GPU-Accelerated Text Mining

Yongpeng Zhang, Frank Mueller
North Carolina State University
Department of Computer Science
Raleigh, NC 27695-7534
yzhang25@ncsu.edu,
mueller@cs.ncsu.edu

Xiaohui Cui, Thomas Potok
Oak Ridge National Laboratory
Computational Sciences and Engineering
Division Oak Ridge, TN 37831
cuix@ornl.gov

ABSTRACT

Accelerating hardware devices represent a novel promise for improving the performance for many problem domains but it is not clear for which domains what accelerators are suitable. While there is no room in general-purpose processor design to significantly increase the processor frequency, developers are instead resorting to multi-core chips duplicating conventional computing capabilities on a single die. Yet, accelerators offer more radical designs with a much higher level of parallelism and novel programming environments.

This present work assesses the viability of text mining on CUDA. Text mining is one of the key concepts that has become prominent as an effective means to index the Internet, but its applications range beyond this scope and extend to providing document similarity metrics, the subject of this work. We have developed and optimized text search algorithms for GPUs to exploit their potential for massive data processing. We discuss the algorithmic challenges of parallelization for text search problems on GPUs and demonstrate the potential of these devices in experiments by reporting significant speedups. Our study may be one of the first to assess more complex text search problems for suitability for GPU devices, and it may also be one of the first to exploit and report on atomic instruction usage that have recently become available in NVIDIA devices.

Keywords

Text Mining, GPGPU, CUDA

1. Introduction

Graphics programming units (GPUs) differ from general-purpose microprocessors in their design for the single instruction multiple data (SIMD) paradigm. Due to the inherent parallelism of vertex shading, GPUs adopted multi-core architectures long before regular microprocessors resorted to such a design. While this decision was driven by the applications in the former case, it was dictated by power and asymptotic single-core frequency limits for the latter. As a result, today's state-of-the-art GPUs consist of many small computation cores compared to few large cores in off-the-shelve CPUs, at the cost of devoting less die area for flow control and data caching in each core. And since GPUs support a higher number of peak floating-point operations per second, researchers are increasingly utilize by the so-called general-purpose graphics programming units (GPGPUs) [11, 6, 16].

It may seem that the incorporation of more cores at the expense of flexibility impedes the adoption of GPUs in areas outside the graphics domain. However, NVIDIA's launch of the Compute Unified Device Architecture (CUDA) with its simple but effective programming model has resulted in the adoption of GPUs by a diversity of domains [15]. The emergence of the NVIDIA CUDA programming model has become a breakthrough toward a more programmer-friendly environment, much in contrast to previous approaches of GPGPU environments. Since then, CUDA has been proved to be well-suited for many applications with only moderate amounts of algorithm re-design and coding efforts. Programmers no longer have to master graphics-specific knowledge, such as was the case with openGL, before being able to efficiently program GPUs.

While it has been demonstrated that CUDA can significantly speedup many computationally intensive applications from domains such as scientific computation, physics and molecular dynamics simulation, imaging and the finance sector [7, 13, 14, 5, 3, 10], it remains less unnoticed in other domains, especially those with more integer computations, with few exceptions[8, 9]. This is partly due to the perception of fast (vector) floating-point calculation being one of the major contributors to performance improvement. However, careful parallel algorithm design may be able to compensate such shortcoming, which is the premise of our work for text search application deployment of GPUs.

We chose to implement one of the fundamental concepts used in information retrieval and text mining, namely the term frequency / inverse document frequency (TFIDF) rank search [2]. This algorithm can be easily extended to quantitatively measure the similarities between any two documents, which is our focus. It thus plays a significant role (in many variations) in text searching, classification, and clustering. The present work explores the opportunities of solving basic text mining problems through an efficient implementation of TFIDF on GPUs via CUDA.

2. TFIDF Problem Description

Term frequency is a measure of how important a term is to a document. The ith term's tf in document j is defined as:

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}} \quad (1)$$

where $n_{i,j}$ is the number of occurrences of the term in document d_j and the denominator is the number of occurrences of all terms in document d_j.

The inverse document frequency measures the general importance of the term in a corpus of documents. It is done by dividing the number of all documents by the number of documents contain-

EPHAM'09, March 22-25, 2009, Seattle, Washington

ing the term, and then taking the logarithm.

$$idf_i = log \frac{|D|}{|\{d_j : t_i \in d_j\}|} \quad (2)$$

where $|D|$ is the total number of documents in the corpus and $|\{d_j : t_i \in d_j\}|$ is the number of documents containing term t_i.

Then

$$tfidf_{i,j} = tf_{i,j} * idf_i \quad (3)$$

The idea of TFIDF can be extended to compare the similarities of two documents d_i and d_j. This is done simply by expressing all common terms' tfidf values in two documents as dot products:

$$sim_{i,j} = \sum_k (tfidf_{k,i} * tfidf_{k,j}) \quad (4)$$

The larger the value is, the more similar these two documents are considered.

There are many ways to calculate the TFIDF given a corpus of documents. The most straightforward method, also used by us, is illustrated in Figure 1. The first step, which is part of the document pre-processing prior to the core TFIDF calculation, excerpts and tokenizes each word of a document. It is also in this step that the stop words are removed. Stop words, also known as the noise words, are common words that do not contribute to the uniqueness of the document [1]. In the second step, some cognate words are transformed into one form by applying certain stemming patterns for each. This is necessary to obtain results with higher precision [12]. In step three, the document hash table is built for each document. The <key, value> pairs in the token hash table are the unique words that appear in the document and their occurrence frequencies, respectively. In step four, all of these token hash tables are reduced into one global occurrence table in which the keys remain the same, but values represent the number of documents that contain the associated key. The TFIDF for each term can be easily calculated by looking up the corresponding values in the hash tables according to Equation 3 as seen in step five.

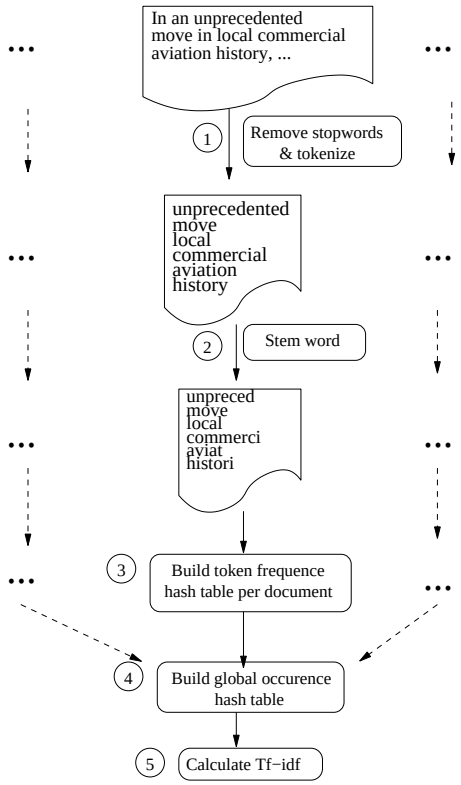

Figure 1: TFIDF Workflow

3. Implementation

One of the key challenges in algorithmic design for GPGPUs is to keep all processing elements busy. NVIDIA's philosophy to ensure high utilization is to oversubscribe, i.e., more parallel work is dispatched than there are physical stream processors available. Using latency-hiding techniques, a processor stalled on a memory reference can thus simply switch context to another dispatched work unit.

In order to fully utilize the large number of streaming processors in NVIDIA's GPUs, we process files in batches with the batch size chosen as 64. Several kernels are developed to implement the steps described in Section 2. The inputs of the tokenize_kernel are raw data streams. One block is assigned to process each document stream. In the kernel, every special character is substituted in-place by a uniform special character and subsequently ignored. This can be done very efficiently with coalesced global memory access. The tokenized stream is then fed into the RemoveAffix_kernel to strip affixes. In this kernel, document streams are divided evenly and processed by multiple CUDA threads.

The kernel requires extensive data movement between host and GPU memories by DMA. First, to handle a large amount of documents/files, especially when total document size is larger than the GPU global memory, the document hash tables needs to be flushed out to host memory once they are completely constructed. Second, the raw data of a document is pushed from host memory to GPU global memory at the beginning of each batch process. To reduce the overhead of memory movement, we developed the CPU/GPU collaboration framework shown in Figure 2. In each batch iteration, the CPU thread first launches the two pre-processing kernels asynchronously. Before it calls the next kernels that write to the document hash table buffer in the GPU's global memory, it waits for

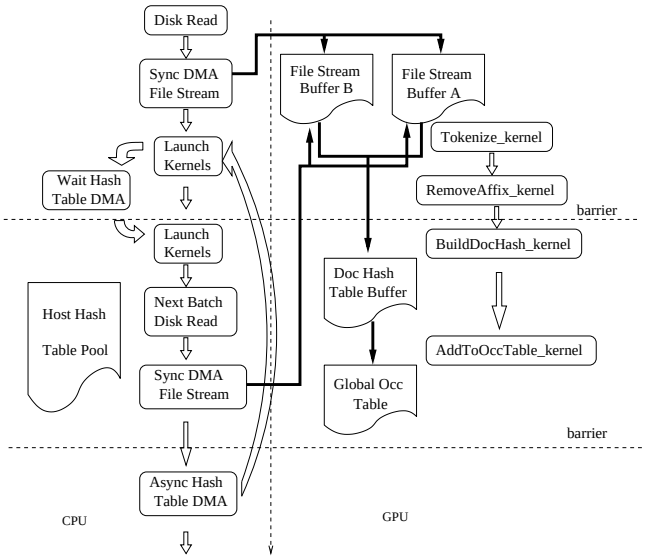

Figure 2: CPU/GPU Collaboration Framework

the completion signal of the previous batch's DMA that transfers the old batch table to host memory. When the GPU is busy generating the document hash tables and inserting tokens into the global occurrence table, the CPU can prefetch the next batch of files from disk and copy them to an alternate file stream buffer. At the end of the batch iteration, the CPU again asynchronously issues a memory copy of the document hash table to the host's memory. Only in the next batch's iteration will the completion of this DMA be synchronized. In this manner, part of the DMA time is overlapped with the GPU calculation by (a) double buffering the document raw data in GPU and (b) overlapping the hash table memory copy in the current batch with the stream preprocessing (tokenize and stem kernels) of the next batch [4].

To further reduce the DMA overhead, one may reduce the size of the document hash table. This table differs from the global occurrence table, which resides in GPU global memory but need not be copied to host until the end of execution. Therefore, the data structures of these tables differ slightly as shown in Figure 3. The document hash table contains a header and an array of entries, which are internally linked as a list if they belong to the same bucket. The header is used to determine the bucket size and to find the first entry in each bucket. In contrast, the global hash table consists of a big array of entries evenly divided into buckets. Because the number of unique terms is considered limited no matter how large the corpus size is, the number of buckets and the bucket size can be chosen sufficiently large to avoid possible bucket overflows.

Another effort to reduce the size of the document hash table avoids storing the actual term/word in the table. Instead, every entry simply maintains an index pointing to the corresponding entry in the global occurrence table where the actual term is saved. To reduce the number of hash key computations at hash insertion and during hash searches, the key is saved as an "unsigned long" in both hash tables. To further reduce the probability of hash collisions (two terms sharing the same key), another field called identity is added as an "unsigned int" to help differentiate terms. The identity is then constructed as $(term\ length << 16)|(first\ char << 8)|(last\ char)$.

Upon investigation, we determined that atomic operations supported by certain GPUs via CUDA are facilitating the construction of a concise document hash table without adversely affecting the parallelism of the algorithm. We alternatively provide another method to generate the same hash table for GPUs without support for atomic operations. Even though the latter method is slower than the first, it is required for GPU devices that do not have atomic operation support (*i.e.*, devices with CUDA compute capability 1.0 or earlier).

3.1 Hash Table Updates using Atomic Operations

Access to hash table entries *via* atomic operations is realized in two steps as depicted in Figure 4. In the first step, the document stream is evenly distributed to a set of CUDA threads. The number of threads, L, is chosen explicitly to maximize GPU's utilization. A buffer storing the intermediate hash table, which is close to the structural layout of the global occurrence table, but with a smaller number of buckets K, is used to sort terms by their bucket IDs. Every time a thread encounters a new term in the stream and obtains its bucket ID, it issues an atomic increment (atomic-add-one) operation to affect the bucket size. (Notice that the objective of this algorithmic TFIDF variant is not to identify identical terms. Instead, its chief objective is to compute a similarity metric.) If we assume that terms are distributed randomly, then contention during the atomic increment operation is the exception, *i.e.*, threads of the same warp are likely atomically incrementing disjoint bucket size entries.

In the next step, the intermediate hash table is reduced to the final, more concise document hash table shown in Figure 3. Each CUDA thread traverses one bucket in the intermediate hash table, detects duplicate terms, and, if finds a new term, reserves a place in the en-

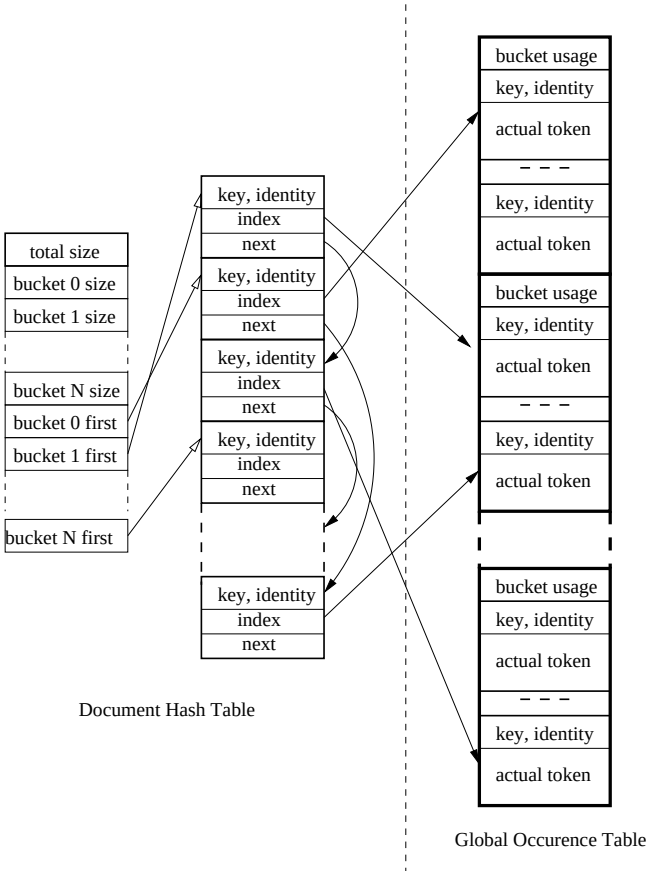

Figure 3: Hash Table Data Structures

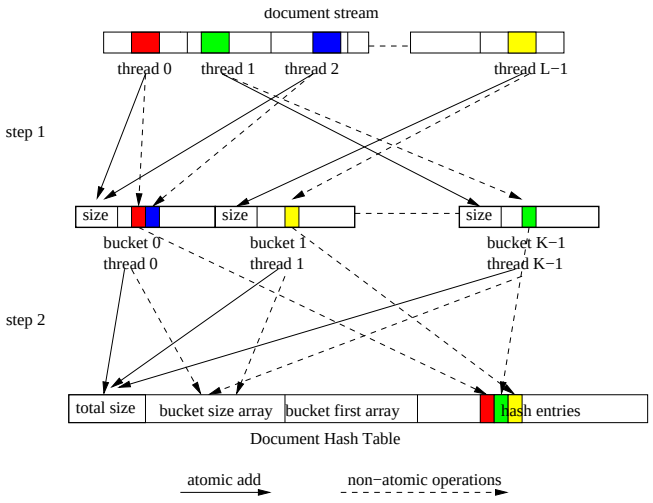

Figure 4: Building a Hash Table with Atomic Operations

try array by atomically incrementing the total size. It then pushes the new entry into the header of the linked bucket list. Since different threads operate on disjoint buckets, each linked list per bucket is accessed in mutual exclusion, which guarantees absence of write conflicts between threads.

3.2 Hash Table Updates without Atomic Operations

In GPUs without atomic instruction support, the document stream is first split into M packets, each of which is pushed into a different hash sub-table owned by one thread in a block, as shown in step 1 of Figure 5. By giving each thread a separate hash sub-table, we guarantee write protection (mutually exclusive writes of the values) between threads. In step 2, K threads are re-assigned to different buckets of the sub-table, identical terms are found in this step, and statistics for each bucket are generated. Because terms have been grouped by their keys in step 1, there will be no write conflicts between threads at this step either. The bucket size information is processed in step 3 to finally merge sub-tables to compose the final document hash table.

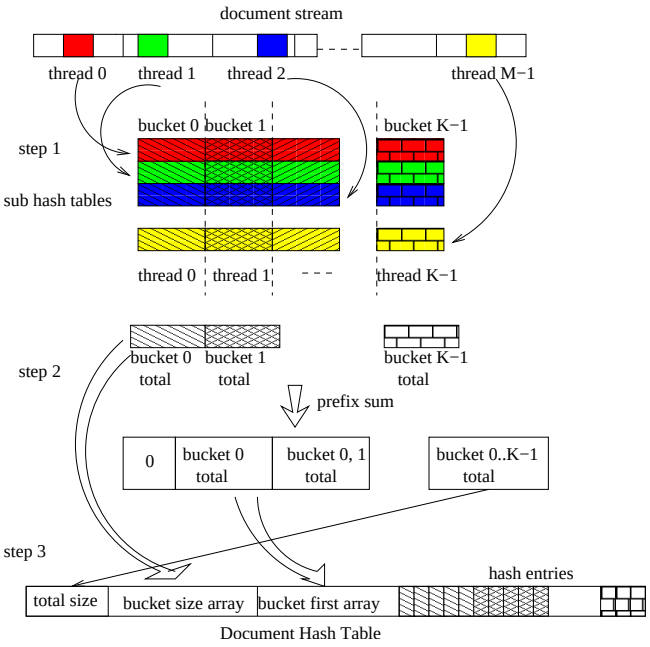

Figure 5: Building a Hash Table without Atomic Operation

3.3 Discussion

The two procedures detailed above to handle hash tokens in a document do not require information from any other documents. Thus, each document can be processed simultaneously and independently in different GPU blocks. With a sufficiently large number of documents, we can fully utilize the GPU cores and exploit NVIDIA's latency hiding on memory references through oversubscription. However, in the first step of the second method, the number of packets M per document is delimited due to memory constraint and the efficiency of the following steps. We choose a value of $M = 16$ in our implementation. To compensate for this constraint, we can spawn more threads L in the first method, *e.g.*, by choosing $L = 512$. This constraint on parallelism results in a non-atomic approach that is slower than its atomic variant.

From the memory usage's perspective, the non-atomic approach consumes more global memory simply because the intermediate hash tables in the non-atomic approach are larger than that in the atomic approach. Both of the above methods cannot handle very large single documents that exceed the size of the global memory. Since our problem domain is that of Internet news articles, which typically do not exceed more than 10K words, documents fits in memory for our implementation. This framework is even suitable for arbitrarily large corpus sizes as we could reused without changes both intermediate hash tables and the document hash table, the latter of which is flushed to host memory for each batch of files.

4. Experimental Results

During experimentation, our CUDA variant was compared against a functionally equivalent CPU baseline version (single-threaded in C/C++) of the TFIDF benchmark implementation. The test platform was a Linux Fedora 8 Core with a dual-core AMD Athlon 2 GHz CPU with 2 GB of memory. The installation included the CUDA Compilation tools (Version 1.1) of the CUDA 2.0 beta release and NVIDIA's Geforce GTX 280 as a GPU device. The test input data was selected by the original TFIDF designers as a subset of Internet news documents with variable sizes ranging from around 50 to 1000 English words (after stop-word removal).

We first compare the execution time for one batch of 96 files. The individual module speedup and their percentages in total are shown in Figure 6 and Figure 7.

Notice that the speedup on the y-axis of Figure 7 is depicted on a

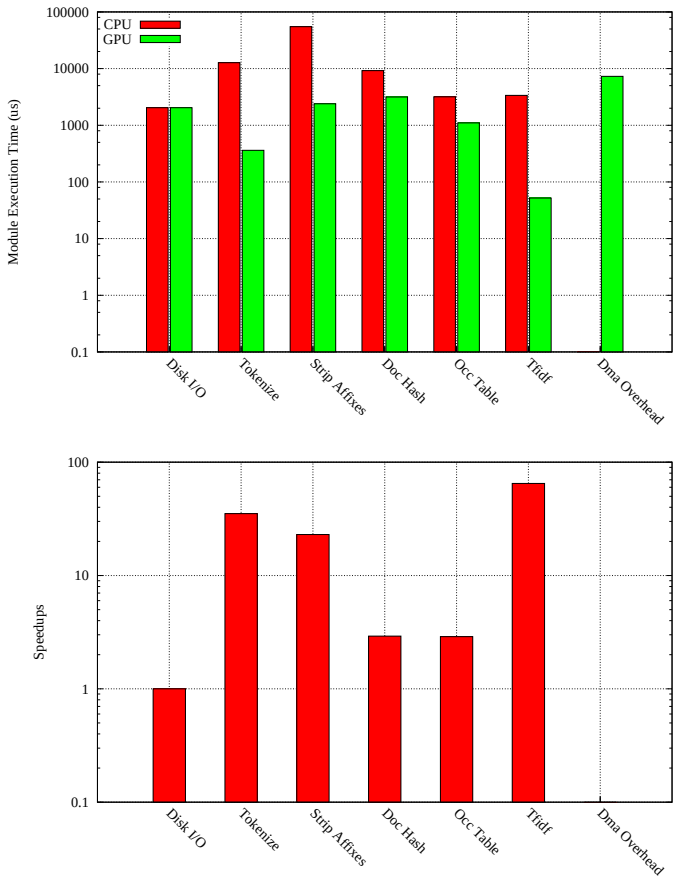

Figure 6: Per-Module Performance: CPU baseline vs. CUDA

logarithmic scale. Compared to the CPU baseline implementation, we achieve more significant speedups for those modules engaged in the pre-processing phase (factor of 30 times faster in tokenize and 20 times faster in strip affixes kernels) than for those at the hash table construction phase (around 3 times faster in both document hash table and occurrence table insertion kernels). The limits in speedup during the latter are due to the multi-step hash table construction algorithms described in Section 3. The algorithm has certain overheads that the CPU benchmark does not contain. These overheads include (a) the construction of intermediate or hash sub-tables; (b) branching penalties suffered from the SIMD nature of GPU cores due to the imbalance in the distribution of tokens for a hash table's buckets; and (c) non-coalesced global memory access patterns as a result of the randomness of the hash key generation. Furthermore, the kernel for occurrence table insertion does not fully exploit all GPU cores because insertion is inherently serialized over files to avoid write conflicts within the same hash table bucket.

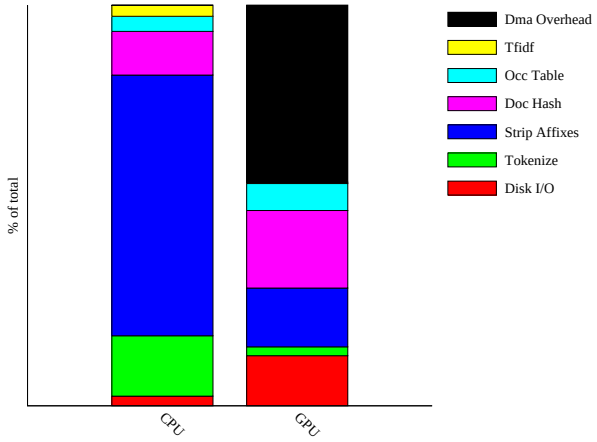

Figure 7: Per-Module Contribution to Overall Execution Time

We also observe a reduction in the calculation time to the extent that the DMA overhead has become the largest contributor to overall time in a *single batch* scenario accounting for almost half of the total execution time. The combined time with disk I/O exceeds the total kernel execution time on GPU.

The observation above gives us the motivation to mitigate the memory overhead by double buffering the stream and hash tables when the corpus size gets larger. While we cannot hide the DMA overhead of a first batch, the DMA time of subsequent batches can be completely overlapped with the computational kernels in a *multi-batch* scenario. Figure 8 shows the execution time of CPU and CUDA with different corpus sizes.

The execution time of both of the two aforementioned methods are measured. With almost perfect parallelization between GPU calculation and data migration, we can hide almost all the kernel execution time in the DMA transfer and disk I/O time, which indicates a lower bound of the execution time. As a result the the asymptotic average batch processing time is almost half comparing to the single batch execution time, in which case the calculation and DMA cannot be overlapped. We also observe that the overall acceleration rates are 9.15 and 7.20 times faster than the CPU baseline.

5. Conclusion

In this paper, we presented a hardware accelerated implantation of TFIDF rank search algorithm exploiting GPU devices through

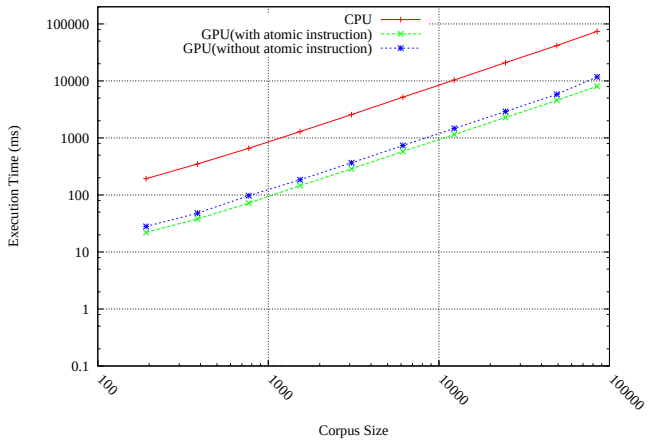

Figure 8: Execution Time with Different Corpus Size

NVIDIA's CUDA. We developed two highly parallelized methods to build hash tables, one with and one without atomic operation support. Even though floating-point calculations are not dominating this text mining algorithm and its text processing characteristics limits the effectiveness due to non-synchronized branching and diverging, data-dependent loop bounds, we achieved a significant speedup over the baseline algorithm on a general-purpose CPU. More specifically, we achieve up to a 30-fold speedup over CPU-based algorithms for selected phases of the problem solution on GPUs with overall wall-clock speedups ranging from six-fold to eight-fold depending on algorithmic parameters. This experiment demonstrates the potential of GPUs to accelerate even integer-oriented, branch-dominated massive data text mining algorithms by carefully redesigning data structures to provide massive parallelism, which makes these problems suitable for latency hiding by exploiting task over subscription in GPUs.

6. References

[1] http://en.wikipedia.org/wiki/stop_words.

[2] http://en.wikipedia.org/wiki/tf-idf.

[3] http://www.nvidia.com/object/cuda_home.html.

[4] Tong Chen and Zehra Sura. Optimizing the use of static buffers for dma on a cell chip. In *In The 19th International Workshop on Languages and Compilers for Parallel Computing*, 2006.

[5] Matthew Curry, Lee Ward, Tony Skjellum, and Ron Brightwell. Accelerating reed-solomon coding in raid systems with gpus. In *International Parallel and Distributed Processing Symposium*, April 2008.

[6] Zhe Fan, Feng Qiu, A. Kaufman, and S. Yoakum-Stover. Gpu cluster for high performance computing. page 47, 2004.

[7] Massimiliano Fatica and Won-Ki Jeong. Accelerating matlab with cuda. In *HPEC*, September 2007.

[8] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast computation of database operations using graphics processors. In *SIGMOD '04*, pages 215–226, New York, NY, USA, 2004. ACM.

[9] Naga K. Govindaraju, Nikunj Raghuvanshi, and Dinesh Manocha. Fast and approximate stream mining of quantiles and frequencies using graphics processors. In *SIGMOD '05*, pages 611–622, New York, NY, USA, 2005. ACM.

[10] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *HiPC*, pages 197–208, 2007.

[11] Naga Govindaraju... Jogn D. Owens, David Luebke. A survey of general-purpose computation on graphic hardware. *Technique Report*, 26(1):80–113, 2007.

[12] Mark Kantrowitz, Behrang Mohit, and Vibhu Mittal. Stemming and its effects on tfidf ranking (poster session). In *SIGIR '00: Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 357–359, New York, NY, USA, 2000. ACM.

[13] Hubert Nguyen(ed). *GPU Gems 3*. Addison-Wesley Professional, 2007.

[14] Antonio J. Rueda Ruiz and Lidia M. Ortega. Geometric algorithms on cuda. In *GRAPP*, pages 107–112, 2008.

[15] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 73–82, 2008.

[16] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program gpus for general-purpose uses. In *Architectural Support for Programming Languages and Operating Systems*, pages 325–335, 2006.