# Efficient Parallel Lists Intersection and Index Compression Algorithms using Graphics Processing Units

Naiyong Ao, Fan Zhang*, Di Wu
Nankai-Baidu Joint Lab
Nankai University

Douglas S. Stones
School of Mathematical Sciences and Clayton School of Information Technology
Monash University

Gang Wang*, Xiaoguang Liu*, Jing Liu, Sheng Lin
Nankai-Baidu Joint Lab
Nankai University

## ABSTRACT

Major web search engines answer thousands of queries per second requesting information about billions of web pages. The data sizes and query loads are growing at an exponential rate. To manage the heavy workload, we consider techniques for utilizing a Graphics Processing Unit (GPU). We investigate new approaches to improve two important operations of search engines – lists intersection and index compression.

For lists intersection, we develop techniques for efficient implementation of the binary search algorithm for parallel computation. We inspect some representative real-world datasets and find that a sufficiently long inverted list has an overall linear rate of increase. Based on this observation, we propose Linear Regression and Hash Segmentation techniques for contracting the search range. For index compression, the traditional d-gap based compression schemata are not well-suited for parallel computation, so we propose a Linear Regression Compression schema which has an inherent parallel structure. We further discuss how to efficiently intersect the compressed lists on a GPU. Our experimental results show significant improvements in the query processing throughput on several datasets.

## 1. INTRODUCTION

Current large-scale search engines answer thousands of queries per second based on information distributed on billions of webpages, requiring efficient management of terabytes of data. Index decompression and lists intersection are two time-consuming operations used to process a query [3, 23, 25]. In this paper we focus on improving the efficiency of these search engine algorithms and, in particular, we focus on optimizing these two operations for modern Graphics Processing Units (GPUs).

*Email: {zhangfan555, wgzwpzy, liuxguang}@gmail.com


*Proceedings of the VLDB Endowment,* Vol. 4, No. 8


Previous research work about improving the performance of decompressing and intersecting lists mainly focused on implementing these algorithms on single-core or multi-core CPU platforms, while the GPU offers an alternative approach. Wu et al. [24] presented a GPU parallel intersection framework, where queries are grouped into batches by a CPU, then each batch is processed by a GPU in parallel. Since a GPU uses thousands of threads during peak performance, we will require some kind of batched algorithm to make optimum use of GPU. In this paper we consider techniques for improving the performance of the GPU batched algorithm proposed in [24] assuming sufficient queries at the CPU end.

We begin with the problem of uncompressed sorted lists intersection on the GPU, and then consider how to efficiently intersect compressed lists. For uncompressed lists, we aim to contract the initial bounds of the basic binary search algorithm [7]. We propose two improvements, Linear Regression (LR) and Hash Segmentation (HS). In the LR method, to intersect two lists $\ell_A$ and $\ell_B$, for each element in $\ell_A$ we propose bounds for its location in $\ell_B$ based on a linear regression model. In the HS algorithm, an extra index is introduced to give more precise initial bounds. For the case of compressed lists, we will introduce a compression method called Linear Regression Compression (LRC) to substantially improve the decompression speed.

Upon inspection of representative real-world datasets from various sources, we find that the inverted lists show significant linear characteristics regardless of whether the docIDs are randomly assigned or have been ordered by some processes. The aim of this paper is to improve the efficiency of search engine algorithms by exploiting this linearity property on a GPU. Through experimentation, we find that an inverted list which has been reordered to have high locality (i.e. clusters of similar values) [6] does not necessarily show the best performance. An inverted index which has random sorted docIDs will have better performance in the algorithms described in this paper.

## 2. PRELIMINARIES

### 2.1 Lists Intersection

For simplicity, we consider the problem of querying for a subset of a large text document collection. We consider a document to be a set of *terms*. Each document is assigned a unique *document ID* (docID) from $1, 2, \ldots, \mathcal{U}$, where $\mathcal{U}$ is the

number of documents. The most widely used data structure for text search engines is the *inverted index* [23], where, for each term $t$, we store the strictly increasing sequence $\ell(t)$ of docIDs showing in which document the term appears. The sequences $\ell(t)$ are called *inverted lists*. If a $k$-term query is made for $t_1, t_2, \ldots, t_k$, then the inverted lists intersection algorithm simply returns the list intersection $\cap_{1\leq i\leq k}\ell(t_i)$. We may also assume that

$$|\ell(t_1)| \leq |\ell(t_2)| \leq \cdots \leq |\ell(t_k)|, \tag{1}$$

otherwise we may re-label $t_1, t_2, \ldots, t_k$ so that (1) holds.

To illustrate, if the query "2010 world cup" is made, the search engine will find the inverted lists for the three terms "2010", "world", and "cup", which may look like

$$\begin{aligned}\ell(\text{cup}) &= (13, 16, 17, 40, 50),\\ \ell(\text{world}) &= (4, 8, 11, 13, 14, 16, 17, 39, 40, 42, 50),\\ \ell(2010) &= (1, 2, 3, 5, 9, 10, 13, 16, 18, 20, 40, 50).\end{aligned}$$

The intersection operation returns the list intersection

$$\ell(\text{cup}) \cap \ell(\text{world}) \cap \ell(2010) = (13, 16, 40, 50).$$

In practice, search engines usually partition the inverted indexes into levels according to the frequency in which the corresponding term is queried. For example, Baidu, which is currently the dominant search engine in Chinese, stores the most frequently accessed inverted indexes in main memory for faster retrieval. In this paper, we also store the frequently accessed inverted indexes in the GPU memory, and only consider queries that request these inverted indexes.

## 2.2 Index Compression

For the case of index compression, we only consider compressing and decompressing the docIDs. Investigating compression and decompression algorithms for other pertinent information, such as data frequency and location, is beyond the scope of this paper.

In real-world search engines, typically the lists $\ell(t)$ are much longer than in the above example. Some form of compression is therefore needed to store the inverted lists $\ell(t)$; a straightforward approach is *variable byte encoding* [17]. To further reduce the index size, modern search engines usually convert an inverted list $\ell(t)$ to a sequence of *d-gaps* $\Delta\ell(t)$ by taking differences between consecutive docIDs in $\ell(t)$. For example, if $\ell(t) = (8, 26, 30, 40, 118)$, then the sequence of d-gaps is $\Delta\ell(t) = (8, 18, 4, 10, 78)$. If $\ell(t)$ is assumed to be a strictly increasing sequence from $\{1, 2, \ldots, \mathcal{U}\}$ chosen uniformly at random, then the elements in $\Delta\ell(t)$ will conform to a geometric distribution [23]. Moreover, if $|\ell(t)|$ is sufficiently large with respect to $\mathcal{U}$, we can expect $\ell(t)$ to approximately increase linearly (see Section 4.3 for details).

# 3. RELATED WORK

The problem of computing the intersection of sorted lists has received extensive interest. Previous work focuses on "adaptive" algorithms [2, 4, 11], which make no *a priori* assumptions about the input, but determine the type of instance as the computation proceeds. The run-time should be reasonable in most instances, but not in a worst-case scenario. For instance, the algorithm by Demaine et al. [8] proceeds by repeatedly cycling through the lists in a round-robin fashion.

In the area of parallel lists intersection, Tsirogiannis et al. [22] studied lists intersection algorithms suitable for the characteristics of *chip multiprocessors* (CMP). Tatikonda et al. [21] compared the performance between intra-query and inter-query models. Ding et al. [10] proposed a parallel lists intersection algorithm *Parallel Merge Find* (PMF) for use with the GPU.

Compression algorithms which have a good compression ratio or fast decompression speed have been studied extensively. Some examples are Rice Coding [26], S9 [1], S16 [25], PForDelta [13], and so on.

A straightforward method of compressing inverted lists $\ell(t)$ is to instead store the sequence of d-gaps $\Delta\ell(t)$, whose values are typically much smaller than the values in $\ell(t)$. Smaller d-gaps allow better compression when storing inverted lists. Therefore, reorder algorithms can be used to produce "locality" in inverted lists to achieve better compression. Bladford et al. [6] described a *similarity graph* to represent the relationship among documents. Each vertex in the graph is one document, and edges in the graph correspond to documents that share terms. Recursive algorithms are used to generate a hierarchical clustering based on the graph, where the docIDs are assigned during a depth-first traversal. Shieh et al. [19] also used a graph structure similar to that of the similarity graph, however the weight of the edges was determined by the number of terms existing in both the two documents. The cycle with maximal weight in the graph is then found, and the docIDs are assigned during the traversal of the cycle. To reorder the docIDs in linear time, Silvestri et al. [20] used a "k-means-like" clustering algorithm.

# 4. GPU BASED LISTS INTERSECTION

We direct readers unfamiliar with GPU and the CUDA architecture to Appendix A for an introduction. See Appendix B for details about the datasets used in this article.

## 4.1 Parallel Architecture

In order to fully utilize the processing power of the GPU, we store queries in a buffer until sufficiently many are made, then process them simultaneously on the GPU in one kernel invocation. Since we are assuming heavy query traffic, we also assume that there are no delays due to buffering. We use the batched intersection framework PARA, proposed by Wu et al. [24]. Suppose we receive a stream of queries that give rise to the inverted lists $\ell_j(t_i)$ from the $i$-th term in the $j$-th query. The assumption (1) implies that $|\ell_j(t_1)| = \min_i |\ell_j(t_i)|$. In PARA, a CPU continuously receives queries until $\sum_j |\ell_j(t_1)| \geq c$, where $c$ is some desired "computational threshold", and sends the queries to the GPU as a batch. The threshold indicates the minimum computational effort required for processing one batch of queries.

A bijection is then established between docIDs in $\ell_j(t_1)$ and GPU threads, which distributes computational effort among GPU cores. Each GPU thread searches the other lists to determine whether the docID exists. After all the threads have finished searching, a *scan* operation [18] and *compaction* operation [5] are performed to gather the results. Since the search operation occupies the majority of the time, optimization for search algorithm is crucial to system performance. In our paper, we focus on improving the search operation of PARA.

## 4.2 Binary Search Approach

In this paper we use *binary search* (BS) [7] as a "base" algorithm for comparison between parallel algorithms. Although it is neither the fastest algorithm on the CPU nor on the GPU, it provides a baseline from which we can compare the performance of the discussed algorithms. More efficient algorithms, such as the *skip list* algorithm [14, 16] and *adaptive* algorithms [4, 9] are inherently sequential, so run efficiently on a CPU but not on a GPU, and cannot be used to give a meaningful comparison. For state-of-art GPU lists intersection, we give an analysis of *Parallel Merge Find* (PMF) in Appendix C and show that binary search is better choice of baseline in our case.

We choose binary search as our underlying algorithm and adopt *element-wise* search techniques rather than *list-wise* merging [4]. More specifically, we have a large number of threads running in parallel, with each independently searching for a single docID from the shortest list $\ell_j(t_1)$ in the longer lists $(\ell_j(t_i))_{2\le i\le k}$. Moreover, we will discuss methods for contracting the initial search range in order to reduce the number of global memory accesses required.

## 4.3 Linear Regression Approach

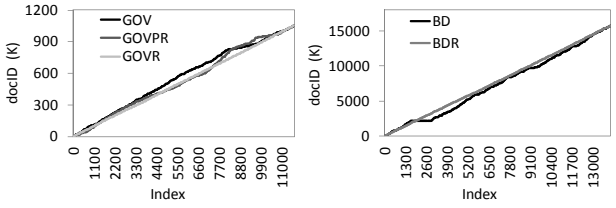


**Figure 1: Scatter plots (sampling every 50-th)**

Figure 1 gives some examples of scatter plots for inverted lists $\ell(t)$ obtained from the datasets GOV, GOVPR, GOVR, BD, and BDR. We plot the $i$-th element in $\ell(t)$ versus the index $i$. Figure 1 suggests that inverted lists $\ell(t)$ tend to have linear characteristics.

*Interpolation search* (IS) [15] is the most commonly used search algorithm, and exploits the linearity property of inverted lists. Interpolation search performs $O(\log\log|\ell(t)|)$ comparisons on average on a uniformly distributed list $\ell(t)$, although it can be as bad as $O(|\ell(t)|)$. In our preliminary work, we find that interpolation search is significantly slower than the binary search on the GPU.

1) On a GPU, SIMT architecture is used. Each kernel invocation waits for every thread to finish before continuing. In particular, a single slow thread will cause the entire kernel to be slow.
2) As mentioned earlier, modern real-world datasets generally reorder docIDs to improve compression ratio. Reordering leads to local non-linearity in the inverted lists. Interpolation search does not perform well in these circumstances.
3) A single comparison in an interpolation search is more complicated than in a binary search – the former issues 3 global memory accesses while the latter issues only 1.

In conclusion, interpolation search does not suit the GPU well. However, we will now describe a way to use the linearity of inverted lists to reduce the initial search range of a binary search.

We have shown the approximate linearity of inverted lists, which motivates using *linear regression* (LR) to contract the search range. Since this approach just contracts the initial search range of binary search, it does not introduce additional global memory accesses. Moreover, it is not impacted by local non-linearity significantly.

Provided $|\ell(t)|$ and $\mathcal{U}$ are sufficiently large, we can approximate $\ell(t)$ by a line $f_t(i) := \alpha_t i + \beta_t$, where $\alpha_t$ and $\beta_t$ can be found using a *least-squares linear regression*. Suppose we want to search for the value $x \in \{1, 2, ..., \mathcal{U}\}$ in $\ell(t)$. Then we can estimate the position of $x$ in $\ell(t)$ by $f_t^{-1}(x) = (x - \beta_t)/\alpha_t$. For $i \in \{1, 2, \ldots, |\ell(t)|\}$, let $\ell[i]$ be the $i$-th element of $\ell = \ell(t)$ and define the *maximum left deviation* $L_t = \max_i(f_t^{-1}(\ell[i]) - i)$ and the *maximum right deviation* $R_t = \max_i(i - f_t^{-1}(\ell[i]))$. If $x$ is actually in $\ell(t)$, then $x = \ell[i]$ for some $i \in \{j : f_t^{-1}(x) - L_t \le j \le f_t^{-1}(x) + R_t\}$, which we call the *safe search range* for $x$. We depict this concept in Figure 2.

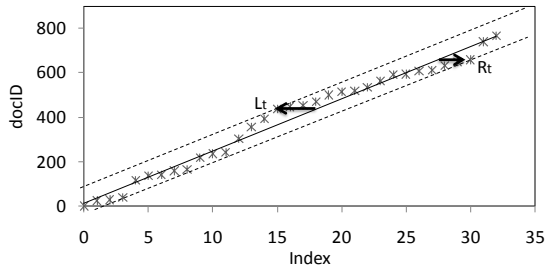


**Figure 2: Linear Regression approach**

A simple strategy for implementing this observation, is to store precomputed values of $\alpha_t$, $\beta_t$, $L_t$, and $R_t$ for all terms $t$. Then whenever we want to search $\ell(t)$ for $x$, we can simply compute the safe search range of the stored values, and begin a binary search whose range is the safe search range. Also note that care needs to be taken to avoid rounding errors.

Compared with binary search on the range $\{1, 2, ..., |\ell(t)|\}$, the performance improvement is determined by the *contraction ratio* $(L_t + R_t)/|\ell(t)|$. A small contraction ratio implies that the search range is contracted, so the subsequent binary search is faster. We inspect several representative real-world datasets and tabulate the average contraction ratio and the average *coefficient of determination* $R^2_{xy}$ in Table 1. Note that the inverted lists of datasets that have been randomized tend to be more linear, that is $R^2_{xy}$ is closer to 1. Moreover, when the inverted lists are more linear, the contraction ratio tends to be better.

Another possible strategy is to use a *local safe range*, which is the same as the safe search range strategy, but the inverted list is first divided into $g$ segments (similar to the segmentation of PMF in Appendix C) and the safe search range strategy is applied to each segment individually. Local safe range will obtain a narrower search range, but requires additional storage. Moreover, experimental results suggest that local safe range is not superior due to extra floating point operations.

## 4.4 Hash Segmentation Approach

Another range restricting approach we consider is *hash segmentation* (HS). We partition the inverted list $\ell(t)$ into *hash buckets* $B_h$, where $x \in \ell(t)$ is put into the hash bucket $B_h$ if $h = h(x)$ for some hash function $h$. As usual, we

Table 1: Average contraction ratio and $R^2_{xy}$ on different datasets

| $\lvert\ell(t)\rvert$ | GOV | | GOVPR | | GOVR | | GOV2 | | GOV2R | | BD | | BDR | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ratio | $R^2_{xy}$ | ratio | $R^2_{xy}$ | ratio | $R^2_{xy}$ | ratio | $R^2_{xy}$ | ratio | $R^2_{xy}$ | ratio | $R^2_{xy}$ | ratio | $R^2_{xy}$ |
| (0K,100K) | 0.2198 | 0.9613 | 0.2496 | 0.9573 | 0.1227 | 0.9870 | 0.4294 | 0.8731 | 0.1025 | 0.9898 | 0.3323 | 0.9271 | 0.1113 | 0.9891 |
| [100K,200K) | 0.0375 | 0.9991 | 0.0751 | 0.9952 | 0.0030 | 0.9999 | 0.2460 | 0.9636 | 0.0033 | 0.9999 | 0.1357 | 0.9746 | 0.0033 | 0.9999 |
| [200K,400K) | 0.0306 | 0.9995 | 0.0618 | 0.9966 | 0.0019 | 0.9999 | 0.1516 | 0.9847 | 0.0023 | 0.9999 | 0.0760 | 0.9943 | 0.0022 | 0.9999 |
| [400K,600K) | 0.0186 | 0.9997 | 0.0565 | 0.9972 | 0.0012 | 0.9999 | 0.1217 | 0.9896 | 0.0016 | 0.9999 | 0.0661 | 0.9957 | 0.0017 | 0.9999 |
| [600K,800K) | 0.0069 | 0.9999 | 0.0388 | 0.9985 | 0.0008 | 0.9999 | 0.1296 | 0.9884 | 0.0013 | 0.9999 | 0.0667 | 0.9973 | 0.0014 | 0.9999 |
| [800K,1M) | 0.0060 | 0.9999 | 0.0308 | 0.9990 | 0.0006 | 0.9999 | 0.1076 | 0.9946 | 0.0011 | 0.9999 | 0.0838 | 0.9955 | 0.0009 | 0.9999 |

assume that $B_h$ is a strictly increasing ordered set. If we wish to search for $x \in \{1, 2, ..., \mathcal{U}\}$ in an inverted list $\ell(t)$, we need only check whether $x \in B_{h(x)}$ using a binary search.

As per our earlier discussion, real-world inverted lists tend to have linear characteristics, so we choose a very simple hash function. Let $k$ be the smallest integer such that $\mathcal{U} \leq 2^k$. For some $m \leq k$, we define $h(x) = h_m(x)$ to be the leading $m$ binary digits of $x$ (when written with exactly $k$ binary digits in total), which is equivalent to $h(x) = \lfloor x/2^{k-m} \rfloor$. Many hash buckets $B_h$ will be empty, specifically those with $h > h(\max_i \ell[i])$ and most likely $B_{h(\max_i \ell[i])}$ will contain fewer docIDs than the other non-empty hash buckets.

In contrast to PMF, the cardinalities $|S_j|$ are predetermined and are all equal to $2^{k-m}$. Moreover, the hash buckets do not need compaction when $k$-term queries are made and $k \geq 3$. The advantage of this scheme is that finding the hash bucket $B_h$ that $x$ belongs to can simply be performed by computing its hash value $h(x)$ (as opposed to using a Merge Find algorithm).

Implementing this scheme introduces some overhead. For every term $t$, the hash bucket $B_h$ can be stored as a pointer (or an offset) to the minimum element of $B_h$ (or, if $B_h$ is empty, we store a pointer to the minimum element of the next non-empty hash bucket). So for each term $t$ we will require storage of $d+1$ pointers, where $d = h(\max_i \ell[i])+1 \in [1, 2^m]$. When we want to search for $x$ in $\ell(t)$, we compute its hash value $h = h(x)$. If $h > h(\max_i \ell[i])$, then $x \notin \ell(t)$ and the search is complete. Otherwise, we find the pointers for $B_h$ and perform a binary search with the search range restricted to $B_h$. In practice, the computation of $h(x)$ and finding $B_h$ is negligible, so the number of steps required to find $x$ will be roughly the number of steps required to perform a binary search for $x$ in $B_h$ where $h = h(x)$. The bigger $m$ is, the fewer comparisons need to be performed by the binary search. However, if $m$ is too big, it will cause overhead issues.

The set of terms in a given document and the assignment of docIDs are determined by random processes outside of our control. We assume that the probability of any given docID $x \in \{1, 2, \ldots, \mathcal{U}\}$ being in a random $\ell(t)$ is $p = p(t) = \frac{|\ell(t)|}{\mathcal{U}}$. Therefore, for a given term $t$, the cardinality of a non-empty hash bucket $B_h$ approximately follows a binomial distribution $|B_h| \sim \text{Bi}(|\ell(t)|, p)$ where $p = 2^{k-m}/\mathcal{U}$, with mean $|\ell(t)|p$ and variance $|\ell(t)|p(1-p)$.

Figure 3 displays a count of hash bucket cardinalities over all inverted lists $\ell(t)$ in the GOV dataset provided $2^{12}+1 \leq |\ell(t)| \leq 2^{13}$ and all non-empty hash buckets, where $m = 5$. We see the binomial distribution appearing, however it is stretched since $|\ell(t)|$ is not constant. In practice, we choose $m$ dynamically depending on $|\ell(t)|$, for example, we define the algorithm *HS256* to have the minimum $m$ such that $|\ell(t)|/256 \leq 2^m$ (which we consider in Section 6). Even if the inverted lists are not particularly linear, hash segmentation still performs well; see Table 3 for experimental results.

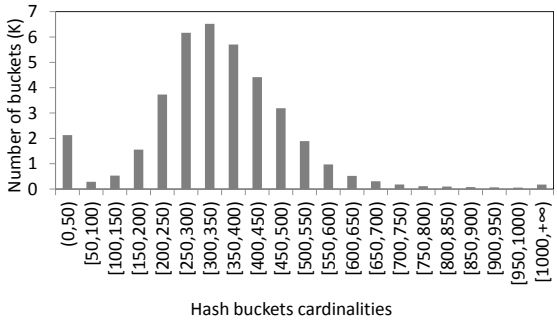


**Figure 3: Distribution of hash buckets size**

# 5. GPU BASED INDEX COMPRESSION

## 5.1 PFor and ParaPFor

*Patched Frame-of-Reference* (PFor) [13, 27] divides the list of integers into segments of length $s$, for some $s$ divisible by 32. For each segment $a$, we determine the smallest $b = b(a)$ such that most integers in $a$ (e.g. 90%) are less than $2^b$, while the remainder are *exceptions*. Each of the integers in $a$, except the exceptions, can be stored using $b$ bits. For each exception, we instead store a pointer to the location of the next exception. The values of the exceptions are stored after the $s$ slots. If the offset between two consecutive exceptions is too large, i.e. requires more than $b$ bits to write, then we force some additional exceptions in-between. We call PFor on $\Delta\ell(t)$ *PForDelta* (PFD).

A variant of PFD, called *NewPFD*, was presented in [25]. In NewPFD, when an exception is encountered, the least significant $b$ bits are stored in a $b$-bit slot, and the remaining bits (called the *overflow*) along with the pointer are stored in two separate arrays. The separate arrays may be encoded by S16 [25], for example.

Decompression speed is more important to the performance of query processing since the inverted lists are decompressed while the user waits. On the other hand, compression is used only during index building. Consequently, we will focus on optimizing the decompression performance. Typically PFor has poor decompression performance on the GPU. In PFor, the pointers are organized into a linked list. Therefore the decompression of the exceptions must be executed serially. The number of global memory accesses

a thread performs is proportional to the number of exceptions. So we make a modification to PFor called *Parallel PFor* (ParaPFor). Instead of saving a linked list in the exception part, we store the indices of exceptions in the original segment (See Appendix D for details). This modification will lead to a worse compression ratio, but gives much faster decompression on the GPU because exceptions can be recovered concurrently. Consequently, we will describe a new index compression method, Linear Regression Compression.

## 5.2 Linear Regression Compression

As described in Section 4.3, for typical inverted lists, linear regression can be used to describe the relationship between the docIDs and their indices. We fix some term $t$. Given a linear regression $f_t(i) := \alpha_t i + \beta_t$ of an inverted list $\ell(t)$, a given index $i$, and its *vertical deviation* $\delta_t(i)$, the $i$-th element of $\ell(t)$ is $f_t(i) + \delta_t(i)$. Therefore, it is possible to reconstruct $\ell(t)$ from a list of vertical deviations (VDs) and the function $f_t$.

Vertical deviations may be rational numbers, positive or negative, but for implementation, we map them to the non-negative integers. Let $M_t = -\min_i \lceil \delta_t(i) \rceil$ (which is stored), so $M_t + \lceil \delta_t(i) \rceil \geq 0$ for all $i$. Since the $i$-th element of $\ell(t)$ is $f_t(i) + \delta_t(i)$, which is always a positive integer, we store $\lambda_t(i) = M_t + \lceil \delta_t(i) \rceil$. Hence the $i$-th element of $\ell(t)$ is $f_t(i) + \delta_t(i) = \lfloor f_t(i) + \lambda_t(i) - M_t \rfloor$.

We can perform compression by applying any index compression technique on the normalized vertical deviations $(\lambda_t(i))_{1 \leq i \leq |\ell(t)|}$. In this paper, we will use ParaPFor. We call this compression method *Linear Regression Compression* (LRC). The advantage of LRC is that it can achieve higher decompression concurrency over d-gap based compression schemata.

We give a detailed analysis of LRC in Appendix E, which gives a theoretical guarantee of the compression ratio and can be easily extended to the contraction ratio of LR (Section 4.3).

The *fluctuation range* of vertical deviations in LRC is $\max_i \lambda_t(i)$. Again, if we divide the list $(\delta_t(i))_{1 \leq i \leq |\ell(t)|}$ into segments, we can observe smaller fluctuation ranges locally. We consider two segmentation strategies:

- Performing linear regression globally and then performing segmentation to obtain better local fluctuation ranges (*LRCSeg*),
- Performing segmentation first, then performing linear regression compression for each segment (*SegLRC*).

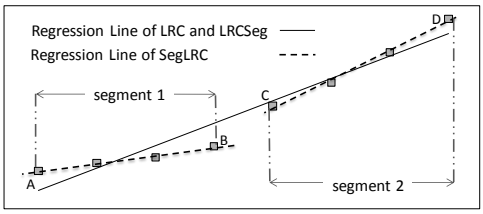


**Figure 4: LRC, LRCSeg, and SegLRC**

We depict these three methods in Figure 4. Note that the two local regression lines of SegLRC have much better goodness of fit than the global regression line of LRC and LRCSeg. LRCSeg obtains a local fluctuation range on segment 2, smaller than the global fluctuation range of LRC. Both LRCSeg and SegLRC give significantly better compression than LRC (see Figure 7 for a comparison).

## 5.3 Lists Intersection with LRC on the GPU

PFD compresses d-gaps, so to find the $i$-th element of an inverted list $\ell(t)$, we need to recover the first $i$ elements from $\Delta\ell(t)$. In LRC however, if the binary search accesses the $i$-th element of $\ell(t)$, we can decompress that element alone. The number of global memory accesses required is proportional to the number of comparisons made.

Algorithm 1 presents the lists intersection with LRC on the GPU. For simplicity, we will forbid exceptions, which should not significantly effect the compression ratio (see Figure 7 for a comparison). We consider the inputs to be

- $k$ inverted lists $\ell_C(t_1), \ell_C(t_2), \ldots, \ell_C(t_k)$ that have been compressed using LRC, where we assume condition (1),
- for $i \in \{2, 3, \ldots, k\}$, an auxiliary ordered list $\mathcal{H}(t_i)$ which contains the $\lceil |\ell(t_i)|/s \rceil$ elements of $\ell(t_i)$ whose coordinates are congruent to 0 (mod $s$). In fact, $\mathcal{H}(t_i)$ comprises the headers of all the segments of $\ell(t_i)$.

**Algorithm 1** *Lists Intersection with LRC*

**Input:** $k$ compressed lists $\ell_C(t_1), \ell_C(t_2), \ldots, \ell_C(t_k)$ and $k-1$ ordered lists $\mathcal{H}(t_2), \mathcal{H}(t_3), \ldots, \mathcal{H}(t_k)$ stored in global memory
**Output:** the lists intersection $\cap_{1 \leq i \leq k} \ell(t_i)$

1: **for** *each thread* **do**
2: Recover a unique docID $p$ from $\ell_C(t_1)$ using *ParaPFor Decompression* (see Appendix D.2) and linear regression of $\ell(t_1)$.
3: **for** *each list* $\ell_C(t_i)$, $i = 2 \ldots k$ **do**
4: Compute the safe search range $[f_{t_i}^{-1}(p) - L_{t_i}, f_{t_i}^{-1}(p) + R_{t_i}]$.
5: Perform binary search for $p$ in the search interval $[(f_{t_i}^{-1}(p) - L_{t_i})/s, (f_{t_i}^{-1}(p) + R_{t_i})/s]$ of $\mathcal{H}(t_i)$, to obtain $x$ such that $\mathcal{H}(t_i)[x] \leq p < \mathcal{H}(t_i)[x+1]$.
6: Perform binary search for $p$ in the $x$-th segment of $\ell_C(t_i)$.
7: If $p$ is not found in $\ell_C(t_i)$, then break.
8: **end for**
9: If $p$ is found in $k$ lists, then record $p \in \cap_{1 \leq i \leq k} \ell(t_i)$.
10: **end for**

We assume that one global memory access takes time $t_a$, while one comparison takes time $t_c$. Therefore it takes $2t_a$ to decompress an element from $\ell_C(t_i)$ during each step of the binary search (Line 6). The total running time required to perform lists intersection under LRC is at most

$$\sum_{i=2}^{k} \Bigg( \overbrace{(t_a + t_c) \left\lceil \log \frac{\mathtt{cr}_{t_i} |\ell(t_i)|}{s} \right\rceil}^{\text{Line 5}} + \overbrace{(2t_a + t_c) \lceil \log(s) \rceil}^{\text{Line 6}} \Bigg) \quad (2)$$

per thread, where $\mathtt{cr}_{t_i}$ is the global contraction ratio of $\ell(t_i)$ for all $i \in \{2, 3, \ldots, k\}$. In fact, the total running time required by LRCSeg is also given by (2). For comparison, we could also perform lists intersection with LRC without the auxiliary lists, when the corresponding total running time is at most

$$\sum_{i=2}^{k} \Big( (2t_a + t_c) \lceil \log(\mathtt{cr}_{t_i} |\ell(t_i)|) \rceil \Big)$$

per thread. Experimental results suggest that the former takes 33% less GPU time than the latter. The cost we pay for such achievement is 0.80% reduction of compression ratio due to the space occupied by the auxiliary ordered lists.

In SegLRC it is also possible to reduce the search range using the linear regression technique described in Section 4.3. After locating the segment, binary search can be performed

on the compressed list segment, where again the search range can be reduced by applying the linear regression technique to the segment. The total running time per thread required to perform lists intersection under SegLRC is at most

$$\sum_{i=2}^{k}\left((t_a+t_c)\left\lceil\log\frac{\mathtt{cr}_{t_i}|\ell(t_i)|}{s}\right\rceil+(2t_a+t_c)\lceil\log(\mathtt{cr}'_{t_i}s)\rceil\right),$$

where $\mathtt{cr}'_{t_i}$ is the maximum local contraction ratio of $\ell(t_i)$ for all $i\in\{2,3,\ldots,k\}$.

Furthermore, we can narrow the search range by combining HS (as described in Section 4.4) with LRC. While compressing, we apply LRC to the hash buckets. Although the buckets may vary in size, experimental results show that the compression ratio is almost the same as SegLRC (when segments are of fixed width). We call this method *HS_LRC*. During lists intersection, we can locate the segment by the docID's hash value, and then we use a local linear regression to narrow the search range. Experimental results suggest that performance of lists intersection improves greatly as a result, which we will discuss in the next section.

# 6. EXPERIMENTAL RESULTS

Appendix F.1 lists the details of experimental platform.

## 6.1 Throughput and Response Time

We now consider the performance of algorithms under different computational threshold $c$, as usual, assuming heavy query traffic. As mentioned in Section 4.1, the chosen computational threshold used for PARA determines the minimum computational effort in one batch. A higher threshold makes better use of the GPU, and the throughput will increase. Furthermore, less PCI-E transfers are invoked since more data can be packed into one PCI-E transfer. Since fewer large PCI-E transfers are faster than many smaller transfers, the overhead of PCI-E transferring could be reduced.

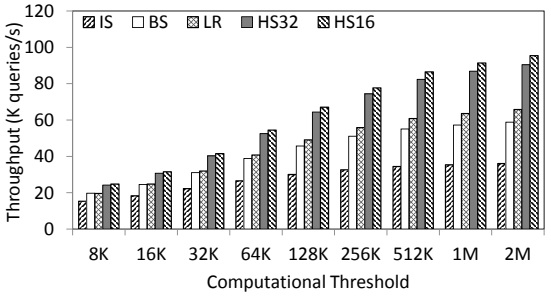


**Figure 5: System throughput on GOV**

We first test the throughput of different parallel intersection algorithms on uncompressed lists. As Figure 5 illustrates, LR and HS improve the throughput significantly. This is mainly due to the reduction of memory accesses. The search range of LR is determined by the safe search range, while the search range of HS is determined by the bucket size. When threshold is 1M, HS16 boosts the throughput to 91382 queries/s, which is 60% higher than BS. The cost we pay for such achievement is 9% extra memory space. The throughput of HS16 maintains the obvious upward trend even when the threshold reaches 2M. Such trend suggests the potential of the GTX480 has not been fully utilized. Search engines with lighter load could equip their servers with slower, less power-consuming GPU, like GTX460, so as to save energy and reduce carbon emission.

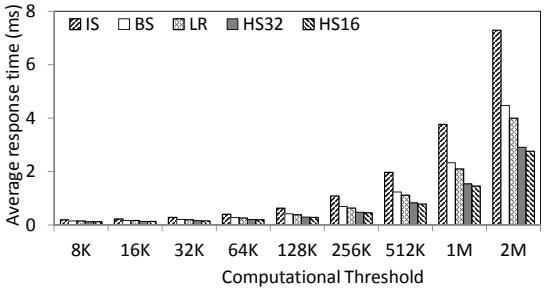


**Figure 6: Average response time on GOV**

The response time is a crucial indicator to user experience, so it is another important performance criterion of lists intersection. In the PARA framework, the response time is the processing time for each batch, which consists of three parts: CPU time, GPU intersection time, and transfer time. Since a higher threshold implies that a batch contains more queries before processing, the response time will be prolonged. Figure 6 presents the response time of each algorithm on uncompressed lists. When the threshold is less than 128K, the difference in response times is indistinguishable. This is because all threads can process small batches within similar time. As the threshold grows, the advantage of more efficient algorithms becomes more significant. Major search engines have strict requirements on response time, so the choice of threshold should achieve a balance between throughput and response time.

We also compare different algorithms intersecting compressed lists. See Appendix F.2 for details.

## 6.2 Compression and Decompression

We will now compare the compression ratio and decompression speed of the index compression techniques proposed in this paper. We restrict the proportion of exceptions to at most 0.6. We set the segment length in PFD, NewPFD, and ParaPFD to 64, while the segment length in LRC, LRCSeg, SegLRC, and HS_LRC will be 256. We use the GOV dataset for comparison. For compression, we take the compression ratio over all inverted lists $\ell(t)$, whereas for decompression we take the decompression speed of shortest inverted list $\ell(t_1)$ over all queries.

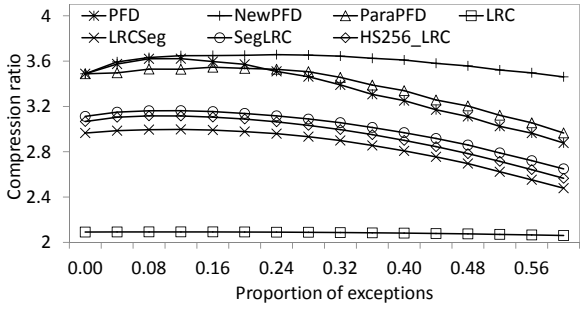


**Figure 7: Compression ratio on GOV**

We calculate the compression ratio as the size of the original inverted lists to the total size of the compressed lists plus the auxiliary information. As Figure 7 shows, the best compression ratio is obtained with PFD, NewPFD, and

ParaPFD, while the compression ratio of LRCSeg, SegLRC, and HS256_LRC is also reasonable. The compression ratio of all the methods (except LRC) initially increases as the proportion of exceptions increases, but then decreases, which implies that too many exceptions reduce compression efficiency. In particular, allowing no exceptions achieves a compression ratio close to the maximum. For LRC, the compression ratio remains practically unchanged as the proportion of exceptions varies, indicating that the distribution of vertical deviations in segments are similar, to which we attribute the significant improvement in compression when we adopt LRCSeg.

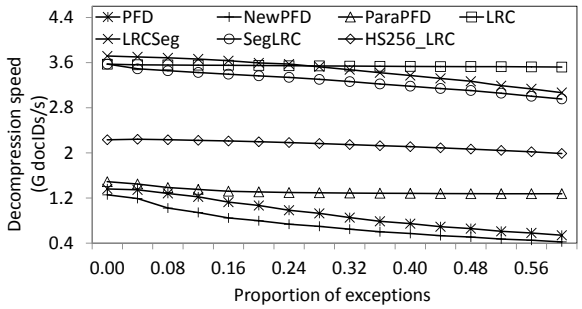


**Figure 8: Decompression speed on GOV**

Figure 8 shows the decompression speed of the shortest lists of all the queries in the GOV dataset. We only include the shortest lists since the algorithms presented in this paper only need to decompress the shortest list completely for each query. We can see that the best results are obtained with LRC, LRCSeg, and SegLRC, achieving significantly faster decompression over PFD and NewPFD. For PFD and NewPFD the decompression speed varies with the number of exceptions, whereas ParaPFD and the LR based methods have nearly constant decompression speed.

**Table 2: Optimal compression ratio**

| | PFD | NewPFD | ParaPFD | LRC | LRCSeg | SegLRC | HS256_LRC |
|---|---|---|---|---|---|---|---|
| GOV | 3.62 | 3.66 | 3.55 | 2.09 | 3.00 | 3.16 | 3.12 |
| GOVPR | **3.63** | **3.68** | **3.57** | 2.02 | 2.85 | 3.11 | 3.00 |
| GOVR | 3.61 | 3.64 | 3.53 | **2.62** | **3.26** | **3.23** | **3.22** |
| GOV2 | **3.71** | **3.75** | **3.63** | 1.73 | 2.82 | 3.19 | 3.18 |
| GOV2R | 3.60 | 3.62 | 3.51 | **2.41** | **3.24** | **3.22** | **3.21** |
| BD | **2.78** | **3.09** | **2.89** | 1.59 | 2.03 | 2.26 | 2.10 |
| BDR | 2.58 | 2.61 | 2.55 | **1.99** | **2.39** | **2.38** | **2.34** |

Table 2 gives the compression ratios of the various algorithms over different datasets, optimized with respect to the proportion of exceptions. LR based algorithms perform poorly on GOVPR, since GOVPR has been sorted by PageRank, producing "locality". An important observation is that LR based algorithms perform best on GOVR, GOV2R, and BDR. Figure 1 indicates that GOVR and BDR have strong linearity, and moreover, in Table 1 the $R^2_{xy}$ values for GOVR, GOV2R, and BDR are closer to 1, implying that the inverted lists of the randomized datasets tend to be more linear. Therefore, the vertical deviations should typically be smaller and the compression ratios should therefore be better. All this suggests that LR based algorithms will benefit from randomized docIDs.

## 6.3 Speedup and Scalability

We use the optimized version of skip list [16] as the "baseline" algorithm on the CPU, from which other algorithms can be compared. Experimental results indicate that it is the fastest single-threaded algorithm, when compared to those mentioned in [2, 4, 22]. The speedup obtained using multi-CPU-based architecture is bounded above by the number of available CPUs [22]. In Table 3 we tabulate the speedup of various algorithms over different datasets. HS16 achieves the greatest speedup among all algorithms on uncompressed lists on all datasets, whereas HS128_LRC achieves the greatest speedup for compressed lists. HS16 and HS32 achieve their greatest speedup on GOVPR rather than GOVR, indicating that linearity is not a key factor in HS algorithms. For compressed lists intersection, the speedup achieved by HS256_LRC is comparable to that of BS on uncompressed lists. On BD and BDR, we also achieve 14.67x speedup to uncompressed lists, and 8.81x speedup to compressed lists.

In Table 3, we notice that LR performs better on BD than BDR, and attribute this anomaly to branch divergency. Using CUDA Profiler, we find that the number of divergent branches is approximately 24% and 28% greater on BDR than BD when running BS and LR, respectively. Branch divergency plays a larger role when the dataset is randomized, such as in BDR. Thus we have a trade-off between branch divergency and linearity.

To investigate the speedup depending on the number of GPU cores, we flush the BIOS of GTX480 so as to disable some of the Streaming Multiprocessors (SMs). Figure 9 (a) shows the speedup of HS16 and HS256_LRC on the GOV dataset as the number of SMs increases. We set the computational threshold to 1M to fully utilize the GPU processing power. We can see that the speedup of both algorithms is almost directly proportional to the number of SMs. As the number of SMs increases, the efficiency of both algorithms decreases slightly, which is common to all parallel algorithms.

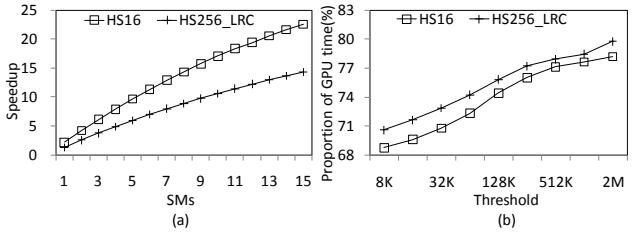


**Figure 9: (a) Speedup as the number of SMs increases and (b) the proportion of GPU Time**

Figure 9 (b) shows the utilization of GPU of HS16 and HS256_LRC on the GOV dataset with respect to the computational threshold. Note that the computational threshold decides the batch size, therefore, for a batched computing model, it is actually the problem size. We can see that the proportion of GPU time to the total execution time increases as the computational threshold increases, which implies both algorithms obtain an increasing efficiency as the problem size increases. Our experimental results show that our new algorithms can maintain efficiency at a fixed value by simultaneously increasing the number of processing elements and the problem size, that is they are scalable [12].

Figure 9 (b) also illustrates another phenomenon. CPU-GPU transfers and CPU computation always occupy more than 20% of total execution time. Overlapping CPU computation and transfers with GPU computation therefore presents itself as a future avenue for improvement.

**Table 3: Speedup**

| | Uncompressed | | | | | Compressed | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **IS** | **BS** | **LR** | **HS32** | **HS16** | **ParaPFD (0.2)** | **ParaPFD (0.0)** | **LRC** | **LRCSeg** | **SegLRC** | **HS256_LRC** | **HS128_LRC** |
| **GOV** | 8.69 | 14.59 | **16.22** | 22.14 | 23.29 | 4.05 | 5.86 | 10.79 | 10.79 | 12.07 | **14.26** | **14.38** |
| **GOVPR** | 7.67 | **14.81** | 15.86 | **22.45** | **23.54** | **4.10** | **5.94** | 10.68 | 10.67 | 11.74 | 14.10 | 14.22 |
| **GOVR** | **9.48** | 14.29 | 16.19 | 20.96 | 22.01 | 3.97 | 5.77 | **10.97** | **10.98** | **12.42** | 14.21 | 14.24 |
| **BD** | 2.16 | **9.86** | **9.98** | **14.19** | **14.67** | **2.65** | **3.87** | 7.22 | **7.24** | 7.42 | 8.60 | 8.78 |
| **BDR** | **5.64** | 8.70 | 9.42 | 12.35 | 12.96 | 2.43 | 3.53 | **7.24** | 7.22 | **7.93** | **8.80** | **8.81** |

# 7. CONCLUSION

In this paper, we present several novel techniques to optimize lists intersection and decompression, particularly suited for parallel computing on the GPU. Motivated by the significant linear characteristics of real-world inverted lists, we propose the Linear Regression (LR) and Hash Segmentation (HS) algorithms to contract the initial search range of binary search. For index compression, we propose the Parallel PFor (ParaPFor) algorithm that resolves issues with the decompression of exceptions in PFor that prevent it from performing well in a parallel computation. We also present the Linear Regression Compression (LRC) algorithm which further improves decompression concurrency and can be readily combined with LR and HS. We discuss the implementation of these algorithms on the GPU.

Experimental results show that LR and HS, especially the latter, improve the lists intersection operation significantly, and LRC also improves the index decompression and lists intersection on compressed lists while still achieving a reasonable compression ratio. Experimental results also show that LR based compression algorithms perform much better on randomized datasets.

# 8. REFERENCES


[1] V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1):151–166, 2005.

[2] R. Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *Combinatorial Pattern Matching*, pages 400–408, 2004.

[3] R. Baeza-Yates and A. Salinger. Experimental analysis of a fast intersection algorithm for sorted sequences. In *Proc. 12th International Conference on String Processing and Information*, pages 13–24, 2005.

[4] J. Barbay, A. López-Ortiz, and T. Lu. Faster adaptive set intersections for text searching. *Experimental Algorithms: 5th International Workshop*, pages 146–157, 2006.

[5] M. Billeter, O. Olsson, and U. Assarsson. Efficient stream compaction on wide SIMD many-core architectures. In *Proc. Conference on High Performance Graphics*, pages 159–166, 2009.

[6] D. Blandford and G. Blelloch. Index compression through document reordering. In *Proc. Data Compression Conference*, pages 342–351, 2002.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[8] E. D. Demaine, A. López-Ortiz, and J. Ian Munro. Adaptive set intersections, unions, and differences. In *Proc. 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 743–752, 2000.

[9] E. D. Demaine, A. López-Ortiz, and J. Ian Munro. Experiments on adaptive set intersections for text retrieval systems. *Third International Workshop on Algorithm Engineering and Experimentation*, pages 91–104, 2001.

[10] S. Ding, J. He, H. Yan, and T. Suel. Using graphics processors for high performance IR query processing. In *Proc. 18th International Conference on World Wide Web*, pages 421–430, 2009.

[11] V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *ACM Comput. Surv.*, 24(4):441–476, 1992.

[12] A. Grama, A. Gupta, and V. Kumar. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Parallel & Distributed Technology: Systems & Applications*, 1(3):12–21, 1993.

[13] S. Héman. Super-scalar database compression between RAM and CPU-cache. Master's thesis, Centrum voor Wiskunde en Informatica Amsterdam, 2005.

[14] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[15] Y. Perl, A. Itai, and H. Avni. Interpolation search – a log log N search. *Comm. ACM*, 21(7):550–553, 1978.

[16] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Comm. ACM*, 33(6):668–676, 1990.

[17] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *Proc. 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 222–229, 2002.

[18] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *Proc. 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, pages 97–106, 2007.

[19] W.-Y. Shieh, T.-F. Chen, J. J.-J. Shann, and C.-P. Chung. Inverted file compression through document identifier reassignment. *Inform. Process. Manag.*, 39(1):117–131, 2003.

[20] F. Silvestri, R. Perego, and S. Orlando. Assigning document identifiers to enhance compressibility of web search engines indexes. In *Proc. 2004 ACM Symposium on Applied Computing*, pages 600–605, 2004.

[21] S. Tatikonda, F. Junqueira, B. Barla Cambazoglu, and V. Plachouras. On efficient posting list intersection with multicore processors. In *Proc. 32nd international ACM SIGIR conference on Research and Development in Information Retrieval*, pages 738–739, 2009.

[22] D. Tsirogiannis, S. Guha, and N. Koudas. Improving the performance of list intersection. *Proc. VLDB Endowment*, 2(1):838–849, 2009.

[23] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999.

[24] D. Wu, F. Zhang, N. Ao, G. Wang, X. Liu, and J. Liu. Efficient lists intersection by CPU–GPU cooperative computing. In *25th IEEE International Parallel and Distributed Processing Symposium, Workshops and PhD Forum (IPDPSW)*, pages 1–8, 2010.

[25] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. 18th International Conference on World Wide Web*, pages 401–410, 2009.

[26] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):1–56, 2006.

[27] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Super-scalar RAM–CPU cache compression. In *Proc. 22nd International Conference on Data Engineering (ICDE'06)*, page 59, 2006.

# APPENDIX

## Acknowledgements

This paper is partially supported by the National High Technology Research and Development Program of China (2008A A01Z401), NSFC of China (60903028, 61070014), Science & Technology Development Plan of Tianjin (08JCYBJC1300 0), and Key Projects in the Tianjin Science & Technology Pillar Program. Stones was supported in part by an Australian Research Council discovery grant.

We would like to thank the reviewers for their time and appreciate the valuable feedback. We thank Caihong Qiu for the initial idea of using linear regression to contract the search range. Thanks to Didier Piau for providing the ideas about calculating the central moment. Thanks to Baidu for providing the dataset. Thanks to Hao Ge, Zhenyuan Yang, Zhiqiang Wang and Liping Liu for providing the important suggestions. Thanks to Guangjun Xie, Lu Qi, Ling Jiang, Xiaodong Lin, Shuanlin Liu and Huijun Tang for providing the helpful comments. Thanks to Shu Zhang for providing technical support concerning the CUDA. Thanks also to Haozhe Chang, Di He, Mathaw Skala, and Derek Jennings for discussing the theoretical analysis.

## A. GPU AND CUDA ARCHITECTURE

*Graphics Processing Units* (GPUs) are notable because they contain many processing cores, for example, in this work we use the NVIDIA GTX480 (Fermi architecture), which has 480 cores. Although GPUs have been designed primarily for efficient execution of 3D rendering applications, demand for ever greater programmability by graphics programmers has led GPUs to become general-purpose architectures, with fully featured instruction sets and rich memory hierarchies.

*Compute Unified Device Architecture* (CUDA) [30] is the hardware and software architecture that enables NVIDIA GPUs to execute programs written with C, C++, and other languages. CUDA presents a virtual machine consisting of an arbitrary number of *Streaming Multiprocessors* (SMs), which appear as 32-wide SIMD cores. The SM creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*. When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps that get scheduled by a warp scheduler for execution. A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. When this does not occur, it is referred to as *branch divergency*.

One of the major considerations for performance is memory bandwidth and latency. The GPU provides several different memories with different behaviors and performance that can be leveraged to improve memory performance. We briefly discuss two important memory spaces here. The first one is *global memory*. Global memory, where our experimental inverted indexes reside, is the largest memory space among all types of memories in the GPU. Accessing global memory is slow and access patterns need to be carefully designed to reach the peak performance. The second one is *shared memory*. Shared memory is small readable and writable on-chip memory and as fast as registers. A good technique for optimizing a naive CUDA program is using shared memory to cache the global memory data, performing large computations in the shared memory and writing the results back to the global memory.

## B. DATASETS

To compare the various algorithms, we use the following datasets:

1) TREC *GOV* [33] and *GOV2* [32] document sets are collections of Web data crawled from Web sites in the .gov domain in 2002 and 2004, respectively. They are widely used in IR community. We use the document sets to generate the inverted indexes. The docIDs are the original docIDs, and we do not have access to how the docIDs were assigned. We use Terabyte 2006 (T06) [28] query set which contains 100,000 queries to test GOV and GOV2 document sets. We focus on in-memory index, so we exclude the inverted lists $\ell(t)$ for which $t$ is not in T06. The GOV and GOV2 datasets we used contain 1053372 and 5038710 html documents, respectively.
2) We use the method proposed in [31] to generate the *PageRank* for all the documents in the GOV dataset, and reorder the docIDs by PageRank, i.e. the document with largest PageRank is assigned docID 1 and so on. We call this index *GOVPR*.
3) We form another two datasets from GOV and GOV2 by randomly assigning docIDs (Fisher-Yates shuffle [29]). We call these indexes *GOVR* and *GOV2R*, respectively.
4) Baidu dataset *BD* was used on Baidu's cluster (obtained via private communication). Baidu is the leading search engine in China responding billions of queries each day. BD contains 15749656 html documents crawled in 2009. We use Baidu 2009 query set which contains 33337 queries to test BD document set.
5) We also generate the dataset *BDR* from the BD dataset by randomly assigning docIDs.

## C. PARALLEL MERGE FIND

*Parallel Merge Find* (PMF) [10] is the fastest GPU lists intersection algorithm to date. The docIDs are partitioned into ordered sets

$$S_1 = (1, 2, \ldots, |S_1|),$$
$$S_2 = (|S_1| + 1, |S_1| + 2, \ldots, |S_1| + |S_2|),$$
$$S_3 = (|S_1| + |S_2| + 1, |S_1| + |S_2| + 2, \ldots, |S_1| + |S_2| + |S_3|),$$

and so on up to $S_g$. The inverted lists $\ell(t_i)$ are then split into $g$ parts $\ell(t_i) \cap S_j$. Splitting the inverted lists is performed by a Merge Find algorithm.

A parallelized binary search is then performed on the partial inverted lists $\ell(t_1) \cap S_j, \ell(t_2) \cap S_j, \ldots, \ell(t_k) \cap S_j$, where each element of $\ell(t_1) \cap S_j$ is assigned to a single GPU thread.

In the case of a 2-term query, if $\ell(t_1)$ is divided evenly, i.e. if each $\ell(t_1) \cap S_j$ has roughly the same cardinality, then the number of steps required by the GPU to perform PMF is bounded below by

$$\overbrace{\left\lceil \frac{g}{C} \right\rceil \log |\ell(t_2)|}^{\text{Merge Find}} + \overbrace{\left\lceil \frac{|\ell(t_1)|}{C} \right\rceil \log \frac{|\ell(t_2)|}{g}}^{\text{binary search}},$$

where $C$ is the maximum concurrency of the GPU.

We need to apply Merge Find $g$ times, of which we can perform at most $C$ in parallel, and each requires at least $\log |\ell(t_2)|$ steps. Afterwards, we apply binary search for each

element in $\ell(t_1)$, and again we can perform $C$ in parallel, with each search requiring at least $\log(|\ell(t_2)|/g)$ steps.

For comparison, the number of steps required for the binary search on the GPU (without splitting according to PMF) is bounded below by

$$\left\lceil \frac{|\ell(t_1)|}{C} \right\rceil \log |\ell(t_2)|.$$

Hence PMF is only superior to binary search when the shortest list $\ell(t_1)$ is sufficiently long, say containing millions of elements.

In Table 4 we list some data concerning the average ratios of the lengths of the inverted lists $\ell(t_i)$ in a $k$-term search using the GOV dataset.

For $k$-term searches with $k \in \{2, 3, \ldots, 6\}$, in Stage $x$ we write

$$R := \frac{|\ell(t_1) \cap \ell(t_2) \cap \cdots \cap \ell(t_x)|}{|\ell(t_{x+1})|} \tag{3}$$

as a percentage. The percentage in brackets after the $k$ value gives the proportion of queries that are $k$-term queries. We also write the average length of $\ell(t_1)$ in Stage $x$ in K (x1000), denoted $|\bar{\ell}|$. Table 4 shows that, in real-world datasets, most of queries have a short shortest list $\ell(t_1)$. As the algorithm progresses, that is as $x$ increases, (3) decreases significantly. Hence, even if we adopt a segmentation method like PMF, it is hard to improve segment pair merging using shared memory because of large length discrepancy. Therefore, performance is obstructed by a large number of global memory accesses.

Since we allocate 256 threads in one GPU block and the distribution of docIDs is approximately uniform, a GPU block might need to search through a range containing

$$256/(0.054\%) \approx 470000$$

docIDs, much larger than the shared memory on each SM. If a GPU block runs out of memory on one SM, the number of active warps will be reduced and the parallelism will be less effective. Therefore, we set the goal of reducing the dependency on global memory access.

**Table 4: Length ratio $R$ as intersection proceeds**

| | Number of terms | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | **2** (16.1%) | | **3** (24.5%) | | **4** (22.8%) | | **5** (14.8%) | | **6** (8.24%) | |
| $x$ | $R$ **(%)** | $\|\bar{\ell}\|$ **(K)** | $R$ **(%)** | $\|\bar{\ell}\|$ **(K)** | $R$ **(%)** | $\|\bar{\ell}\|$ **(K)** | $R$ **(%)** | $\|\bar{\ell}\|$ **(K)** | $R$ **(%)** | $\|\bar{\ell}\|$ **(K)** |
| **1** | 13.6 | 8.6 | 23.1 | 8.7 | 28.6 | 8.4 | 32.0 | 8.0 | 34.4 | 8.0 |
| **2** | - | - | 0.97 | 2.0 | 1.76 | 1.6 | 2.38 | 1.5 | 2.82 | 1.5 |
| **3** | - | - | - | - | 0.22 | 0.8 | 0.35 | 0.6 | 0.50 | 0.6 |
| **4** | - | - | - | - | - | - | 0.087 | 0.5 | 0.13 | 0.4 |
| **5** | - | - | - | - | - | - | - | - | 0.054 | 0.4 |

# D. PARAPFOR ALGORITHM

## D.1 Compression

We consider a single segment of length $s$ consisting of the integers $G = (g_0, g_1, \ldots, g_{s-1})$, where we assume that $0 \le g_i < 2^{32}$ for all $i \in \{0, 1, \ldots, s-1\}$. The parameter $s$ should be chosen to best suit the hardware available; in this paper we choose $s$ to be the number of threads in one GPU block. The least significant $b$ bits of the value $g_j$ are stored in $l_j$. The index of the $j$-th exception is stored in $i_j$ and the overflow is stored in $h_j$. We assign the variables:

- $b$ as described in Section 5.1,
- the width of every slot $i_j$ is `ib` $:= \lceil \log(\max_j i_j) \rceil$,
- the width of every slot $h_j$ is `hb` $:= \lceil \log(\max_j g_{i_j}) \rceil - b$, and
- `en` is the number of exceptions.

These four pieces of information are stored in a header.

## D.2 Decompression

The process of decompression is split into three distinct stages.

1. Thread $j$ reads $l_j$ and stores it in shared memory.
2. Thread $j$ reads $i_j$ and stores it in a register.
3. Thread $j$ reads $h_j$ and recovers the $j$-th exception $g_{i_j}$ by concatenating $h_j$ and $l_{i_j}$.

Only the first `en` threads will be used in Stages 2 and 3. This process is illustrated in Figure 10 when the exceptions happen to be $g_0$, $g_1$, and $g_{s-2}$.

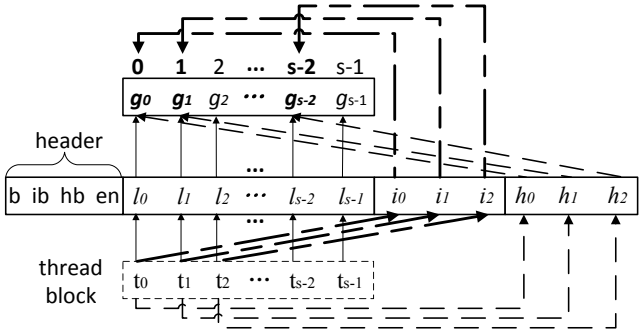


**Figure 10: ParaPFor decompression**

Algorithm 2 describes ParaPFor decompression. Global memory access is required in Lines 3, 4, 8, and 9. Since all the numbers in the algorithm are at most 32-bit wide, each thread accesses global memory between 2 and 4 times.

**Algorithm 2** *ParaPFor Decompression*

**Input:** Compressed segment $G'$ in global memory
**Output:** $(g_0, g_1, \ldots, g_{s-1})$ in shared memory
1: **for** *each thread* **do**
2: $j \leftarrow$ threadIdx.x (i.e. assign the $j$-th element to the current thread)
3: Extract $b$, `ib`, `hb`, and `en` from header
4: Extract $l_j$
5: $g_j \leftarrow l_j$
6: __synchthreads()
7: **if** $j <$ `en` **then**
8: Extract $i_j$
9: Extract $h_j$
10: $g_{i_j} \leftarrow g_{i_j} \mid (h_j << b)$ (i.e. concatenate $h_j$ and $l_{i_j}$)
11: **end if**
12: __synchthreads()
13: **end for**

# E. THEORETIC ANALYSIS OF LRC

The aim of this section is to give a theoretic analysis of LRC, and estimate the compression ratio with respect to different length of inverted lists.

Inverted lists $\ell(t)$ are generated by random processes outside of our control. Here we will assume that $\ell(t)$ is chosen uniformly at random from all ordered lists of length $n$ and with elements belonging to $\{1, 2, \ldots, m\}$. This assumption will be more reliable when e.g. the docIDs are renumbered at random. Let $X_i$ be the random variable for the docID of the $i$-th document in the inverted list. There are $\binom{k-1}{i-1}\binom{m-k}{n-i}$ sorted lists for which the $i$-th element is $k$ (we choose $i-1$ elements less than $k$ to place before $k$ and $m-k$ elements greater than $k$ to place after $k$). Hence

$$\Pr(X_i = k) = \frac{\binom{k-1}{i-1}\binom{m-k}{n-i}}{\binom{m}{n}}.$$

Using the binomial identities $\binom{k}{i} = \frac{k}{i}\binom{k-1}{i-1}$ and

$$\sum_{k=1}^{m} \binom{k}{i}\binom{m-k}{n-i} = \binom{m+1}{n+1},$$

we can find that

$$\mathrm{E}(X_i) = \sum_{k=1}^{m} k \Pr(X_i = k) = \frac{m+1}{n+1} i.$$

One can draw a regression line for $\ell(t)$ :

$$f(i) := \frac{m+1}{n+1} i,$$

so that $\mathrm{E}(X_i) = f(i)$ for all $1 \le i \le n$. We use this line to approximate the linear regression line.

We claim that a docID in an inverted list can be compressed in $\lceil \log(t) + 1 \rceil$ bits provided the probability of the vertical difference between the regression line and the point $(i, X_i)$ greater than some $t > 0$ is smaller than a sufficiently small value $\epsilon$. So we will focus on finding a bound of the form:

$$\Pr[|X_i - f(i)| \ge t] = \Pr[|X_i - \mathrm{E}(X_i)| \ge t] < \epsilon. \quad (4)$$

Chebyshev's Inequality implies that, for all $t > 0$,

$$\Pr[|X_i - \mathrm{E}(X_i)| \ge t] \le \frac{\mathrm{Var}(X_i)}{t^2}. \quad (5)$$

However, (5) is too loose for our purposes. So next we will show how to improve the upper bound of (5). For all $r > 0$ and $t > 0$,

$$\Pr[|X_i - \mathrm{E}(X_i)| \ge t] = \Pr[|X_i - \mathrm{E}(X_i)|^r \ge t^r],$$

so by Markov's Inequality,

$$\Pr[|X_i - \mathrm{E}(X_i)| \ge t] \le \frac{\mathrm{E}[|X_i - \mathrm{E}(X_i)|^r]}{t^r}.$$

This inequality will enable us to improve the bound based on the $r$-th central moment.

Let $(x)_p = x(x+1)\cdots(x+p-1)$ denote the rising factorial function, and $x^{\underline{p}} = x(x-1)\cdots(x-p+1)$ denote the falling factorial function.

**Lemma E.1**

$$\mathrm{E}[(X_i)_p] = (i)_p \frac{(m+1)_p}{(n+1)_p}.$$

PROOF. Let $X_i^{m,n}$ denote the random variable $X_i$ for given parameters $m$ and $n$. Then

$$\begin{aligned}&\Pr(X_i^{m,n} = k) \cdot (k)_p \\ &= \Pr(X_{i+p}^{m+p,n+p} = k+p) \cdot (i)_p \frac{(m+1)_p}{(n+1)_p}.\end{aligned}$$

After summing over $k \in \{1, 2, \ldots, m\}$, the left-hand side becomes $\mathrm{E}[(X_i^{m,n})_p]$, while the right hand,

$$\sum_{k=0}^{m} \Pr(X_{i+p}^{m+p,n+p} = k+p) \cdot (i)_p \frac{(m+1)_p}{(n+1)_p} = 1 \cdot \frac{(m+1)_p}{(n+1)_p}.$$

∎

Next, we will give the formula of the central moment of $X_i$ of any order.

**Lemma E.2** *For $0 \le i \le n$,*

$$\sum_{i=0}^{n} (x)_i (-1)^{n-i} \left\{ {n \atop i} \right\} = x^n,$$

*where $\left\{ {n \atop i} \right\}$ is the Stirling number of the second kind.*

PROOF. Apply $x \to -x$ in the identity $\sum_{i=0}^{k} x^{\underline{i}} \left\{ {n \atop i} \right\} = x^n$. ∎

**Theorem E.3** *The $r$-th central moment $\mathrm{E}[|X_i - \mathrm{E}(X_i)|^r]$ is*

$$\mathrm{E}(X_i)^r + \sum_{l=1}^{r} (-1)^{r-l} \binom{r}{l} \sum_{j=0}^{l} (-1)^{l-j} \left\{ {l \atop j} \right\} \mathrm{E}[(X_i)_j] \mathrm{E}(X_i)^{r-l}.$$

PROOF. By the Binomial Theorem,

$$[X_i - \mathrm{E}(X_i)]^r = \sum_{l=0}^{r} (-1)^{r-l} \binom{r}{l} X_i^l \mathrm{E}(X_i)^{r-l}.$$

Now apply Lemma E.2 to $X_i^l$. ∎

We will use $r = 22$ to find a bound of the form (4), since the 22-nd central moment is a fairly high order to give us a relatively accurate bound value. By Theorem E.3 and Lemma E.1,

$$\mathrm{E}[|X_i - \mathrm{E}(X_i)|^2] = \frac{i(1+m)(1-i+n)(m-n)}{(1+n)^2(2+n)}. \quad (6)$$

For fixed $m$ and $n$ satisfying $m \ge n$, (6) is maximized when $i = \lfloor (n+1)/2 \rfloor$, so also is $\mathrm{E}[|X_i - \mathrm{E}(X_i)|^{22}]$. We will focus on this point next.

As an example, set $\epsilon = 10^{-5}$ in (4) and $m = 2^{24}$ according to the number of documents in the BDR dataset. We list $\lceil \log(\min(t)) + 1 \rceil$ for various $n$ in Table 5, where $\min(t)$ is the least $t$ that satisfies (4). Additionally, we pick some lists with length $n \in \{100\mathrm{K}, 200\mathrm{K}, 400\mathrm{K}, 800\mathrm{K}, 1\mathrm{M}, 2\mathrm{M}\}$ from the compressed index (using LRC algorithm) of BDR dataset. The *average bit-width* of a compressed list is the total number of bits of the list divided by the number of docIDs contained in the inverted list. Table 5 shows that the theoretical bit-width $\lceil \log(\min(t)) + 1 \rceil$ is close to the average bit-width of real lists with respect to different $n$. So if an inverted list consists of randomized docIDs, we can estimate the compression ratio based on the above method.

**Table 5: $\lceil \log(\min(t)) + 1 \rceil$ and average bit-width**

| $n$ | 100K | 200K | 400K | 800K | 1M | 2M |
|---|---|---|---|---|---|---|
| $\lceil \log(\min(t)) + 1 \rceil$ | 18 | 18 | 17 | 17 | 17 | 16 |
| **average bit-width** | 17 | 17 | 17 | 16 | 15 | 15 |

# F. EXPERIMENTAL RESULTS

## F.1 Experimental Platform

A brief overview of the hardware used in the experiments is given in Table 6.

**Table 6: Platform details**

| | |
|---|---|
| **O.S.** | 64-bit Redhat Linux AS 5 with kernel 2.6.18 |
| **CUDA Version** | 3.0 |
| **Host** | |
| **CPU** | AMD Phenom II X4 945 |
| **Memory** | 2GB × 2 DDR3 1333 |
| **PCI-E BW CPU → GPU** | 3.0GB/s |
| **PCI-E BW GPU → CPU** | 2.6GB/s |
| **Device** | |
| **GPU** | NVIDIA GTX 480 (Fermi architecture) |
| **SMs × Cores/SM** | 15 (SMs) × 32 (Cores/SM) = 480 (Cores) |
| **Memory BW** | 177.4GB/s |

## F.2 Compressed Lists

Table 7 compares different algorithms intersecting compressed lists on various datasets. TP denotes the throughput (queries/s) and RT denotes the response time (ms/batch). In the case of ParaPFD, the number in parentheses gives the proportion of exceptions used in the experiment. To obtain a short response time, we set the computational threshold to 1M, which limits the response time of LR based algorithms to under 3ms. The performance of each individual algorithm on the different datasets is similar.

SegLRC's performance is better than LRC and LRCSeg. The reason is that the search range is reduced by the local contraction ratio. We can see that HS256_LRC performs better than all of the other algorithms, although the decompression speed is not as good as the other LR based algorithms (see Figure 8 for a comparison).

**Table 7: Throughput and response time**

| | GOV | | GOVPR | | GOVR | | BD | | BDR | |
|---|---|---|---|---|---|---|---|---|---|---|
| | **TP** | **RT** | **TP** | **RT** | **TP** | **RT** | **TP** | **RT** | **TP** | **RT** |
| **ParaPFD (0.2)** | 15879 | 7.66 | 16069 | 7.57 | 15587 | 7.81 | 58937 | 4.71 | 53952 | 5.15 |
| **ParaPFD (0.0)** | 22998 | 5.29 | 23302 | 5.22 | 22639 | 5.38 | 86000 | 3.23 | 78338 | 3.55 |
| **LRC** | 42336 | 2.88 | 41905 | 2.90 | 43034 | 2.83 | 160570 | 1.73 | 160898 | 1.73 |
| **LRCSeg** | 42347 | 2.88 | 41862 | 2.91 | 43078 | 2.83 | 160958 | 1.73 | 160338 | 1.73 |
| **SegLRC** | 47335 | 2.57 | 46054 | 2.64 | 48078 | 2.50 | 164903 | 1.68 | 176134 | 1.58 |
| **HS256_LRC** | 55955 | 2.18 | 55315 | 2.20 | 55735 | 2.18 | 191120 | 1.45 | 195520 | 1.42 |

# G. REFERENCES


[28] S. Büttcher, C. L. A. Clarke, and I. Soboroff. The TREC 2006 terabyte track. In *Proc. 15th Text Retrieval Conference (TREC 2006)*, 2006.

[29] R. Fisher and F. Yates. *Statistical Tables for Biological, Agricultural and Medical Research*. Oliver and Boyd, 1963.

[30] NVIDIA Corporation. NVIDIA CUDA Programming Guide v3. 2010.

[31] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Computer Networks and ISDN Systems*, 30(1-7):107-117, 1998.

[32] E. M. Voorhees. Overview of trec 2004. In *In NIST Special Publication 500-261: The Thirteenth Text Retrieval Conference Proceedings (TREC 2004)*, pages 1–12, 2004.

[33] E. M. Voorhees. Overview of TREC 2002. In *Proc. 11th Text Retrieval Conference (TREC 2002)*, pages 1–16, 2003.