Implementing Random Indexing on GPU

Lukas Polok and Pavel Smrz, {ipolok,smrz}@fit.vutbr.cz, Brno University of Technology, Faculty of Information Technology, Bozetechova 2, 61266 Brno, Czech Republic

Keywords: GPGPU, term co-occurrence, word space models

Abstract

Vector space models have received a significant attention in recent years. They have been applied in a wide spectrum of areas including information filtering, information retrieval, document indexing and relevancy ranking. Random indexing is one of the methods employing distributional statistics of term co-occurrences to generate vector space models from a set of documents. If the size of the document collection is large, a significant computational power is required to compute the results.

This paper presents an efficient implementation of the random indexing method on GPU which allows efficient training on large datasets. It is only limited by the amount of memory available on the GPU. Various ways to overcome the dependence on the GPU memory are discussed. Speedups in magnitude of tens are achieved for training from random seed vectors, and even much higher figures for retraining. The implementation scales well with both the term vector dimension and the seed length.

1. INTRODUCTION

Vector space models [1] (a.k.a. word space models or term vector models) represent text documents as vectors of terms. A collection of documents can be represented by a document-term matrix. Each entry of the matrix then indicates the presence of a term in a document, the count of its occurrences or a weight characterizing the importance of a given term for a particular document within the collection.

Generalizing the idea, terms themselves can be represented by high-dimensional context vectors. Contexts are given by the co-occurring terms. The rows of the co-occurrence matrix represent unique terms and the columns their context, which can either be documents or other terms, yielding terms-by-documents and terms-by-terms matrices, respectively.

The co-occurrence matrices are usually very sparse. According to the Zipf's law [2], the use of most words in any natural language is limited to a small set of contexts and only a small set of words can be used universally in any context. On the other hand, when increasing the number of documents, the number of terms does not stop growing at some point nor does it approach an asymptote – it grows steadily instead [2]. The direct construction of the co-occurrence matrix is therefore inefficient or even infeasible as the number of documents and terms increases.

Various techniques have been developed to reduce the number of dimensions in co-occurrence matrices and thus to speed up any subsequent computation based on them. For example, one can employ common dimension reduction techniques such as SVD (singular value decomposition) [3]. However, this can be computationally prohibitive for large dimensions and it has other disadvantages as well. First, the direct dimension reduction techniques do not avoid the computation of large term co-occurrence matrices. It then results in high memory demands. Second, once new data is available, it is necessary to update the term co-occurrence matrix (or even to calculate it from scratch if it is not viable to keep it in memory) and to calculate the SVD all over (although it is possible to project new data to a reduced space, it can only introduce a limited amount of new information, or possibly no diverse information at all) [4].

Random indexing [5] is a novel method for calculating context vectors already in a reduced space, therefore effectively avoiding a large co-occurrence matrix and enabling incremental improvements to an existing model once new data is available. The core idea is very simple – each term is assigned a sparse seed vector, containing just ±1's and then the documents are scanned through with a fixed-size window and contexts of unique terms are accumulated based on their occurrences under the window. This was further improved in [6] by taking the relative position of terms into account by involving a permutation of seed vectors, based on their corresponding term positions.

The use of random seed vectors, containing just ±1's is not coincidental as it can be shown that the context vectors representing different contexts are going to be nearly orthogonal. Also, it can be shown that for a D-dimensional vector, there can be just D orthogonal vectors, but many more nearly orthogonal vectors [7]. Therefore, the dimensionality of context vectors produced by random indexing is naturally reduced.

Once the context vectors are evaluated, it may be beneficial to repeat the algorithm once more using just the generated context vectors as the seed vectors. (This is referred to as „retraining“ in the following text.) The reason for this is that if some terms have similar semantic meaning, then contexts generated by those terms should also be similar – a property which is not guaranteed when using the random seed vectors. Near-orthogonality of the resulting context vectors is transitional in this case (use of near-orthogonal seed vectors results again in near-orthogonal context vectors).

This paper deals with an efficient implementation of the random indexing algorithm on GPU. It can be divided into several key stages: pre-processing documents into a form viable for the GPU calculation, optimizing scheduling of computations in order to maintain constantly high load on the GPU, optimizing the retraining stage's speed and memory demands, and, finally, the term vector normalization and the summation into document vectors.

2. RELATED WORK AND BASIC NOTIONS

To the best of our knowledge, there are no GPU implementations of random indexing to date. There are, however, GPU implementations of SVD [8, 9, 10] (among others), which could be used to reduce the dimensionality of the plain term co-occurrence matrix, as described above. The implementation of the random indexing method which inspired our GPU implementation is called Semantic Vectors [11].

The code to be run on the modern GPUs can be written in high-level programming languages. This simplifies the development of the GPU-enhanced algorithms. Nevertheless, one still needs to understand the general characteristics of the GPU computing model to produce a code that can be executed efficiently.

Today's GPUs are based on the SIMT architecture (single instruction multiple threads), natively supporting massive multithreading. There are typically several multiprocessors, each capable of executing several threads in parallel. A group of threads running on a single multiprocessor is called a warp. The number of threads in a warp is referred to as the SIMT width. Threads are scheduled in a hierarchy of local and global blocks. The memory access and the thread divergence present the main bottlenecks in the GPU-based computation.

Memory accesses are most efficient when coalesced: each thread reads one address from a contiguous block aligned on an integer multiple of the local block size. The global memory is not cached on most of the GPUs. If one is unable to make memory accesses coalesced, it might be more efficient to use the texture memory, which is cached under the condition that the reads are spatially local. The new GPUs (NVIDIA Fermi) have a global memory cache which can be more efficient than the texture cache for some tasks. However, the global memory cache behavior is not well documented, so that it might be still more convenient to use the texture memory in specific situations. If neither is possible, one has to make sure that enough threads are scheduled to hide the memory access latency.

Threads are said to be divergent when executing different instructions. It is most commonly caused by branching. If the threads in a single warp are divergent, the execution paths needs to be calculated in sequence and the results masked out. It is therefore beneficial to avoid branching, or, at least, to make sure that all the threads in a single warp take the same branch.

To define the random indexing algorithm, let us begin with a formal definition of inputs and outputs. The set of unique terms (the vocabulary) will be denoted:

$$T = \{t_1, t_2, \cdots, t_M\} \tag{2.1}$$

The input then consists of document vectors containing the terms:

$$\mathbf{d} = \begin{pmatrix} i_1 & i_2 & \cdots & i_N \end{pmatrix}^T \big| \, i_x \in T \tag{2.2}$$

where N is the document vector dimensionality, and it equals the document length, in terms (words). Finally, the matrix of context vectors is given as:

$$\mathbf{T} = \begin{bmatrix} \mathbf{c}_1 & \mathbf{c}_2 & \cdots & \mathbf{c}_M \end{bmatrix}^T \tag{2.3}$$

where rows c_1 through c_M are the context vectors for terms t_1 through t_M. The dimensionality of the context vectors (usually in the order of hundreds or thousands [12, 13, 1]) forms also an input to the algorithm.

To proceed with random indexing, the matrix of seed vectors needs to be defined. It has the same size as the context vector matrix:

$$\mathbf{S} = \begin{bmatrix} \mathbf{s}_1 & \mathbf{s}_2 & \cdots & \mathbf{s}_M \end{bmatrix}^T \tag{2.4}$$

The seed vectors, as mentioned above, are sparse vectors, containing just a few ±1's. They are initialized randomly (thus the name random indexing). The number of the nonzero elements (usually around ten [12]) forms, again, an algorithm input. The task is now to traverse the document vector and to update the context vectors. To do that, a window function, which selects parts of the document, needs to be defined:

$$\begin{aligned} \mathbf{w}(n) &= \begin{pmatrix} w_{n,1} & w_{n,2} & \cdots & w_{n,2s+1} \end{pmatrix}^T \\ &= \begin{pmatrix} i_{n-s} & i_{n-s+1} & \cdots & i_n & \cdots & i_{n+s-1} & i_{n+s} \end{pmatrix}^T \end{aligned} \tag{2.5}$$

where *n* is the window center (the position of the term in focus in the document), and *s* is a half window size. Note that **w** is a *2s+1* dimensional vector. Also note that *s* is a global parameter of the algorithm, not a parameter of the window function. For the sake of simplicity, handling boundary conditions is omitted here (assume $s < n \leq N - s$ always holds). A permutation function, which, given a permutation number and a term index, returns a permutated context vector for that particular term, is further needed [6]:

$$\varphi(p, r) = \begin{cases} \mathbf{s}_r <<< -p & p < 0 \\ \mathbf{c}_r & p = 0 \\ \mathbf{s}_r >>> p & p > 0 \end{cases} \tag{2.6}$$

where operators <<< and >>> denote left and right rotation, respectively.

It is now possible to update the context vector of a term:

$$\mathbf{c}_{w_{n,s+1}}' = \sum_{j=1}^{2s+1} \varphi(j-s-1, w_{n,j}) = \mathbf{c}_{w_{n,s+1}} + \sum_{j=1}^{s} \varphi(-j, w_{n,s-j+1}) + \varphi(j, w_{n,s+j+1}) \quad (2.7)$$

The first part of the sum corresponds to the left half of the window and the second part of the sum corresponds to the right one. Note that the term in focus in the center of the window comes through the permutation function unchanged. The algorithm for calculating the term vectors then takes the following form:

```
1. Initialize T to zeros
2. Initialize rows of S to random sparse vectors
3. For every possible window position in the document
    4. Apply Equation 2.7
5. Normalize context vectors
```

Algorithm 2.1: Building term vectors

Extending the algorithm to handle multiple documents is trivial, it just involves repetition of step 3 for each document, as well as minor modifications to the window function (Equation 2.5) so it selects terms from the right document.

3. DATA STRUCTURES

As mentioned above, documents are represented by vectors of terms (2.2). In the context of parallel processing, it is not reasonable to work directly with the documents as they vary in length. It is beneficial to define a new data structure – a chunk. A chunk is a vector of terms, not unlike the document, but it has a constant length, chosen appropriately for an efficient processing. Chunking refers to the process of splitting documents into chunks. The chunk is a vector of term id's, l_{chunk} long:

$$\mathbf{C} = \begin{pmatrix} c_1 & c_2 & \cdots & c_{l_{chunk}} \end{pmatrix} \quad (3.1)$$

When dividing one document into multiple chunks, one has to make sure that each chunk is self-contained. Since the data are scanned by the sliding window (2.5), it is necessary to repeat s terms from the end of each chunk at the beginning of the next one.

When joining more documents in a single chunk, it is necessary to separate documents in such a way that the last term in a document doesn't affect the context of the first term in the following one. For this reason, a dummy term is introduced. It is a new virtual term with id $M+1$ (2.1) which carries null contextual information (its seed vector $\mathbf{s}_{M+1}$ is a null vector). Documents are simply padded by s occurrences of the dummy term. The need to check the boundary conditions established for the window function (2.5) can also be avoided by padding the beginning of the first document and the end of the last document with s occurrences of the dummy term. Under these circumstances the following relationships can be established:

$$l_{docs} = \sum_{D} N_d \quad (3.2)$$

$$l_{padded\,docs} = l_{docs} + s(D+1) \quad (3.3)$$

$$N_C = \left\lceil \frac{l_{padded\,docs} - 2s}{l_{chunk} - 2s} \right\rceil \quad (3.4)$$

where D is the number of documents, N_d is the length of document d and N_C is the number of the chunks. It can be seen that for $l_{chunk} >> s$, N_C becomes just the ratio of the sum of document lengths to the chunk size (the ideal efficiency). Also note that l_{chunk} needs to be chosen so that it is greater than $2s$. Chunking is a very simple operation with complexity linear to the total length of all the documents.

For efficient processing on a GPU, it is useful to have a list of occurrences of all the terms in a chunk. Such a list contains a set of positions of occurrences for each particular term:

$$o_i = \{x \mid c_x = t_i\} \quad (3.5)$$

$$\mathbf{O} = \begin{pmatrix} o_1 & o_2 & \cdots & o_M \end{pmatrix} \quad (3.6)$$

It can be expected that some of **O**'s elements are going to be empty sets, because either there are more terms in total than l_{chunk} or the terms are distributed unevenly across the documents. In practice, there is a small group of terms much higher frequent than the rest of the terms [4] so **O** is sparse.

Constructing the occurrence list is simple as well, it is possible to use a hash table to map term id's to the occurrence list and a simple vector of vectors to store the occurrence positions.

4. SPARSE VECTOR IMPLEMENTATION ON GPU

The naive algorithm (algorithm 2.1) is not a good candidate for parallelization as it results in memory access conflicts once multiple threads process the same term. To avoid the conflicts, a special attention needs to be paid to the order in which the window positions are inspected. The algorithm can simply employ the term occurrence list and process each term separately:

```
1. Initialize T to zeros
2. Initialize rows of S to random sparse vectors
3. For each chunk C
    4. Calculate term occurrence list O
    5. For each term $t_i$
        6. For every position in $o_i$
            7. Apply equation 2.7
8. Normalize context vectors
```

Algorithm 4.1: Building term vectors

The algorithm contains several nested loops. It also involves the calculation of **O**. When implemented sequentially on a CPU, it is slightly slower than algorithm 2.1. However, it is possible to schedule multiple threads, each to process a part of loop 5. The memory access conflicts are avoided as every thread deals with its own term vector only.

When implementing on a GPU, it is beneficial if the threads running in a single warp have the same execution path. This is rather simple to achieve for the loop evaluating equation 2.7 (step 7 in algorithm 4.1) since the window size is constant. On the other hand, the outer loop (step 6 in algorithm 4.1) will have a different number of passes in each thread as it can be expected that each term will have a different number of occurrences in the chunk. This inevitably leads to the thread divergence.

A potential remedy lies in sorting **O** by the number of occurrences (the size of o_i) so that the most frequent terms are processed by the first group of threads and the least frequent terms by the last group of threads. This order is better than its reverse because it can be expected that the number of terms is not going to be an integer multiple of the SIMT width, and it is better to have idle threads when processing terms with a small number of occurrences rather than the opposite. Even if the individual terms have different frequencies, the frequencies are now similar among the threads in each multiprocessor and the computational efficiency is much higher.

In experiments with real data, however, it turned out that the most frequent terms are much more frequent than the average terms. It resulted in a high slack and the need to subdivide the work to fine-grained work-items. The new schema for the division of occurrence sets o_i to passes, slices and work-items is demonstrated in Figure 4.1.

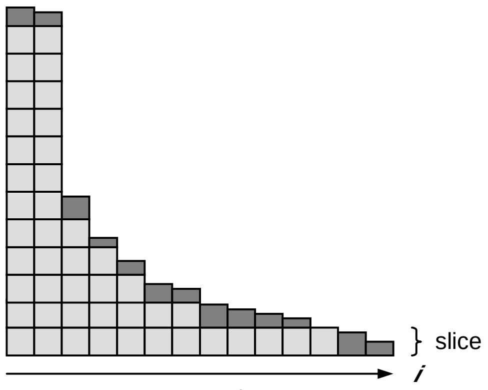


Figure 4.1: The division of term occurrence sets into slices and work-items

Each column contains a term occurrence set o_i, the arrow denotes the direction in which i grows. Each term occurrence set is subdivided into constant-length work-items, represented by the light gray blocks, and remaining work-items, the dark gray blocks. The rows define the slices. The horizontal scale is referred to as the slice size (the number of the terms processed in the particular slice). The vertical scale is referred to as the length (the work-item length is the number of the term occurrences in the work-item; the slice length is the maximal length of the work-items in the slice).

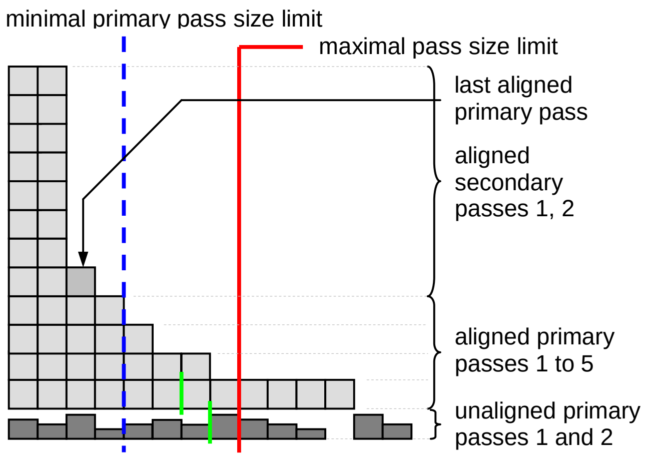
The time to process a single work-item is proportional to the product of the work-item length and the window size. As GPU threads have a limited amount of time to run, the slice length has to be set carefully so that they do not timeout.

When assigning the work-items to the passes, the maximum number of work-items is limited by the maximum thread block size. The maximum pass size depends on the memory and the register usage per thread. Some slices may exceed the maximum pass size. They will need to be further subdivided.

It also needs to be decided how the remainder work-items will be processed. Processing them in each pass as they occur in consecutive slices is not optimal. A smaller standard deviation of the work-item length (and thus a higher efficiency) can be achieved when all the remainder work-items are processed in the first passes. These passes are referred to as unaligned, while the rest of the passes as aligned. For large chunks, it is also possible to minimize the thread divergence by splitting the first passes into several smaller passes containing work-items of equal lengths. For example, over 12,000 terms with remainder slices were observed when working with the King James' Bible [14] dataset (see Section 7). There were 4,000 terms occurring just once, 1,700 terms occurring twice and under 1,000 terms occurring three times.

As mentioned above, there are always a few words much more frequent than the rest of the words in a natural language text (this is illustrated by the first two columns in figure 4.1). An additional optimization step is therefore necessary to avoid many short passes that would lead to a low GPU load. Once the pass size drops below a threshold, the passes are re-arranged. A new dummy term id is assigned to work-items (different from the dummy term used for the chunk padding and the document separation). The work-items are processed in parallel. Results are stored in a separate dummy vector bank. After finishing the computation, the results are added to the original term vectors. Each thread performs an addition of a single term vector element. These passes are called secondary, the preceding ones primary. Vector addition is referred to as the summation step in the following text.

Sometimes there are enough terms with just a single work-item left. To avoid the summation step overhead, it is then more efficient to process them as one last primary pass. The whole scheduling strategy is depicted on figure 4.2. The solid vertical line corresponds to the maximal pass size limit – all the slices crossing this line need to be broken into multiple passes. Two short lines at the bottom demonstrate the division. The dashed vertical line shows the minimal primary pass size limit. Once the slice size drops below this limit, the rest of the work-items are processed in a secondary pass. The dark grey square corresponds to the last work-item. It will be processed in the last aligned primary pass. Note that this is only for the sake of the description brevity – processing a single work-item in a separate pass would be inefficient.



minimal primary pass size limit
maximal pass size limit
last aligned primary pass
aligned secondary passes 1, 2
aligned primary passes 1 to 5
unaligned primary passes 1 and 2

Figure 4.2: Division of work-items into passes

To process the task on a GPU, the work-items need to be passed to the threads. The required data are stored in three buffers: the chunk buffer contains concatenated documents, the occurrence buffer contains all non-empty elements of **O** in a single contiguous array, and the work-item buffer contains a single work-item for each thread to work on. One work-item consists of a term id which points to the term vector buffer, a number of term occurrences (the work-item length) and an offset to the occurrence buffer. The arrangement is shown in figure 4.3.

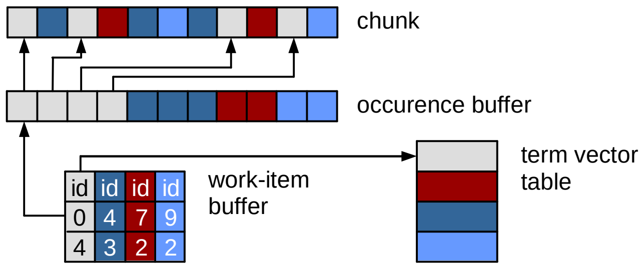


Figure 4.3: Data buffers for GPU kernels

The developed strategy keeps the GPU workload high and is significantly more efficient than the sorted occurrence list approach mentioned above. Nevertheless, there is still the memory bandwidth bottleneck. GPUs perform well if the memory accesses are coalesced. In the described approach, there is no way to coalesce most of the accesses as each thread works with a completely different data. It is possible to coalesce accesses to the occurrence list but it has a minimal impact in terms of the processing time as the occurrence list is accessed in the outer loop only and it needs to be interleaved with the CPU processing as well. It is also possible to employ the GPU constant memory for the work-item buffer if it is small enough. However, it does not make a big difference too – the work-item buffer is accessed once by each thread only. Thus, the only viable option is to schedule a sufficient number of threads so that the GPU has enough work to cope with the memory access latency.

Further optimization steps have been followed. Specialized kernels for the term vectors of power-of-two dimensions enabled using the bitwise AND operator when calculating the seed vector permutation (2.6). The computation takes 1 clock cycle instead of approximately 2 cycles for the modulo operator on the GPU [15]. There are also two versions of each kernel, for the aligned and unaligned passes. The former enables loop unrolling and other compile-time optimizations. When the unrolling is not possible, loops are optimized by using the Duff's device [16]. It results in an almost 4 % speedup.

5. DENSE VECTOR IMPLEMENTATION ON GPU

The previous section described the computation of term vectors from random seed vectors. The seed vectors are sparse and typically rather short. When retraining term vectors from results of the previous pass, however, the sparse seed vectors are replaced by the high-dimensional term vectors. It is therefore possible to schedule a single thread per term vector element to perform the computation in parallel. The complex task division to primary and secondary passes or work-items is no longer needed. As illustrated in figure 5.1, just the occurrence list is sufficient.

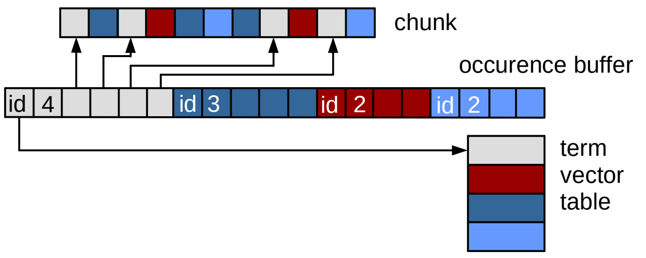


Figure 5.1: Data buffers for retraining kernels

The kernel function gets only the offset of the occurrence buffer and the number of occurrences to be processed. It is still necessary to break long runs into smaller ones to avoid the kernel timeout but it is sufficient to apply a simple approach of the division to pieces with the size not exceeding a given threshold. The memory accesses are coalesced if the dimensionality of the term vectors is an integer multiple of the SIMT width. To achieve the best performance, it is therefore advisable to choose it accordingly.

The only downside of the retraining is the necessity to have all the term vectors in the GPU memory. There are millions of terms in large datasets so that the term vector buffers will easily exceed the GPU memory capacity. The algorithm is, however, only working with the chunks containing just a limited number of terms. Various caching schemata, such as the least recently used algorithm [17], can be applied to dynamically upload and download the term vectors so that only an immediately required subset is present in the GPU memory.

6. TERM VECTOR NORMALIZATION ON GPU

Vector normalization is a simple task. It consists of two steps – calculating the length of the vector and multiplying the vector by its reciprocal length. Both the steps can be realized as reduction operations implemented by processing a single vector at a time. Even though it is the way vector normalization is implemented in popular libraries such as CUBLAS [18], it is inefficient due to idle threads. It may be faster to calculate the length of each vector in a single thread. If there are enough vectors to keep the GPU busy, the operation is efficient. The downside is that the memory accesses are not coalesced. A better strategy is to calculate the length of each vector in a thread block, utilizing the shared memory and the read coalescing. The results of the three approaches described above and for CUBLAS function `cublasSnrm2` are compared in figure 6.1 (denoted VAT (vector at a time), VPT (vector per thread), VTB (vector per thread block), and Snrm2 (CUBLAS `cublasSnrm2`).

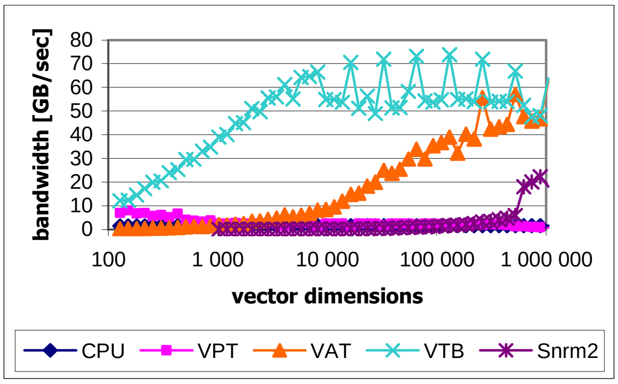


Figure 6.1: GPU vector reduction bandwidth

All the tests were run with an equal amount of data, only the vector dimensions (and hence the number of vectors) changed. CPU time is therefore almost constant. It can be seen that the vector per thread-block strategy is the most beneficial. It surpasses `cublasSnrm2()`from the standard CUBLAS library by more than 40 GB/sec.

Vector scaling is also a very simple operation. The implementation can be straightforward but various optimizations can be applied too. The simplest version calculates a single multiplication of an element by the appropriate vector scale per thread. The corresponding bandwidth is rather low (about 30 GB/sec). Profiling reveals a large number of uncoalesced reads caused by reading vector scales from the global memory. The vector scales can be also stored in the constant memory. It slightly improves the performance but it is still well under the GPU capacity. The best implemented kernel pre-fetches the scales to the local memory, yielding the bandwidth of 70 GB/s. The performance of the CUBLAS function `cublasSscal()`is suboptimal again. The graph in figure 6.2 summarizes the results. Note that “the spikes” in the graph are caused by the element misalignment occurring when the size of the vectors does not correspond to the power of two.

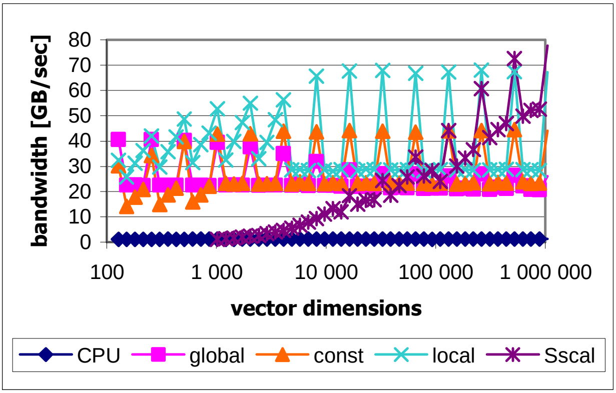


Figure 6.2: GPU vector scaling bandwidth

7. RANDOM INDEXING SPEEDUPS

Two datasets were used for the experiments described in this section – the King James’ Bible dataset [14] and a part of the English GigaWord [19], specifically the “Central News Agency of Taiwan” (CNA). The limited size of the former one (the King James’ Bible) makes it ideal for experimenting with different configurations of the algorithms. At the same time, the dataset is large enough to fill a single 4MB chunk, so scaling to larger datasets can be expected to be approximately linear. The latter dataset is a collection of about 7 million newspaper articles, approximately 15 GB in size. The reason for using just a part of this dataset was to reduce the number of the terms so they would fit in the GPU memory. Additionaly, the dataset was lemmatized using an existing library (libturglem-0.2 [20]), reducing the number of terms down to 97,335 (in the CNA part of the dataset alone). Both datasets were indexed using CLucene library [21] for easier processing.

A simple "C" implementation of the algorithm was originally developed to verify GPU results, but as it turned out it is considerably fast, it was included in the results. The experiments compared the runtimes of our “C” implementation, of the Semantic Vectors package (version 1.3) [11], and of the GPU implementation. Times needed for loading data from disk or storing the results were left out. The GPU times include copying data to the GPU memory as well as copying the results back. To ensure all the GPU operations have finished, the `clFinish()`function was always called.

The recorded values include the times of term vector training and retraining on the real data, and of term vector normalization on synthetic data of varying dimensionality. Note that the term vector normalization time is marginal compared to the calculation of the term vectors. For the sake of brevity, the total times are not reported.

The implementations were built using the Microsoft Visual Studio 2008 compiler. The CPU time of all the algorithms was measured on a pair of unloaded AMD Opteron 2360 SE processors (8 cores running at 2.6 GHz in total) running Windows XP x64. The GPU time was measured on NVIDIA GeForce GTX 260, using the most recent drivers. An average time of four runs for each test is given.

The first series of tests regards calculating term vectors from sparse seeds on the King James' Bible dataset. The seed length (the number of nonzero elements in the seed vector) was set to 10 and then to 100. The term vector dimensionality varied from 256 to 4,096. The chunk size was set to four megabytes for all the tests. The GPU implementation ran with the maximal slice length of 32 occurrences, the maximal pass size 12,500 terms (given by the maximal number of scheduled threads), the minimal primary pass size 2,000 terms and the minimal last primary pass size 200 terms. The dummy vector banks were allocated to hold 10,000 vectors. The results are shown in figure 7.1.

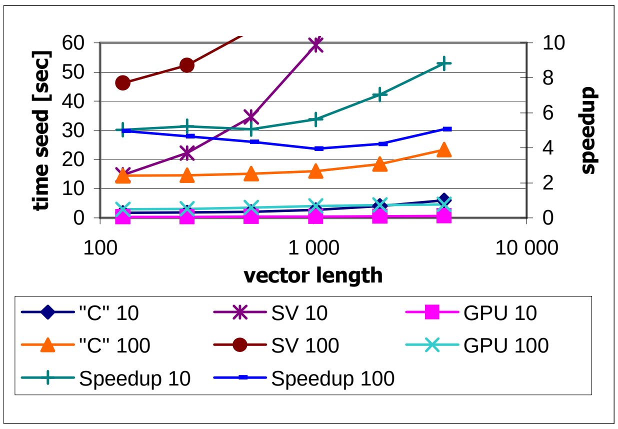


Figure 7.1: Building term vectors from sparse seeds, seed lengths 10 and 100 (note "SV" denotes Semantic Vectors)

The second series of tests involved retraining term vectors from the results of the first pass (using the same dataset). The term vector dimensionality varied from 256 to 4,096 as in the first test. The chunk size was set to 4MB again. The maximal slice length for the GPU implementation was set to 1,024 occurrences. The results can be seen in figure 7.2. The excellent scaling of the GPU implementation is noteworthy. It can be expected to keep linear up to 16,384-dimensional vectors – the limit of threads running at a time (512 threads per block × 32 processors). Note also that the Semantic Vectors package is not much slower than the "C" implementation in retraining, at least for short vectors. This is probably caused by the similar order of processing used in both implementations.

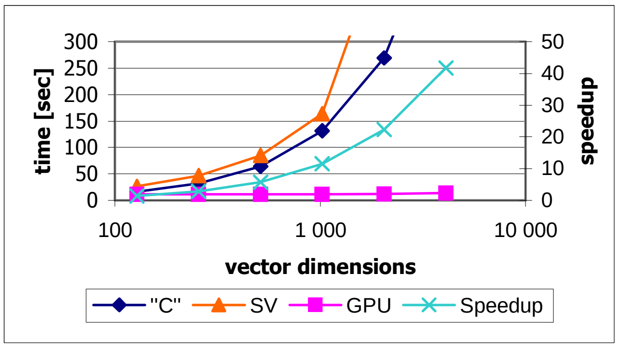


Figure 7.2: Retraining term vectors from results of the previous pass

The third test deals with the term vector normalization speedup of GPU, compared to CPU. Term vector dimensions vary from 128 to 1,048,576. The most efficient method for both vector reduction and scaling was always chosen. The results are shown in figure 7.3. Again, note the spikes are caused by the misalignment of the non-power-of-two dimensional vectors.

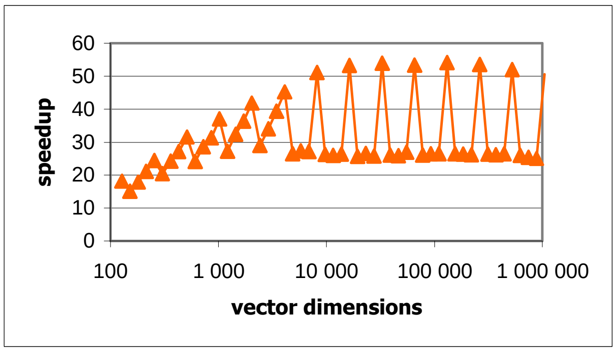


Figure 7.3: Vector normalization speedup (GPU compared to "C" implementation)

The last series of tests were conducted on the GigaWord dataset. The values of the internal parameters for the algorithms were calculated using a simple decision tree which attempts to suggest optimal parameters. The term vector dimensionality was set to 2,048, the window size and the seed length both to 10. Table 7.1 summarizes the results. The lower speedup can be explained by the much higher number of terms (97,335 in English GigaWord, 17,000 in King James' Bible) and thus more irregular memory access patterns. That could be remedied by introducing a minimal term frequency limit to reduce the number of terms. No tests were run using the SemanticVectors package as it proved to be significantly slower than the "C" implementation.

Table 7.1: Term vector calculation times and speedup

	Calculating TV	Retraining TV
Time "C"	5434.128 s	27766.092 s
Time GPU	1725.120 s	4738.241 s
Speedup	3.15 ×	5.86 ×

8. CONCLUSIONS AND FUTURE DIRECTIONS

A practical tool for calculating term context vectors and document context vectors from documents indexed by the CLucene [21] library was implemented. It can be easily modified to use another library, as data structures passed between CLucene and the algorithms are rather simple. The GPU implementation is significantly faster than both the Semantic Vectors package and the baseline "C" implementation. The speedup factor ranges from 4 to 9 for building term vectors from sparse seed vectors while a larger speedup occurs with shorter seed vectors. The tests used a minimal seed vector length of 10, which is practical [12] for most applications; a 9-fold speedup can therefore be seen as realistic. For retraining from the results of the previous pass, speedups are typically much higher, for 4,096 dimensions, it exceeds 40-fold. Speedup of the parallel vector normalization is up to 70-fold, but the time spent in this stage of the algorithm is negligible.

The implementation is limited by the amount of memory available on the GPU, as it requires storing all the term vectors and eventually also the seed vectors in the GPU memory. Today's top GPUs are able to hold hundreds of thousands to millions of term vectors, which should be sufficient unless the processed dataset is extremely large. Implementation of an algorithm for swapping unused term vectors from GPU memory to host memory is suggested. That would enable processing of virtually unlimited number of terms.

Another limitation is the maximal number of scheduled threads and the thread timeout, which may vary from one generation of GPUs to the next. it can be easily solved by adjusting algorithm parameters, such as pass size or slice length, without any significant performance loss.

The main disadvantage of this implementation is its GPU memory limitation. The next work will be directed towards implementing the proposed scheme of term vector caching, effectively removing this limitation.

One component of the algorithm – the random seed vector generator – was left unoptimized,. Generating pseudo-random numbers on GPUs was described by several authors [22, 23, 24]. It usually takes several seconds to generate random seed vectors using standard "C" libraries, so it probably could be optimized. But, in the end, it is still only a small amount of time, compared to the total algorithm run time.

Another possible place for optimization could be to attempt to improve locality of the memory accesses when retraining the term vectors. The measure of similarity between the terms is already available, giving some degree of information about the term vectors, which are going to be referenced. It should therefore be possible to reorganize the order in which the term vectors are processed based on clustering in high-dimensional context space to achieve better memory access locality.

ACKNOWLEDGEMENTS

The research leading to these results has received funding from the European Community's 7th Framework Programme FP7/2007-2013 under grant agreement n.215453 – WeKnowIt and Artemis JU grant agreement n. 100223 – eSonia.

References

[1] G. Salton, A. Wong, C. S. Yang, "A Vector Space Model for Automatic Indexing", Communications of the ACM, vol. 18, nr. 11, pages 613–620, 1975

[2] G. K. Zipf, "Human Behaviour and the Principle of Least Effort", Addison-Wesley, 1949

[3] G. H. Golub and C. Reinsch, "Singular value decomposition and least squares solutions", Numerische Mathematik 14 (5): pp 403–420, 1970. doi:10.1007/BF02163027

[4] Magnus Sahlgren, "An Introduction to Random Indexing", Methods and Applications of Semantic Indexing Workshop at the 7th International Conference on Terminology and Knowledge Engineering, TKE 2005, 2005

[5] P. Kanerva, "Sparse distributed memory", The MIT Press, 1988

[6] Sahlgren, M., Holst, A. & Kanerva, P., "Permutations as a Means to Encode Order in Word Space", Proceedings of the 30th Annual Meeting of the Cognitive Science Society (CogSci'08), July 23-26, Washington D.C., USA, 2008

[7] Hecht-Nielsen, R.; "Context vectors; general purpose approximate meaning representations self-organized from raw data", in Zurada, J. M.; R. J. Marks II; C. J. Robinson, "Computational intelligence: imitating life". IEEE Press, 1994

[8] Sheetal Lahabar, P. J. Narayanan, "Singular Value Decomposition on GPU using CUDA", IEEE International Parallel Distributed Processing Symposium, 2009

[9] J. Krüger, R. Westermann, "Linear algebra operators for GPU implementation of numerical algorithms", proceeding of SIGGRAPH '05 ACM SIGGRAPH Courses, 2005

[10] V. Volkov, J. Demmel, “LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs”, 2008

[11] D. Widdows, K. Ferraro, “Semantic Vectors: A Scalable Open Source Package and Online Technology Management Application”, In Proceedings of the sixth international conference on Language Resources and Evaluation, 2008

[12] P. Kanerva, J. Kristofersson, and A. Holst, “Random indexing of text samples for latent semantic analysis”, in Proceedings of the 22nd Annual Conference of the Cognitive Science Society, page 1036. Erlbaum, 2000

[13] K. Lund, C. Burgess, “Producing high-dimensional semantic spaces from lexical co-occurrence”, Behavior Research Methods, Instruments, & Computers, 28, pages 203–208, 1996

[14] "The Bible, King James Version Complete Contents", available online at http://www.gutenberg.org/ebooks/7999, 2004

[15] “NVIDIA OpenCL Programming Guide”, available online at http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_OpenCL_ProgrammingGuide.pdf, 2010

[16] Stroustrup, Bjarne, “The C++ Programming Language, Third Edition”, Addison-Wesley, ISBN 0-201-88954-4, 1997

[17] E. J. O’Neil, P. E. O’Neil, G. Weikum, “The LRU-K page replacement algorithm for database disk buffering”, In Proceedings of the 1993 ACM SIGMOD international conference on Management of data, pages 297-306, 1993

[18] “NVIDIA CUBLAS User Guide”, available online at http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUBLAS_Library.pdf, 2010

[19] D. Graff, C. Cieri, “English Gigaword”, Linguistic Data Consortium, Philadelphia, 2003

[20] “European language lemmatizer“, availbale online at http://lemmatizer.org, 2010

[21] M. McCandless, E. Hatcher, O. Gospodnetic, “Lucene in Action, Second Edition“, Manning Publications Co., ISBN 1-933-98817-7, pages 328-332, 2010

[22] W. B. Langdon, “PRNG Random Numbers on GPU”, 2007

[23] Wai-Man Pang; Tien-Tsin Wong; Pheng-Ann Heng, “Generating massive high-quality random numbers using GPU“, Evolutionary Computation, CEC 2008, 2008

[24] A. Zafar, M. Olano, “Tiny encryption algorithm for parallel random numbers on the GPU”, Proceedings of the 2009 symposium on Interactive 3D graphics and games, 2009