 22nd International Conference on Data Engineering*, pp. 59–59, 2006.

[17] H. Yan, S. Ding, and T. Suel, "Inverted index compression and query processing with optimized document ordering," in *Proceedings of the 18th International Conference on World Wide Web*, pp. 401–410, 2009.

[18] D. Lemire and L. Boytsov, "Decoding billions of integers per second through vectorization," *Software: Practice and Experience*, vol. 45, no. 1, pp. 1–29, 2015.

[19] J. Wang, C. Lin, R. He, M. Chae, Y. Papakonstantinou, and S. Swanson, "Milc: Inverted list compression in memory," *Proceedings of the VLDB Endowment*, vol. 10, no. 8, pp. 853–864, 2017.

[20] J. S. Culpepper and A. Moffat, "Efficient set intersection for inverted indexing," *ACM Transactions on Information Systems*, vol. 29, no. 1, pp. 1–25, 2010.

[21] J. Wang, D. Park, Y.-S. Kee, Y. Papakonstantinou, and S. Swanson, "Ssd in-storage computing for list intersection," in *Proceedings of the 12th International Workshop on Data Management on New Hardware*, pp. 1–7, 2016.

[22] S. E. Robertson, S. Walker, M. Beaulieu, and P. Willett, "Okapi at trec-7: Automatic ad hoc, filtering, vlc and interactive track," *Nist Special Publication SP*, no. 500, pp. 253–264, 1999.

[23] F. Silvestri and R. Venturini, "Vsencoding: efficient coding and fast decoding of integer lists via dynamic programming," in *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, pp. 1219–1228, 2010.

[24] P. Ferragina, I. Nitto, and R. Venturini, "On optimally partitioning a text to improve its compression," *Algorithmica*, vol. 61, no. 1, 2011.

[25] A. L. Buchsbaum, G. S. Fowler, and R. Giancarlo, "Improving table compression with combinatorial optimization," *Journal of the ACM*, vol. 50, no. 6, pp. 825–851, 2003.

[26] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, 2011.

[27] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," in *Design Automation Conference*, 2012.

[28] "The fastpfor C++ library: Fast integer compression." https://github.com/lemire/FastPFor.

[29] "Common crawl - ccnews dataset." http://commoncrawl.org/2016/10/news-dataset-available/.

[30] M. Petri and A. Moffat, "Compact inverted index storage using general-purpose compression libraries," *Software: Practice and Experience*, vol. 48, no. 4, pp. 974–982, 2018.

[31] A. Mallia, M. Siedlaczek, J. Mackenzie, and T. Suel, "PISA: Performant indexes and search for academia," *Proceedings of the Open-Source IR Replicability Challenge (OSIRRC) co-located at SIGIR*, pp. 50–56, 2019.

[32] "Text retrieval conference (trec)." https://trec.nist.gov/.

[33] JEDEC, *JEDEC Standard JESD235A: High Bandwidth Memory (HBM) DRAM*. JEDEC Solid State Technology Association, 2015.

[34] L. H. Thiel and H. Heaps, "Program design for retrospective searches on large data bases," *Information Storage and Retrieval*, vol. 8, no. 1, pp. 1–20, 1972.

[35] G. Ottaviano and R. Venturini, "Partitioned elias-fano indexes," in *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, pp. 273–282, 2014.

[36] G. E. Pibiri and R. Venturini, "Variable-byte encoding is now space-efficient too," *CoRR*, vol. abs/1804.10949, 2018.

[37] D. Lemire, L. Boytsov, and N. Kurz, "Simd compression and the intersection of sorted integers," *Software: Practice and Experience*, vol. 46, no. 6, pp. 723–749, 2016.

[38] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and W. Chen, "Efficient document analytics on compressed data: Method, challenges, algorithms, insights," *Proceedings of the VLDB Endowment*, vol. 11, no. 11, 2018.

[39] I. Rae, A. Halverson, and J. F. Naughton, "In-rdbms inverted indexes revisited," in *Proceedings of the IEEE 30th International Conference on Data Engineering*, pp. 352–363, 2014.

[40] S. Shah and A. Shaikh, "Hash based optimization for faster access to inverted index," in *Proceedings of the 2016 International Conference on Inventive Computation Technologies*, vol. 1, pp. 1–5, 2016.

[41] J. Zhou, Q. Guo, H. V. Jagadish, L. Krcal, S. Liu, W. Luan, A. K. H. Tung, Y. Yang, and Y. Zheng, "A generic inverted index framework for similarity search on the gpu," in *Proceedings of the IEEE 34th International Conference on Data Engineering*, pp. 893–904, 2018.

[42] D. Wang, W. yu, R. J. Stones, J. Ren, G. Wang, X. Liu, and M. Ren, "Efficient gpu-based query processing with pruned list caching in search engines," in *Proceedings of the IEEE 23rd International Conference on Parallel and Distributed Systems*, pp. 215–224, 2017.

[43] D. Wu, F. Zhang, N. Ao, G. Wang, X. Liu, and Jing Liu, "Efficient lists intersection by cpu-gpu cooperative computing," in *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, pp. 1–8, 2010.

[44] N. Ao, F. Zhang, D. Wu, D. S. Stones, G. Wang, X. Liu, J. Liu, and S. Lin, "Efficient parallel lists intersection and index compression algorithms using graphics processing units," *Proceedings of the VLDB Endowment*, vol. 4, no. 8, pp. 470–481, 2011.

[45] Gunther, Milne, and Narasimhan, "Assessing document relevance with run-time reconfigurable machines," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 10–17, 1996.

[46] J. Yan, N. Xu, Z. Xia, R. Luo, and F. Hsu, "A compression method for inverted index and its fpga-based decompression solution," in *Proceedings of the 2010 International Conference on Field-Programmable Technology*, pp. 261–264, 2010.

[47] J. Yan, Z. Zhao, N. Xu, X. Jin, L. Zhang, and F. Hsu, "Efficient query processing for web search engine with fpgas," in *Proceedings of the IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pp. 97–100, 2012.