Best-effort Semantic Document Search on GPUs

Surendra Byna
NEC Laboratories America
Princeton, NJ
sbyna@nec-labs.com

Jiayuan Meng
University of Virginia
Charlottesville, VA
jiayuan@gmail.com

Anand Raghunathan
Purdue University
West Lafayette, IN
raghunathan@purdue.edu

Srimat Chakradhar
NEC Laboratories America
Princeton, NJ
chak@nec-labs.com

Srihari Cadambi
NEC Laboratories America
Princeton, NJ
cadambi@nec-labs.com

ABSTRACT

Semantic indexing is a popular technique used to access and organize large amounts of unstructured text data. We describe an optimized implementation of semantic indexing and document search on manycore GPU platforms. We observed that a parallel implementation of semantic indexing on a 128-core Tesla C870 GPU is only 2.4X faster than a sequential implementation on an Intel Xeon 2.4GHz processor. We ascribe the less than spectacular speedup to a mismatch in the workload characteristics of semantic indexing and the unique architectural features of GPUs. Compared to the regular numerical computations that have been ported to GPUs with great success, our semantic indexing algorithm (the recently proposed Supervised Semantic Indexing algorithm called SSI) has interesting characteristics – the amount of parallelism in each training instance is data-dependent, and each iteration involves the product of a dense matrix with a sparse vector, resulting in random memory access patterns. As a result, we observed that the baseline GPU implementation significantly under-utilizes the hardware resources (processing elements and memory bandwidth) of the GPU platform. However, the SSI algorithm also demonstrates unique characteristics, which we collectively refer to as the "forgiving nature" of the algorithm. These unique characteristics allow for novel optimizations that do not strive to preserve numerical equivalence of each training iteration with the sequential implementation. In particular, we consider best-effort computing techniques, such as dependency relaxation and computation dropping, to suitably alter the workload characteristics of SSI to leverage the unique architectural features of the GPU. We also show that the realization of dependency relaxation and computation dropping concepts on a GPU is quite different from how one would implement these concepts on a multicore CPU, largely due to the distinct architectural features supported by a GPU. Our new techniques dramatically enhance the amount of parallel workload, leading to much higher performance on the GPU. By optimizing data transfers between CPU and GPU, and by reducing GPU kernel invocation overheads, we achieve further performance gains. We evaluated our new GPU-accelerated implementation of semantic document search on a database of over 1.8 million documents from Wikipedia. By applying our novel performance-enhancing strategies, our GPU implementation on a 128-core Tesla C870 achieved a 5.5X acceleration as compared to a baseline parallel implementation on the same GPU. Compared to a baseline parallel TBB implementation on a dual-socket quad-core Intel Xeon multicore CPU (8-cores), the enhanced GPU implementation is 11X faster. Compared to a parallel implementation on the same multi-core CPU that also uses data dependency relaxation and dropping computation techniques, our enhanced GPU implementation is 5X faster.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GPGPU '10, March 14, 2010 Pittsburgh, PA, USA.
Copyright (c) 2010 ACM 978-1-60558-935-0/10/03 $10.00

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel Programming

Keywords

GPGPU, Document Search, Supervised Semantic Indexing, CUDA, Dependency Relaxation, Best-effort Computing

1. INTRODUCTION

The emergence of General Purpose Graphics Processing Units (GPGPUs) is fueling acceleration of a wide range of highly parallel and compute-intensive workloads. Programmability of GPUs was limited to APIs such as OpenGL that were designed for graphics workloads. In recent years, new programming frameworks for GPGPUs, such as Nvidia's Compute Unified Device Architecture (CUDA) [9][10], AMD's Brook++ [12], *etc*. have eased the difficulty of programming the highly parallel GPU platforms. However, the process of developing *efficient* GPU implementations is highly application dependent. While some applications are inherently data parallel, there are many that require significant re-structuring of the algorithms and programs to realize high performance on GPUs. In this paper, we focus on parallelizing Supervised Semantic Indexing (SSI) [1], a popular Recognition and Mining (RM) workload.

SSI is used to rank documents in a corpus or database based on their semantic similarity to a given text query. The algorithm searches for a direct association between the words contained in a document and its similarity score, taking into account correlations between words due to synonymy and polysemy. SSI has been

demonstrated to have state-of-the-art performance in searching large datasets such as Wikipedia with very good accuracy [1][3].

A common feature of RM workloads, including the SSI algorithm, is that they perform the same computation operation on enormous amounts of data in an iterative fashion in order to develop a model. Many studies have demonstrated that RM workloads are highly data parallel, and can easily take advantage of data parallel hardware such as GPUs. Although this claim is true for many RM workloads, we show that straightforward parallel implementations of the SSI algorithm do not take full advantage of abundant parallelism offered by today's GPU platforms. In our previous work [7][8], we proposed best-effort computing and data dependency relaxation to improve the performance of parallel implementations on multicore CPUs. In this paper, we expand the applicability of these strategies to throughput-oriented manycore architectures such as GPUs, and apply them to improve the parallel performance of SSI. We describe how best-effort computing strategies can be customized for GPU architectures, and believe that our approach can be extended to other RM applications that demonstrate less-than-optimal degrees of parallelism.

In section 2, we explain the SSI training algorithm. In section 3, we describe a baseline implementation of SSI on GPUs and motivate the problem of low utilization of the GPU's hardware resources, which leads to poor performance. Section 4 presents the unique characteristics of SSI training and Sections 5, 6, and 7 propose techniques to improve the scalability of SSI on GPUs. In section 8, we present the experimental results and evaluate performance of the GPU implementation. In section 9, we discuss related work and conclude in section 10.

2. BACKGROUND

The SSI training algorithm develops a model from TFIDF values of each word in the dictionary. This model is used for searching a given text query. TFIDF is the product of Term Frequency (TF) and Inverse Document Frequency (IDF). TF is the number of times a word occurs in the document divided by the total word count of the document, and IDF is the reciprocal of the ratio of all documents in the corpus that contain the word (thus IDF reduces the importance of commonly occurring words).

Suppose there are N words in the dictionary. A query or document in a corpus of documents contains a subset of words in the dictionary. Since documents only contain a small fraction of all possible words in the dictionary, if we represent a document with a vector, each vector is sparse and contains non-zero elements only for the words that appear in the document. The training phase of SSI learns an internal weight matrix U that translates the presentation of each document from a lengthy and sparse TFIDF vector to a short and dense *semantic vector*. The semantic vector represents the likelihood of a document in a number of machine-learned conceptual categories (C). The dot products of the semantic vectors are then used to calculate the semantic similarity between their corresponding documents.

When the developed model is used for document search, documents are ranked and retrieved based on their similarity with the query document. We now give more details on how the model is learned.

SSI training model is learned in a similar way to an artificial neural network (ANN). The input nodes are the TFIDFs of all words, each connected directly to all output nodes that represent a predefined number of conceptual categories to be learned. The ANN can be represented as a bias vector b with a length of C and a weight matrix U sized N X C, where N is the number of words in the vocabulary and C is the number of conceptual categories. The i^{th} column of U stores each word's weight in contributing to the classification of the i^{th} conceptual category.

The model is refined iteratively, where each training iteration involves three TFIDF vectors representing a query document (q), a labeled relevant document (d), and an irrelevant document (r). Figure 1 provides pseudo-code for the algorithm involved in SSI training. There are three operations in each training iteration:

Input: A corpus of documents with their TFIDF values, learning rate (ε), error threshold, margin criteria (m)
Output: A weighted matrix (U) sized NXC.

Algorithm:
Test error = 1.0
Initialize U with random float values
While (Test error > error threshold)
  Randomly select a query (q), a relevant document (d), and an irrelevant document (r) from the corpus
  /* *Forward propagation:*/
  $s_q = q^T U + b^T$
  $s_d = d^T U + b^T$
  $s_r = r^T U + b^T$
  If ($s_q.s_d - s_q.s_r$) < m
  /* If similarity score between q and d, is not greater than a specified margin criteria m, then calculate gradient vectors to modify the model /*
    $g_q = s_d - s_r$,
    $g_d = s_q$
    $g_r = -s_q$
  End If
  If $g_q > 0$ or $g_d > 0$ or $g_r > 0$
  /* *Backward propagation* */
    $U = U + \varepsilon(q \times g_q^T + d \times g_d^T + r \times g_r^T)$
    $b = b + \varepsilon(g_q + g_d + g_r)$
  End If
End While

Figure 1: The pseudo-code of the SSI algorithm

1. Forward propagation calculates of three *semantic vectors*, each of which is calculated as a weighted sum of TFIDFs for each conceptual category using one of the three documents (q, d, and r).
2. *Comparing Relevancy*: The similarity scores of d and r to q are calculated as $s_q.s_d$ and $s_q.s_r$, respectively, where "." denotes the dot product. The similarity score between the relevant document (d) and the query document (q) is

supposed to be greater than that between the irrelevant document (r) and the query (q). If $s_q.s_d$ is not greater than $s_q.s_r$ by a specified threshold, then, the model has to be adjusted. To adjust the model, three feedback gradient vectors (g_q, g_d, and g_r) are calculated.

3. If the gradient vectors are non-zero, weight matrix (U) and bias vector (b) are modified using q, d, r, and the gradient vectors. Each document's TFIDF vector is multiplied by the corresponding gradient vector to generate a gradient matrix, which is scaled according to a predefined learning rate $\mathcal{E}$, and added to U.

There are two types of dependencies among successive iterations if documents in successive iterations share the same indices of any non-zero TFIDFs. A Read-after-Write (RAW) dependency exists if data updated during the former iteration is read in the forward propagation of the latter iteration. A Write-after-Write (WAW) dependency exists between two successive update operations on same data.

The SSI training converges when specified threshold accuracy is achieved. The training algorithm tests accuracy after a certain number of iterations, called an *epoch* (typically 10,000 iterations).

In this study, we use the Wikipedia corpus [3] consisting of 1.8 million documents to train the model. The number of words in the vocabulary (N) is 30,000 and the number of conceptual categories (C) is 200. This number of categories is sufficient for this model [1]. Related documents are automatically labeled according to the links provided by Wikipedia: a document is tagged as relevant to all documents to which it has links, and it is assumed to be irrelevant to the remaining documents. A pre-processing step calculates TFIDF values for each word.

2.1 GPU Memory Hierarchy

We briefly introduce the memory hierarchy of the Nvidia Tesla C870 GPU that we used in our study.

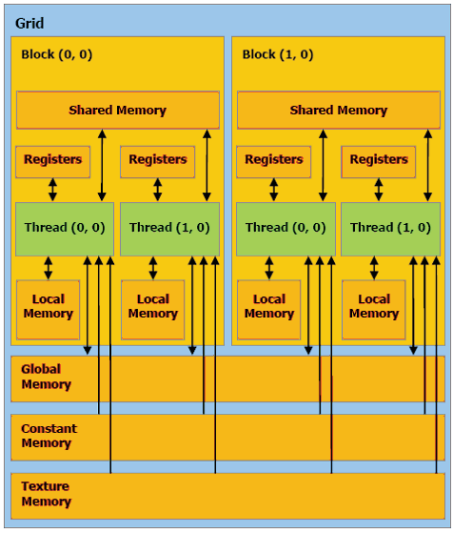

Figure 2: Memory Hierarchy of Nvidia GPUs [21]

Figure 2 shows the memory hierarchy in Tesla GPUs. The registers are local to each thread. The shared memory is shared by all the threads running on a single multiprocessor. A read-only constant memory is shared by all the threads in the texture processing cluster (TPC), which contains two multiprocessors in the Tesla C870 GPUs. A read-only texture cache is also shared by all the threads in the TPC. The C870 GPU has 1.5GB global memory with a theoretical peak memory bandwidth of 77 GB/s. More details of the GPU memory hierarchy and computation capabilities can be found in [21].

3. MOTIVATION

In this section, we illustrate the performance of SSI training on the GPU, where the forward and backward propagation operations in each training iteration are parallelized. We then analyze performance and show that this implementation does not utilize full potential of the GPUs.

3.1 Baseline Implementation on the GPU

In the baseline SSI implementation on the GPU, the weight matrix resides in the GPU memory and queries are sent to the GPU for performing forward and backward propagation. We first transfer the large U matrix (22.5 MB) that contains the initial values of the model to GPU global memory. Document vectors (q, d, and r) are transferred into the constant memory of the GPU for each training instance. Accesses to the constant memory have a small latency of 1 cycle. Since the document vectors q, d, and r are sparse vectors, in the forward and backward operations, although the weight matrix (U) is as large as N X C, only a few rows that correspond to non-zero TFIDFs in the queries are accessed and/or modified. The remaining values are untouched since the result of multiplication with zero values in the documents is a zero and does not affect the weights in the U matrix.

Following the original implementation of the SSI training algorithm (as shown in Figure 1), the forward and backward propagation operations are performed sequentially, but each contains computations that may be parallelized sequentially. We parallelize the three multiplications between sparse vectors and the dense matrix (U) in the forward propagation. These multiplications are different from the sparse matrix-vector multiplications [22][23], where the matrix is sparse. In SSI training, the vectors are sparse, and the weight matrix is dense. Each vector is split into multiple parts that are distributed to multiprocessors of the GPU. The partial vector-matrix multiplication results corresponding to each multiplication are reduced into one dot product. The intermediate results of the forward propagation, the semantic vectors, are brought to the host CPU for comparing relevance values, *i.e.*, dot products of s_q and s_d, and s_q and s_r. If non-zero gradient vectors are generated on the CPU, they are transferred to the constant memory of GPU. and the backward propagation is then performed on GPU.

Table 1: Experimental Setup

	CPU	GPU
Model	Intel Xeon E5420	Tesla C870
Cores	8 (two sockets)	128
Frequency	2.5 GHz	1.35 GHz
Memory size	12 GB	1.5 GB
Threading API	Pthreads, TBB	CUDA 2.3
Compiler	gcc -O3	nvcc 2.3 –O3
OS	64-bit Linux 2.6.18-164.el5	

We tested various ways to split the document vectors and we achieved the best performance when the vectors are split into 8 parts. Since we have three "matrix – sparse vector" multiplications in both forward and backward propagations, when we split each vector into 8 parts, we start 24 blocks on the 16 multiprocessors of C870 GPU. Each block contains 64 threads. We tested the training implementation on a heterogeneous workstation consisting of an Intel Xeon quad-core CPU and a Tesla C870 GPU. Table 1 shows the details of the architecture.

3.2 Performance of Baseline Implementation

Figure 3 compares the performance of a sequential implementation of the training algorithm on the CPU and the baseline implementation on the GPU for executing 100 epochs (an epoch is 10,000 iterations of training). In the legend, C denotes CPU, and G denotes GPU. C→G represents the data transfer from the CPU to the GPU and G→C represents the data transfer from the GPU to the CPU. We use the same notations throughout the paper to show the CPU to GPU data transfers and vice versa. The CPU version of SSI is implemented in C and the GPU version with CUDA 2.3. We saw that the GPU implementation was only 2.4X faster than the CPU implementation. The results after each epoch for both implementations are numerically equivalent. Upon further investigation of the relatively modest performance improvement on the GPU, we noticed that the GPU was only utilizing 20% of its peak memory bandwidth. Moreover, the parallelism available within a forward propagation of a training iteration is often very low, which depends on the number of non-zero elements of the documents fed to the iteration. The computation to data transfer ratio is often close to 1, which is very low for offloading tasks to GPUs. In other words, all the processor cores of the GPU were kept busy, but for a very short time, which is often less than the data transfer time. For documents with less than 50 words, the GPU kernel invocation overhead is also higher than the computation time on the GPU.

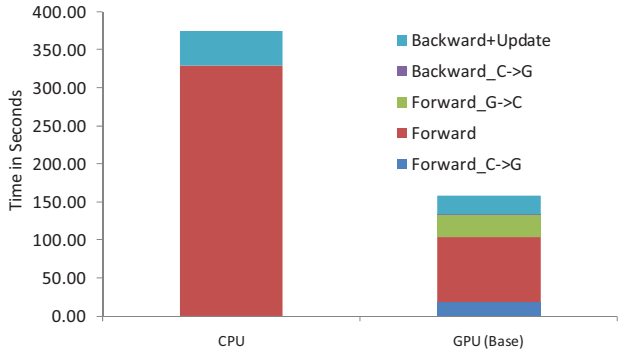

Figure 3: Performance of Conventional Implementation on CPU and GPU.

From these results, it is clear that there is scope for further performance optimization. Parallelization within an iteration is limited by the number of non-zero values in the sparse vectors. For iterations that have very few non-zero values, the GPU is not utilized and the overhead in kernel invocations and data transfer calls dominates, causing significant performance degradation. To improve the performance of GPU implementation, we apply various techniques that are described in the following subsections.

4. UNIQUE WORKLOAD CHARACTERISTICS OF SSI

Many RM applications follow a pattern of computation in which a model is iteratively refined based on a sequence of input training instances. Conventional parallel implementations usually exploit parallelism only within each iteration due to data dependencies across iterations, which results in poor parallel scalability when each training instance consists of few data elements. This is because the performance gains benefited from parallel execution cannot sufficiently offset the overheads in setting up the parallel computation. In other words, the problem size for parallel execution is not large enough for scalable performance. Ideally, the problem size for data parallelism should scale according to the number of training instances in RM algorithms. However, iterative RM algorithms such as SSI limit the problem size of data parallelism within each training instance due to potential dependencies between iterations. For instance, in the case of SSI algorithm, there are RAW and WAW dependencies. To study this issue, we discuss three features that we observe in SSI.

Write Sparsity: One of the characteristics of SSI that can be leveraged to improve the available parallelism is the sparsity of writes in individual iterations. After the first few iterations, only 28% of the iterations perform any updates at all in the backward propagation operation, *i.e.*, there are only 28% of iterations that need to run serially. This feature is important since it offers the potential for concurrent execution of multiple iterations.

Low spatial locality of writes among iterations: Iterations that perform back-propagation rarely update the same part of the model and therefore data races would be rare if they are parallelized. This is because the *q, d*, and *r* vectors are sparse and their non-zero TFIDFs spread across the vocabulary of 30,000 words. Our trace-based characterization shows that on average, only 3.6% of *U* is visited in one iteration, and every four iterations overlap only 23% of their updates.

Error Tolerance: As mentioned earlier, RM applications exhibit a *forgiving nature* in their execution. For example, after 10,000 iterations, the learned models of two implementations may be numerically different. However, they are both acceptable if their resulting accuracies similar. From another perspective, we can always continue to train the model until a desired accuracy is achieved. Therefore, we can measure the time it takes for different implementations to reach the desired accuracy and select the implementation that takes the least time to reach the desired accuracy. As long as the algorithm is converging faster, the occasional inaccuracies introduced in a portion of the model are acceptable.

We now discuss the techniques that we used to exploit these inherent features of SSI and utilize the parallelism offered by the GPUs.

5. DEPENDENCY RELAXATION

As discussed earlier, there are dependencies between successive iterations of the SSI algorithm. In our baseline GPU implementation, we exploited parallelism only within each

iteration. If we assume that there are no dependencies, we can run multiple training iterations concurrently. This increases the number of threads running on the GPUs, which will keep the processor cores busy performing computation operations. However, we cannot blindly drop dependencies. We use the unique characteristics of SSI we described earlier, such as Write Sparsity and Low locality of updates, to judiciously drop dependencies. We call this strategy *dependency relaxation*.

The goal of dependency relaxation is to run as many iterations as possible concurrently with an insignificant accuracy loss. Figure 4 illustrates the idea of dependency relaxation. On the left, we show the conventional method of running SSI, where iterations are executed serially and the model is being modified by each iteration. By relaxing dependencies, multiple (k) iterations run concurrently and modify the model. Because of the Low spatial locality feature of multiple iterations, the chance of modifying the same location of data by concurrent iterations is low. This strategy utilizes the computing power of parallel hardware more efficiently and achieves significant performance gains. Although the reordered updates resulted from parallelizing the iterations lead to a model that is not numerically equivalent to that obtained from the conventional method, there is hardly accuracy loss due to the error-tolerate nature of SSI.

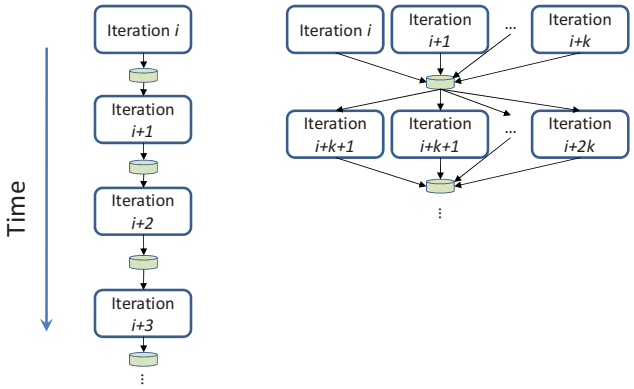

Figure 4: Relaxing dependency to run multiple iterations concurrently

The number of training iterations that we can run concurrently is limited by various architectural features of the GPUs. When we implemented the data dependency on multicore CPUs [8], the limitation was the number of cores, memory bandwidth, and cache coherence traffic. In the case of GPUs, which have are many data parallel (SIMD) cores and have very small read-only cache memories, the limitations vary. We observe that the error tolerance, memory bandwidth due to more data accesses when multiple iterations are executed concurrently, and the requirement for constant memory with more concurrent iterations are GPU specific limitations. As we increase the number of concurrent training iterations, the possibility of overlapping modifications to the model increases. The write operations on GPUs are much more costly than on CPUs because there are no cache memories and have to be written back to the memory. The amount of constant memory, where we place the document vectors for faster access is also another limiting factor. More concurrent iterations require more document vectors and if they do not fit in the constant memory, they have to be placed in the GPU memory and the cost of accessing the memory is 100 times slower than accessing data from the constant cache.

6. DROPPING COMPUTATION

As mentioned earlier, RM applications do not require different implementations to be numerically equivalent after each iteration. Even if an implementation requires more iterations than others to reach the same accuracy, as long as a certain threshold accuracy is achieved in less time than others, the approach is still preferable. We exploit this feature by dropping some non-critical computation that have less impact on the model constructed by the SSI training algorithm.

The words in vectors that have low TFIDF weight are non-critical. Many document vectors contain most commonly used words in vocabulary, such as “the”, “of”, “and”, “a”, “in”, *etc*., which have no discriminative value and hence do not skew the model if dropped from the training computations. To ensure that these words have some representation in the model, we start dropping these words after a few initial iterations.

Our tests show that dropping these computations has negligible impact and the additional number of iterations required to reach the same accuracy as the original algorithm is minimal. On the other hand, the performance improvement achieved due to this computation dropping is significant. While this strategy is application dependent, its impact on the performance of SSI is non-trivial.

Dropping some of the words in document vectors facilitate storing more documents in the constant memory. As we mentioned in the previous section, the constant memory is very limited on the GPUs, and limits the number of concurrent iterations that can be executed. With fewer words, we can run more iterations concurrently.

7. OPTIMIZING DATA TRANSFERS

In GPU based heterogeneous computing, the GPU is typically connected to the host CPU using the PCI Express bus. They work in different physical memories and address spaces. Therefore, to offload any computing to the GPU, the host CPU has to first allocate and transfer data used by the GPU to the GPU memory and then invoke GPU computation kernels. The GPU results have to be transferred back to the CPU, if the CPU needs the results. The cost of these data transfers is not negligible, and sometimes dominates the computation time on the GPU. Hence, it is often advisable to have a larger computation to communication cost ratio to utilize the power of GPUs.

SSI training involves significant data transfer overhead. In the training algorithm, we can transfer the initial weight matrix on to the GPU once in the beginning and transfer it back to the CPU after the threshold accuracy criteria is met. While these two transfers are not costly, in each iteration, we also need to transfer documents q, d, and r to the GPU and retrieve semantic vectors after each iteration. We also observed that the *cudaMemcpy* call itself has a constant overhead, which is often the dominant portion of communication, especially when the document vectors are sparse and of small size. In our experiments with the SSI algorithm, we observe that the data transfer cost for 8 concurrent forward propagation iterations (with data dependency relaxation)

in an epoch of 10,000 iterations is 84% of the total execution time.

To counter this performance problem, we used a pack-and-transfer strategy. During the pack phase, we reorganize the vectors into a temporary buffer and attach the metadata information at the beginning of the package. We send the packed data to the GPU with one data transfer call. The GPU uses the metadata to identify the boundaries of the documents in its computations. Figure 5 illustrates an example for sending data for running two forward propagation iterations concurrently. These iterations require six documents in total. If they were transferred from the CPU to the GPU individually, six *cudaMemcpy* calls are required. Instead, if we pack them into one buffer, with metadata as a header, then all the data can be sent to the GPU with one *cudaMemcpy* call.

In our packing strategy, we consider data coalescing restrictions of GPUs. Threads in these GPUs are executed in groups of 32, called *warps*. The global memory bandwidth is used most efficiently when the simultaneous memory accesses by threads in a half-warp (first or second half of the threads in a warp) can be coalesced into a single memory instruction of 32, 64, or 128 bytes. If data is coalesced, multiple threads can utilize data fetched by one thread. Although the latest NVIDIA GPUs have less stringent restrictions on data coalescing, the Tesla C870 GPU we used has the restriction that the size of a vector has to be a multiple of 64 or 128 bytes and the k^{th} thread in the half-warp must access the k^{th} word in the fetched data block. To accommodate this, during our packing phase, we pad the temporary buffer and the metadata specifies the boundaries of useful data.

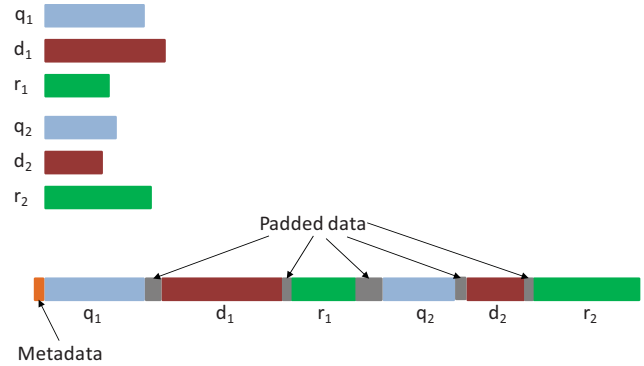

Figure 5: Packing data to reduce the number of data transfer calls

8. EXPERIMENTAL EVALUATION

We now analyze performance with various optimizations that we proposed in the previous section. We first present the results for SSI training with dropping non-critical computation and data dependencies, and data access optimizations. In our SSI training experiments, we use the same Wikipedia corpus as mentioned earlier consisting of 1.8 million documents to train the model. The number of words in the vocabulary (N) is 30,000 and the number of conceptual categories (C) is 200. We set the desired output error for the training process to be less than 1%. Since the error rate fluctuates with training instances, we stop its execution when the error rate remains less than 1% for at least 10 consecutive epochs.

8.1 Dropping Noncritical Computation

We can drop the words in query (q), relevant (d), and irrelevant (r) documents with TFIDF value less than a certain threshold. This reduces the computation overheads on data that does not significantly impact the quality of the model.

Figure 6 shows the performance of SSI training on GPU by dropping words that have TFIDFs less than 0.05 (labeled *GPU (TFIDF < 0.05)* in the figure) and those less than 0.09 (labeled *GPU (TFIDF < 0.09)* in the figure). In our experiments, we varied the threshold for dropping words from 0.01 to 0.10, in increments of 0.01. In the figure we only present results for the case where it has peaked (*i.e.,* 0.05) and where the performance degradation is significant (i.e., 0.09). 4.8% of the words were dropped in each document query on average for the case where TFIDFs are less than 0.05, and 8.3% of words were dropped for the latter case (*i.e.* when TFIDF < 0.09). The performance improvement over the GPU base case with TFIDF < 0.05 is 2X and that with TFIDF < 0.09 case is 1.7X. The performance gain is lower with the latter case because the number of training iterations that is required is 6.2% (roughly 500 epochs) higher. In other words, the performance gain per training instance with the latter case is not substantial enough to offset the increased number of iterations.

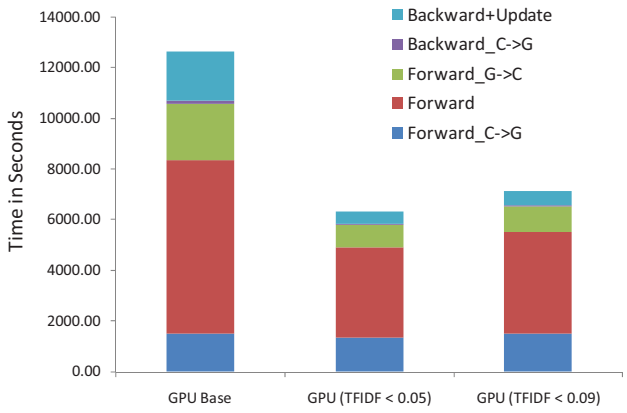

Figure 6: Performance Results with dropping non-critical computation

8.2 Data Dependency Relaxation and Data Access Optimization

While dropping computations and the associated data accesses improves performance, it is also critical to ensure that sufficient parallel threads are present to utilize the large number of cores in GPUs and hide memory access latency. Because the pruned training instances become smaller, more training instances can be executed in parallel to saturate the GPU bandwidth. On top of dropping non-critical computation, we create additional parallelism in our implementation of SSI training by "relaxing" data dependencies and running multiple training instances concurrently.

Figure 7 shows the performance with running 2, 8, and 10 training iterations concurrently (in addition to dropping computation), and

compares the results with the performance when computation is dropped but no data dependency relaxation is performed. All the bars in the graph represent experiments, where words with TFIDFs less than 0.05 are skipped. We optimized data transfer by packing multiple vectors related to the three documents and send the data in one transfer. For instance, 6 documents need to be sent to the GPU when two iterations are batched, 24 documents with a batch size of 8, and 30 documents with a batch size of 10. We pack these multiple documents into one data transfer with meta-data at the header that tells the boundaries between documents.

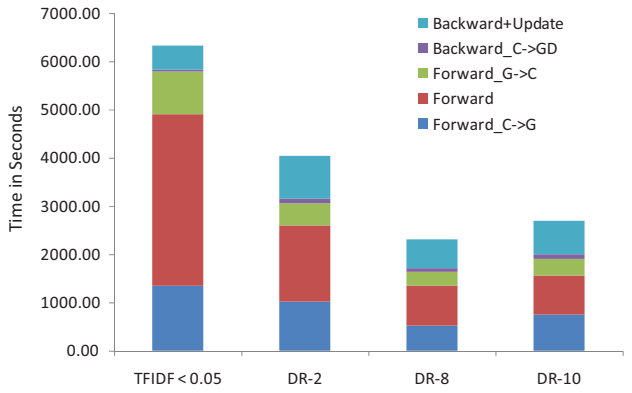

Figure 7: Performance Results with dependency relaxation. These results take into account the effect of dropping computation, where words with TFIDFs less than 0.05 are skipped. The improvement, compared with the baseline implementation, is 3.1X with DR-2, 5.5X with DR-8, and 4.7X with DR-10.

From the figure, we can see that performance peaks when the number of concurrent training instances is 8. When we try more than that, (*e.g.,* 10 concurrent instances), not only the time for forward and backward operations within each training instance, but the number of instances required to reach the threshold error also increases. We observed that the memory bandwidth reaches its peak with the 8 concurrent instances case. When we batch more instances to run concurrently, the GPU's memory bandwidth is overwhelmed and this causes some slowdown. Moreover, with a batch size of 10 or more, the concurrent training instances disrupt each other's results so frequently that the error tolerant nature of the algorithm is no longer able to self-correct the error sufficiently fast. As a result, it requires far more iterations to reach the threshold accuracy. With more than 10 concurrent iterations, all the documents do not fit in the constant memory of the GPU. This results in storing documents in the global memory. In our tests, we observe that running 12 iterations concurrently performs worse than when no data dependency relaxation was applied, (*i.e.* the case where only non-critical computation was skipped).

Overall, for SSI training, we obtain 5.5X performance improvement over the conventional parallel implementation on GPUs and 14.2X over the sequential implementation on the CPU. We applied the data dependency relaxation and dropping computation strategies in implementing a parallel version on a two-socket x86 based quad-core CPU (with 8 cores in total) [8]. We implemented the CPU version using Intel TBB. This achieved 3.1X compared to the sequential implementation on the CPU. Overall the best manycore GPU implementation performs 4.6X faster than the best multicore CPU version.

9. RELATED WORK

Supervised Semantic Indexing [1] is relatively novel semantic analysis algorithm and our implementation is the first on GPUs. Supervised document search has been used in Supervised Latent Semantic Indexing [19], but implementations have only been reported on CPUs. A few studies exist that implement other semantic document search algorithms such as Latent Semantic Analysis (LSA) [14][15]. These implementations optimize Singular Value Decomposition, the compute intensive part of the LSA algorithm, by using CUBLAS [11] for matrix-vector and vector-vector multiplication operations. As explained in the paper, the characteristics of the SSI algorithm are quite different and require different optimization strategies for GPU implementation. In our work, we target SSI training, where we explore many novel strategies based on the concept of best-effort computing to improve available parallelism and reduce data transfer overheads.

GPUs have been used for other Machine Learning applications [20]. However, our study exploits the unique characteristics of SSI application to improve its performance. Many implementations of sparse matrix-vector multiplication with optimized use of GPUs and auto-tuning have been proposed [22][23][24]. In SSI training, the document vectors are sparse and the weight matrix is dense. Our optimizations are designed to span across multiple iterations instead of optimizing just one sparse vector-matrix multiplication.

Chaotic relaxation [17] and asynchronous iterations [18] study relaxing data dependencies in iterative stencil loops. These techniques skip data exchanges across iterations that are used to synchronize the iterations in order to reduce the synchronization costs. The chaotic relaxation strategy has also been used on the GPUs [16] to improve performance of stencil loops. In our data dependency relaxation strategy, we address a different problem, where we increase parallelism through letting any occasional data races to modify the model. In addition to dependency relaxation, we use dropping non-critical computing to exploit error tolerance of the SSI algorithm and optimize data transfers.

Our prior work [7] proposed a best-effort, parallel computing framework, and iterative-convergence programming model with built-in mechanisms to specify best-effort computing strategies. In [8], we proposed data dependency relaxation, computation dropping, and error mitigation techniques for RM algorithms. While all our prior work was proposed for improving utilization of multicore CPUs, in this paper, we extended the best-effort strategies for GPUs. We tuned these strategies specifically for GPUs, where we considered the lack deep cache memory hierarchies in the GPUs and optimized data transfers between the CPU and the GPU.

10. CONCLUSIONS AND FUTURE WORK

In this paper we presented techniques for improving performance of the SSI algorithm on GPUs. After finding that straightforward CUDA implementation of SSI does not utilize the GPUs efficiently, we studied the unique characteristics of the algorithm and devised strategies for improving its performance. The unique

characteristics include – the amount of parallelism in each training instance is data- dependent, each iteration involves the product of a dense matrix with a sparse vector, and the algorithm has a large degree of inherent error resilience. Exploiting these application characteristics, we propose strategies for data dependency relaxation, dropping computation, and for optimizing data transfers between CPUs and GPUs. The application of these strategies enhanced the performance of SSI on GPUs by a factor of 5.5X compared to its straight-forward implementation. The GPU optimized SSI, which utilizes optimizations tuned specifically for the GPUs, also performs 5X faster compared to an implementation on multicore CPUs using the best-effort techniques. We are exploring the atomic operation support provided by CUDA in the latest GPUs to increase the number of concurrent training iterations while avoiding race conditions.

11. ACKNOWLEDGMENTS

The authors would like to gratefully acknowledge Bing Bai for helping us understand the SSI algorithms and providing the reference sequential implementation, and Jonathan Cohen for his advice in utilizing GPUs efficiently.

12. REFERENCES

[1] Bing Bai, Jason Weston, David Grangier, Ronan Collobert, Kunihiko Sadamasa, Yanjun Qi, Olivier Chapelle, Kilian Weinberger, "Supervised semantic indexing", Proceedings of the 18th ACM conference on Information and knowledge management (CIKM), November 2009.

[2] A. Jaleel and M. Mattina and B. Jacob, "Last-level cache (LLC) performance of data-mining workloads on a CMP--A case study of parallel bioinformatics workloads", HPCA 2006.

[3] Gleim, R. and Mehler, A. and Dehmer, M., "Web Corpus Mining by instance of Wikipedia", Web as Corpus, 2007.

[4] Chen, Y. K. and Chhugani, J. and Dubey, P. and Hughes, C. J. and Kim, D. and Kumar, S. and Lee, V. W. and Nguyen, A. D. and Smelyanskiy, M. and Smelyanskiy, M., "Convergence of Recognition, Mining, and Synthesis Workloads and Its Implications", In Proceedings of the IEEE, Vol. 96, No. 5, pp. 790-807, 2008.

[5] Pradeep Dubey, "A Platform 2015 Workload Model: Recognition, Mining and Synthesis Moves Computers to the Era of Tera", White paper, Intel Corporation, 2008.

[6] Chu, Cheng T. and Kim, Sang K. and Lin, Yi A. and Yu, Yuanyuan and Bradski, Gary R. and Ng, Andrew Y. and Olukotun, Kunle, "Map-Reduce for Machine Learning on Multicore", In NIPS 2006, pp. 281-288, 2006.

[7] Jiayuan Meng and Srimat Chakradhar and Anand Raghunathan, "Best-Effort Parallel Execution Framework for Recognition and Mining Applications", IPDPS 2009.

[8] Jiayuan Meng and Srimat Chakradhar, Anand Raghunathan, and Surendra Byna, "Exploiting the Forgiving Nature of Applications for Scalable Parallel Execution", to appear in IPDPS 2010.

[9] Nvidia, CUDA documentation: http://www.nvidia.com/object/cuda_develop.html

[10] Nvidia, "CUDA SDK Code examples", http://www.nvidia.com/object/cuda_get.html

[11] Nvidia, "CUBLAS Library", http://developer.download.nvidia.com/compute/cuda/1_0/CUBLAS_Library_1.0.pdf

[12] AMD, "AMD Stream SDK User Guide v 2.0", 2009.

[13] Intel, *Intel Threading Building Blocks 2.2*, http://www.threadingbuildingblocks.org/

[14] Sean Ahern, David Bremer, John Johnson, Holger Jones, et al., "Applications Kernels on Graphics Processing Units: An Analysis of Hidden Markov Models, Support Vector Machines, Hyperspectral Imaging, and Latent Semantic Indexing", High Performance Embedded Computing Workshop (HPEC 2005), September 2005.

[15] J. M. Cavanagh, T. E. Potok, and X. Cui, "Parallel Latent Semantic Analysis using a Graphics Processing Unit", Proceedings of the 2009 Genetic and Evolutionary Computation Conference, July, 2009.

[16] S. Venkatasubramanian and R. Vuduc, "Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems", Proceedings of the 23rd international conference on Supercomputing (ICS), June 2009.

[17] D. Chazan and W. Miranker, "Chaotic Relaxation", Linear Algebra and its Applications, Vol. 2 No 2: 199-222, 1969.

[18] A. Frommer, D. Szyld, "On asynchronous iterations", Journal of Computational and Applied Mathematics, v.123 n.1-2, p.201-216, Nov. 2000.

[19] Jian-Tao Sun , Zheng Chen , et al., "Supervised Latent Semantic Indexing for Document Categorization," Proceedings of the Fourth IEEE International Conference on Data Mining, p.535-538, November 01-04, 2004.

[20] Steinkraus, D. Buck, I. Simard, P.Y., "Using GPUs for machine learning algorithms", Eighth International Conference on Document Analysis and Recognition (ICDAR 2005), 2005.

[21] Nvidia, "Nvidia Tesla C870 GPU Computing Processor Board Specification", http://www.nvidia.com/docs/IO/43395/C870-BoardSpec_BD-03399-001_v04.pdf

[22] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors", Proc. Supercomputing 2009

[23] F. Vazquez, E. M. Garzon , J. A. Martinez, J. J. Fernandez, "The sparse matrix vector product on GPUs", Technical Report, University of Almeria, June 2009.

[24] J. Choi, A. Singh, R.Vuduc. "Model-driven autotuning of sparse matrix-vector multiply on GPUs." In Proc. Symp. Principles and Practice of Parallel Programming (PPoPP), 2010.