GPU-Based Minwise Hashing

Ping Li
Dept. of Statistical Science
Cornell University
Ithaca, NY 14853
pingli@cornell.edu

Anshumali Shrivastava
Dept. of Computer Science
Cornell University
Ithaca, NY 14853
anshu@cs.cornell.edu

Arnd Christian König
Microsoft Research
Microsoft Corporation
Redmond, WA 98052
chrisko@microsoft.com

ABSTRACT

Minwise hashing [1] is a standard technique for efficient set similarity estimation in the context of search. The recent work of b-bit minwise hashing [3] provided a substantial improvement by storing only the lowest b bits of each hashed value. Both minwise hashing and b-bit minwise hashing require an expensive preprocessing step for applying k (e.g., $k = 500$) permutations on the entire data in order to compute k minimal values as the hashed data. In this paper, we developed a parallelization scheme using GPUs, which reduced the processing time by a factor of $20 \sim 80$. Reducing the preprocessing time is highly beneficial in practice, for example, for duplicate web page detection (where minwise hashing is a major step in the crawling pipeline) or for increasing the testing speed of online classifiers (when the test data are not preprocessed).

Categories and Subject Descriptors

H.3.3 [**Information System**]: Information Storage and Retrieval—*Information Search and Retrieval*

General Terms

Algorithms, Experimentation, Performance

Keywords

GPU, Hashing, Large-Scale Learning

1. INTRODUCTION

Minwise hashing [1] is a standard technique for efficiently computing approximate set similarities in the context of search, with further applications in the context of content matching for online advertising, syntactic similarity algorithms for enterprise information management, Web spam, etc. The development of b-bit minwise hashing [3] provided a substantial improvement in the estimation accuracy and speed by proposing a new estimator that stores only the lowest b bits of each hashed value. Recently, [4] proposed using b-bit minwise hashing in the context of learning algorithms such as SVM or logistic regression, to enable scalable learning, at negligible reduction in learning quality.

The major overhead associated with minwise hashing is the computation of k (e.g., 500) hash signatures for each data vector. While parallelizing the signature computations is conceptually simple, it still comes at the cost of using additional hardware and electricity. Thus, any improvements in the speed of signature computation may be directly reflected in the cost of the required infrastructure.

WWW 2012 Companion, April 16–20, 2012, Lyon, France.
ACM 978-1-4503-1230-1/12/04.

To address this major issue, this paper studies how to speed up the execution of signature computations through the use of graphical processing units (GPUs). GPUs offer, compared to current CPUs, higher instruction parallelism and very low latency access to the internal GPU memory, but comparatively slow latencies when accessing the main memory [2]. As a result, many data processing algorithms (especially ones with random memory access patterns) do not benefit significantly when implemented using a GPU. However, the characteristics of the minwise hashing algorithm make it very well suited for execution using a GPU.

More details about the use of GPUs for (b-bit) minwise hashing are available in a technical report [5], which also addressed two other closely-related, important issues. (i) **Online learning**: b-bit minwise hashing can substantially reduce the data loading time for each training epoch, which is the major bottleneck of online learning. (ii) **Massive permutation matrix**: When the data dimensionality is on the order of billions, it becomes impractical (or too expensive) to use (and store) a permutation matrix for the random permutations required by minwise hashing. Thus, we resort to simple hash functions such as various forms of universal hashing. [5] demonstrated that both the 2-universal (2U) and 4-universal (4U) hash families are practically reliable in the context of using b-bit minwise hashing for training SVM and logistic regression.

In fact, one major limitation of GPUs is that they have fairly limited memory (for storing the permutation matrix). In this paper, we always use 2U and 4U hash functions in our GPU experiments.

2. MINWISE HASHING AND B-BIT MINWISE HASHING

Binary data can be viewed as sets. Consider two sets, $S_1,\ S_2 \subseteq \Omega = \{0, 1, 2, ..., D-1\}$ (where $D = 2^{64}$ in practice). Apply a random permutation $\pi : \Omega \to \Omega$. The collision probability is

$$\mathbf{Pr}\left(\min(\pi(S_1)) = \min(\pi(S_2))\right) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} = R. \qquad (1)$$

where R is known as the *resemblance*. The method of b-bit minwise hashing [3] only stores the lowest b bits (instead of 64 bits) of each hashed value. Define two hashed values as

$$z_1^{(b)} = \text{the lowest } b \text{ bits of } z_1,\ z_2^{(b)} = \text{the lowest } b\text{-bits of } z_2. \quad (2)$$

THEOREM 1. *[3] Assume D is large.*

$$P_b = \mathbf{Pr}\left(z_1^{(b)} = z_2^{(b)}\right) = C_{1,b} + \left(1 - C_{2,b}\right) R, \qquad (3)$$

where $C_{1,b}, C_{2,b}$ are functions of $(D, b, |S_1|, |S_2|, |S_1 \cap S_2|)$. □

Thus, we can estimate P_b (and R) from k permutations $\pi_1, \pi_2, ..., \pi_k$:

$$\hat{R}_b = \frac{\hat{P}_b - C_{1,b}}{1 - C_{2,b}}, \qquad \hat{P}_b = \frac{1}{k}\sum_{j=1}^{k} 1\left\{z_{1,\pi_j}^{(b)} = z_{2,\pi_j}^{(b)}\right\} \qquad (4)$$

which are essentially inner products because

$$1\left\{z_1^{(b)} = z_2^{(b)}\right\} = \sum_{t=0}^{2^b-1} 1\{z_1^{(b)} = t\} \times 1\{z_2^{(b)} = t\} \quad (5)$$

In other words, $\hat{P}_b$ is an inner product between two vectors in $2^b \times k$ dimensions with exactly k 1's. This provides a simple practical strategy for using b-bit minwise hashing for large-scale learning.

3. SIMPLE HASH FUNCTIONS

When the space (D) is not large, it can be affordable to store a "permutation matrix" of size $D \times k$ (of integers). For industrial applications which may use $D = 2^{64}$, this is not practical.

The general idea is to "regenerate" entries of the permutation matrix as needed using (e.g.,) the 2-universal (2U) hash function or the 4-universal (4U) hash function:

$$h_j^{(2U)}(t) = \{c_{1,j} + c_{2,j}\, t \mod p\} \mod D, \quad (6)$$

$$h_j^{(4U)}(t) = \left\{\sum_{i=1}^{4} c_{i,j} t^{i-1} \mod p\right\} \mod D, \quad (7)$$

where $p > D$ is a prime number and $c_{i,j}$'s are chosen uniformly from $\{0, 1, ..., p-1\}$. We only need to store the coefficients $c_{i,j}$. Using simple hash functions may potentially allow us to apply b-bit hashing to high-dimensional data. However, no prior studies have reported the performance of learning algorithms using b-bit minwise hashing with simple hashing functions.

4. LEVERAGING GRAPHICS PROCESSORS FOR FAST HASHING COMPUTATION

In comparison with commodity CPUs, GPUs offer significantly increased computation speed and internal memory bandwidth. GPUs consist of a number of SIMD multiprocessors. At each clock cycle, all processors in a multiprocessor execute identical instructions, but on different parts of the data. Thus, GPUs can leverage spatial locality in data access and group accesses to consecutive memory addresses into a single access; this is called *coalesced access*.

In light of these properties of GPU processing, our algorithms to compute b-bit minwise hashes on a GPU proceeds in 3 distinct phases: First, we read in chunks of 10K-20K sets from disk into main memory and write these to the GPU memory. Then, we compute the hash values and the corresponding minima by applying all k hash functions to the data currently in the GPU and retaining, for each hash function and input set, the corresponding minima.

Because we transfer larger blocks of data, the main memory latency is reduced through the use of main memory pre-fetching. Moreover, because the computation within the GPU itself scans through consecutive blocks of data in the GPU-internal memory, performing the same computation (with a different hash function) for each set entry k times, we can take advantage of coalesced access and the massive parallelism of the GPU architecture.

Experimental Setup: The GPU platform in our experiments is the NVIDIA Tesla C2050, which has 15 Simultaneous Multiprocessors (SMs), each with 2 groups of 16 scalar processors. The two fairly large datasets used in our experiments are described in Table 1.

Results: Table 2 shows the overhead (for $k = 500$) of the CPU-based implementation, broken down into the time required to load the data into memory and the time for the minwise hashing computation, as well as the total overhead for the GPU-based processing (i.e., the last column in Table 2, for batch size = 10K).

Figure 1 shows a breakdown of the overhead for the GPU-based implementation, which we separate into time spent transferring the data and the actual computation; here, we also vary the batch size we use to move data into/out of the GPU. The GPU-based implementation obtains an improvement of 20 - 80 folds over the CPU-based implementation. One parameter we vary here is the batch size we use to move data into/out of the GPU. The time spent within the GPU kernel and transferring the data to the GPU is not affected significantly by the batch size, but the speed at which data is transferred back does vary significantly with this parameter. However, for any setting of the batch size does it hold that the time spent transferring data is an order of magnitude smaller than the time spent on the actual processing within the GPU. This is the key to the large speed-up over CPU implementations we see.

Table 1: Data information (size in LibSVM format). The (expanded) *rcv1* dataset (200GB) contained the original *rcv1* features + all pairwise combinations (products) of features + 1/30 of 3-way combinations of features, for testing the scalability of algorithms.

Dataset	n	D	# Avg Nonzeros	Train / Test
Webspam (24 GB)	350000	16609143	3728	80% / 20%
Rcv1 (200 GB)	781265	1010017424	12062	50% / 50%

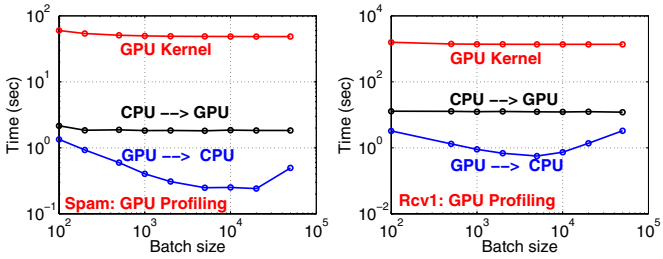

Figure 1: Overhead of three phases of GPU implementation.

Table 2: Data loading and preprocessing ($k = 500$) time (sec)

Dataset	Loading	Permu	2U	4U	2U (GPU)
Webspam	9.7×10^2	6.1×10^3	4.1×10^3	4.4×10^4	51
Rcv1	1.0×10^4	–	3.0×10^4	–	1.4×10^3

5. ACKNOWLEDGMENTS

This work is partially supported by NSF (DMS-0808864, SES-1131848), ONR (YIP-N000140910911), and DARPA.

6. REFERENCES

[1] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the web. In *WWW*, pages 1157 – 1166, Santa Clara, CA, 1997.

[2] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. Fast: fast architecture sensitive tree search on modern cpus and gpus. In *SIGMOD*, pages 339–350, 2010.

[3] Ping Li and Arnd Christian König. Theory and applications b-bit minwise hashing. *Commun. ACM*, 2011.

[4] Ping Li, Anshumali Shrivastava, Joshua Moore, and Arnd Christian König. Hashing algorithms for large-scale learning. In *NIPS*, Vancouver, BC, 2011.

[5] Ping Li, Anshumali Shrivastava, and Arnd Christian König. b-bit minwise hashing in practice: Large-scale batch and online learning and using GPUs for fast preprocessing with simple hash functions. Technical report.