Efficient Weighted Histogramming on GPUs with CUDA

Mo Xu, Ningyi Xu, Chunshui Zhao and Feng-Hsiung Hsu
Microsoft Research Asia, Beijing, China
v-moxu, ningyixu@microsoft.com

ABSTRACT

The histogram is a fundamental statistical tool that has been extensively used in various domains. In data mining and machine learning applications, weighted histogram calculation often serves as a key component in the processing of their massive data sets. However, the *atomic operation*, which is introduced to resolve the collisions in GPU-based parallel histogramming with large number of bins, brings the overhead of instruction serialization and limits the performance and performance predictability. In this work, we present a new method for histogramming on GPUs, which reduces the collision intensity by rearranging the input, and provides predictable performance over data sets with different statistics. Using the *shared memory* effectively, our method shows improved performance over the state-of-the-art implementations. According to the number of bins and sparseness of the values, we then propose a hybrid method which dynamically chooses the best implementation from traditional methods and the new method. An overall speedup of 13x is observed on a data set from a commercial search engine when comparing with the CPU implementation.

Keywords

Histogram, GPU, CUDA, LambdaMART

1. INTRODUCTION

Histograms are extensively used in various domains, such as imaging processing, machine learning and data mining. They are estimates of the probability density of the data. Different from the common histogram, a weighted histogram accumulates the assigned weight of each input data points to the corresponding bin. The calculation of weighted histogram often serves a key component in machine learning and data mining applications such as face detection [6] and ranking [4]. In boosting algorithms, the most widely used weak learner is the binary classifier, whose threshold is often trained by calculating weighted histograms [13], taking most of the running time. Our work is motivated by accelerating the LambdaMART ranking algorithm, which will be introduced briefly in Section 2.1.

Recently, Graphic Processing Units (GPUs) are becoming an important platform for general-purpose data parallel computing [7]. Since modern GPU contains 512 or more cores [5], it provides much greater computation capability and bandwidth than CPU can provide.

GPU-based calculation of weighted histogram, especially with large number of bins, is difficult [12] because the on-chip *shared memory* is limited compared to the memory access requirement of parallel threads. The floating-point weight, unlike the integer value, cannot be compressed, making ordinary optimization trivial. One solution is that many concurrent threads fill in one histogram (rather than a same number of histograms), and each uses *atomic operations* to ensure the correctness. This method allows multiprocessors accommodate enough threads simultaneously, and thus potentially achieves peak performance when the overhead of atomic operations is negligible. This overhead is minimized when the data is uniformly distributed. In common cases, atomic operations lead to serial executions, which may cause notably performance drop.

In this paper, we present a new method to reduce the overhead of atomic operations. The key idea of our method is to create an extra buffer for each thread, then to rearrange the sequence of inputs according to their value and the thread index, so as to reduce collisions when finally updating the bin counters in parallel. Our method outperforms previous methods on the data that is not so uniformly distributed, and achieves better performance predictability.

For histogram calculation with small number of bins, the traditional method without atomic operations is effective. We further use the register to increase the Instruction-Level Parallelism (ILP), and achieve better performance than previous implementation. Specifically, loop unrolling gives an extra 2x speedup. The optimization idea is built upon from Volkov [14].

Since in data mining applications, the histogram calculation with varies number of bins (in LambdaMART, from 2 to 1024) are often required, and data with all kind of distribution is involved, we then present a hybrid method which choose the best implementation according to the number of bins and the collision intensity, which is known in advance.

This hybrid method is built upon the detailed performance analysis of the three separate methods (two for large-number of bins and one for small-number of bins), and provides the best performance for real-world applications.

The contributions of this paper are:

- We present a new method for histogram calculation on GPUs, which efficiently uses the shared memory, and reduces the overhead of atomic operations.
- We optimize the traditional method by exploiting the register on GPU, and achieve better performance than previous implementation.
- We provide a hybrid method for real-world applications, which dynamically chooses the best implementation, and achieves the best performance.

The rest of this paper is organized as follows: we briefly introduce the background of LambdaMART algorithm and the Compute Unified Device Architecture (CUDA) in Section 2. Then the previous work is reviewed in Section 3. Next, we present the existing methods and the novel method and our optimization in Section 4. Section 5 details the methodology for our experiments of which the results can be found in Section 6. Finally, Section 7 concludes the paper.

2. BACKGROUND

We first define our problem by introducing the algorithm. Then we provide some basic knowledge about our programming model and hardware platform, i.e. the CUDA and the NVIDIA Fermi Architecture respectively.

2.1 LambdaMART Ranking algorithm

LambdaMART, which is a promising learning to rank algorithm, combines MART (Multiple Additive Regression Trees) and LambdaRank [2]. The training data is the feature values (f_{ij}) and the relevance labels (l_i, such as bad, good, excellent, etc.) of documents (d_i, URLs). The lambda (λ_i) can be seen as the cost function of the MART [3]. In partitioning each node of the regression tree, both the unweighted histograms and the λ-weighted histograms of all features are required to choose the most discriminative feature and the threshold. After each regression tree is built, λ is recalculated and iteratively used as the weight of histograms of the next tree. Experiments show that the histogram calculation is the bottleneck of this algorithm, costing above 50% of the total training time [12].

We summarize the features characterizing our problem as follows: 1) many histograms (as many as the number of features $\times$ the number of nodes) from one set of feature values need to be recalculated many times (as many as the number of regression trees) with only the λ updating. So if we put the entire feature values on the GPU device memory, the overhead of data-transfer from CPU to GPU will be trivial compared to that of the intense histogram calculations. 2) The range of feature values varies from 2 to 1024, supporting binary features, discrete features and continuous features. For continuous features, 1024 is an empirical number for precision (larger number of bins may cause over fitting). This motivate us to investigate the performance under different number of bins. 3) When calculate the weighted histogram, along with every integer bin counter, there is one floating-point weight need to be stored. This is the main reason why previous optimization for histogram calculation is ineffective for this application. All these characteristics applies to other boosting-tree algorithms.

2.2 CUDA and the GPU architecture

We provide a quick overview of the concept, terminology and basic optimizing strategy of the Compute Unified Device Architecture (CUDA) and NVIDIA GPU architecture that is used in this paper. Detailed information can be found in [9]. A reader who is familiar with CUDA may skip this subsection.

CUDA provides a general-purpose programming model for GPUs. The *kernel code*, running on the GPU, is executed by a number of *threads*, each having a unique *id*. Threads are grouped into thread-blocks, enabling synchronization and the use of a *shared memory*. All threads have their own register, and share the device DRAM, called the *global memory*.

The GPU hardware is consist of an array of *streaming multiprocessors*, each executing one or more thread-block(s) concurrently. The multiprocessor schedules threads as *warps*, which is groups of typically 32 parallel threads within thread-blocks, in an SIMD-manner. An on-chip shared memory and a register file are available on each multiprocessor with lower access latency than that of the global memory. However, the *atomic operations* on shared memory may cause serialization, and result in a huge drop in performance.

To hide the memory latency, the GPU does not (entirely) rely on caches, but instead switches to other *active warps*. The ratio of the number of active warps per multiprocessor to the maximum number of possible active warps (48 in GTX580) is called *occupancy*, which is the metric of the Thread-Level Parallelism (TLP) achieved by the hardware, and is limited by the register and shared memory requirements. Another way to increase the parallelism so as to further hide the latency is to exploit the Instruction-Level Parallelism (ILP), which is a measure of how many of the operations in the serial kernel that can be performed simultaneously.

3. RELATED WORK

Sharp implemented a decision tree and forest on the GPU using Direct3D [12] and found that the aggregation of the histograms on the GPU is comparatively slow. In their profiling with GTX280, histogram calculation takes 96% of the training time, while on CPU it takes 53%, which indicate the difficulty of accelerating the weighted histogram calculation. Previous work focuses on unweighted histogram calculation, which is mostly used in imaging processing applications.

In CUDA Software Development Kit, Podlozhnyuk demonstrated two approaches, one for 64-bin and the other for 256-bin [10]. Both of them use shared memory to achieve efficient data transfer. In the implementation of 64-bin histogram, each thread holds a full histogram (64 counters) on shared memory, and no atomic operation is introduced. To reduce the size of shared memory used by a single thread, hence to ensure necessary occupancy, Podlozhnyuk use single-byte bin counters for each thread. In 256-bin histogram, each warp (32 threads) has a full histogram in shared memory. In this way, the collisions between warps (inter-warp collisions) are avoided, while the collisions within a warp (intra-warp collisions) are inevitable.

Shams and Kennedy presented two methods for histogram calculation with a wide range of bins using CUDA [11]. One method involves simulating a mutex (mutual exclusion) on software for safe access to shared objects, since hardware atomic operation is not supported by NVIDIA GPUs of *compute capability 1.0*.[1] The other method proposed by [11] is to avoid collisions between parallel threads, similar to the 64-bin implementation in CUDA SDK, by reducing the size of bin counters according to the number of bins, so as to achieve necessary occupancy.

Brosh and Tam proposed a self-optimizing histogram algorithm [1], which changes the bin size adaptively according to the data distribution calculated in the first time.

Nugteren presented two method for predictable histogram calculation on GPUs [8]. Considering the high correlation between neighboring pixels in typical image data, Nugteren reduced the amount of shared memory collisions by shuffling the input throughput global memory. Besides, this work also compares four collision-free 256-bin implementations with different mappings of the histogram on the shared memory to avoid bank conflict.

In sum, when dealing with the calculation of many-bin histogram on GPU, all mentioned studies reduce the shared memory usage in either of these two ways: 1) using smaller bin counters, or 2) parallel threads updating one single histogram. Since in weighted histogram calculation, an extra floating point number (namely the λ in Section 2.1) accompanied with each bin counter should also be stored in the shared memory, reducing the size of bin counters brings little performance gain. Our new method focuses on reducing the intensity of the collisions stemming from many threads concurrently updating one single histogram.

4. IMPLEMENTATION METHODS

4.1 Overview

The algorithm of histogram calculation in LambdaMart is described in Algorithm 1. Obviously, Algorithm 1 is *memory-bandwidth limited* since no complex arithmetic operations are involved. Therefore, the most vital issue when mapping the histogram algorithm to GPUs is the optimization for memory bandwidth. Note that the type of the input $Value$ may be *unsigned char* or *unsigned short*, which depends on N_{bins}. However, we constantly use *unsigned int* as global memory fetch type to maximize the bandwidth utilization, and then get the actual value by *shift* or *and* operation.

One straightforward way to calculate histogram is that each thread updating its bin counters in the *global memory* directly. This method requires intensive random access to the

Algorithm 1: Serial Histogram Calculation

```
Input: unsigned int $Value[i][j]$, $i = 1, 2, ..., N_{feature}$,
       $j = 1, 2, ..., N_{doc}$ and float $\lambda[j]$
Output: unsigned int $SumCount[i][k]$ and float
        $Sum\lambda[i][k]$, $k = 1, 2, ..., N_{bins}$
1  for $i = 1; i \leq N_{feature}$ do
2      for $k = 1; k \leq N_{bin}$ do
3          $SumCount[i][k]$=0;
4          $Sum\lambda[i][k]$=0;
5      end
6      for $j = 1; j \leq N_{doc}$ do
7          unsigned int $k = value[i][j]$;
8          //get the actual value k
9          float $lambda = \lambda[i]$;
10         $SumCount[i][k]$++;
11         $Sum\lambda[i][k]$+=$lambda$;
12     end
13 end
```

global memory, which will cause dramatically decrease in throughput [11].

Since shared memory has much higher bandwidth than global memory does, we mainly discuss the techniques of how to use shared memory as cache for higher throughput in next subsections. Traditional mappings of histogram calculation on GPUs can be categorized into two classes according to the number of bins (the N_{bin} in Algorithm 1). For small-N_{bin} histogram, **collision-free method** is efficient, which, however, cannot deal with the cases when N_{bin} is large. For large-N_{bin} cases, we must resort to atomic operations to guarantee the correctness (we call it **collision method**). To reduce the overhead of atomic operations, we then present our new method (referred to as **collision-reduced method**). Finally, for data mining applications, we present a hybrid method, which is built upon the analysis of the three discrete methods, and chooses the optimal implementation among these three method according to N_{bin} and the data statistic.

4.2 Collision-Free Method

4.2.1 Basic Implementation

In this method, each thread in parallel has its own histogram in shared memory, thus collisions can be entirely avoided. The implementation is quite straightforward. Each thread-block processes one feature, and there are $N_{feature}$ blocks on the GPU. Each thread fetches input data from global memory in coalesced way, and put it into the right bin counter according to the data value, iteratively (see Figure 1, Step 1). Then, all histograms in one thread-block are summed up, where each thread takes charge of one bin counter at a time. Finally, the histogram is copied to global memory (see Figure 1, Step 2). In following figures, white boxes denote the storage in shared memory, grey boxes denote that in global memory, and arrows mean parallel threads. N_{thread} means the number of threads within one thread-block. For simplicity, in Figure 1 we omit the λ accompanied with every bin counter.

In this way, the coalesced access to the global memory can be

[1]In this work, we use GPUs of compute capability 2.0, which support hardware atomic operation well.

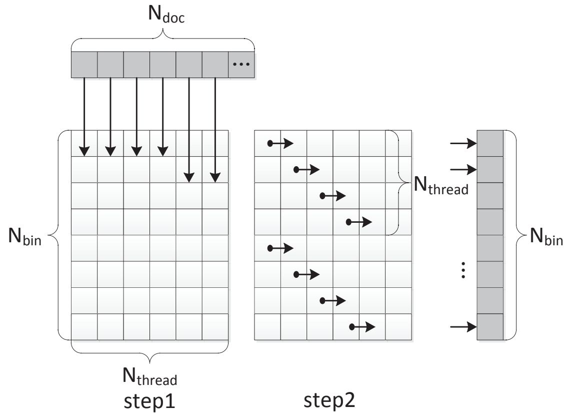

Figure 1: Collision-free method kernel

best achieved. To avoid the bank conflict in shared memory, when summing up the histograms, threads start in a staggered way, as is shown in Figure 1, Step 2. Although by the CUDA profiler we can clearly see that the bank conflict is totally eliminated, there is no appreciable improvement in performance. This is because the summing up operation consumes far less time comparing with reading memory from the global memory.

The major shortcoming of this method is that when the number of bins increases, the number of threads that can simultaneously executing in one multiprocessor ($N_{ActiveThread}$) will decrease dramatically. Thus the theoretical *occupancy* will quickly become intolerably low. Due to the data-intensive nature of this algorithm, the latency of memory fetch can hardly be hidden in the kernel with such low occupancy. Therefore, the performance will be rather poor in this condition (Section 6.1.2).

Quantitatively, the size of one full histogram can be calculated as $N_{bin} \times (sizeof(SumCount) + 4\text{B})$. Given that the type of $SumCount$ is *unsigned short*, then the shared-memory consumption by each thread is $6N_{bin}\text{B}$. Since the total shared memory for each multiprocessor is 48kB, N_{bin} $N_{ActiveThread}$ must not be larger than 8k. And the larger the N_{bin}, the smaller the $N_{ActiveThread}$, and thus the lower occupancy ($\frac{N_{thread}}{1536}$) that GPUs can achieve.

4.2.2 More Efficient Instruction Pipeline

The number of features ($N_{feature}$) and the number of documents (N_{doc}) affect the efficiency of the instruction pipeline on GPUs: larger data size leads to more saturated pipelines. We will illustrate this point in Section 6.1.1. In data mining applications, the number of documents can be very large. So if $N_{feature}$ is too small to occupy the entire device, (This case is common when we use the hybrid method, which partitions the total features.) we can divide the documents into several segments, and merge the sub-histograms after getting the histograms of all these segments.

4.2.3 Exploiting Instruction-Level Parallelism

In this memory-intensive application, the latency of global memory access can hardly be totally hidden by the Thread-Level Parallelism (TLP), especially when the occupancy is relatively low. So we further exploit the GPU registers to achieve higher instruction-level parallelism. Specifically, we add padding to the original input data, and then unroll the loop manually (The "$\#pragma\,unroll$" does not bring any performance gain). This single optimization gives us an extra 2x speedup (see Section 6.1.1). This optimization idea was inspired by Volkov [14].

Note that the loop unrolling also leads to more usage of registers, and thus potentially limits the occupancy, especially when N_{bin} is small and shared memory is not key resource. So the degree of loop unrolling needs to be tuned carefully to achieve the best performance. Detail result can be found in Section 6.1.2.

4.3 Collision Method

When dealing with the histogram calculation with large number of bins, in order to achieve high occupancy, we need to reduce the shared memory used by each thread. In this method, many concurrent threads fill in one single histogram using atomic operations. We illustrate this method by Figure 2. We also try loop unrolling, whose effects in this method in more subtle, and is discussed in Section 6.2.1. This is because in the collision method, the TLP can easily be achieved, and the real bottleneck is the overhead of atomic operations. And the distribution and the sequence of the input data affect the performance to a large degree, as is shown in Section 6.2.3 and Section 6.2.4.

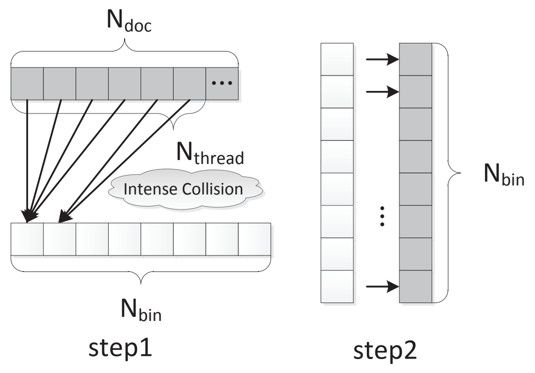

Figure 2: Collision method kernel

4.4 Collision-Reduced Method

4.4.1 Basic Implementation

In this subsection, we present our new method to reduce the intensity of collisions among concurrent threads. The key idea of our method is to create an extra buffer for each thread, then using it to rearrange the sequence of inputs according to their value and the thread index, so as to reduce the collisions. We illustrate our method by Figure 3. Algorithm 2 is the pseudo-code of the CUDA kernel, where *tid* stands for *threadIdx.x*.

Apart from a full histogram shared by all threads in a thread-block, each thread has two extra buffers ($BufferValCount$ and $BufferLambda$ (not shown in Figure 3 for simplicity)). The length of buffer is N_{buffer}. As is shown in the right upper corner of Figure 3, each unit

Algorithm 2: Collision-Reduced Histogram Kernel

```
gridDim.x = $N_{feature} \times \frac{N_{doc}}{N_{buffer} N_{warp} \times 32}$
blockDim.x = $N_{warp} \times 32$
Input: int Value[i][j] and float $\lambda$[j],
i=0, 1, ..., gridDim.x-1, j=0, 1, ..., blockDim.x-1
Output: int SumCount[i][k] and float Sum$\lambda$ [i][k],
k=0, 2, ..., $N_{bins}$-1
__shared__ int BufferValCount[$N_{buffer}$][$N_{warp} \times 32$];
__shared__ float BufferLambda[$N_{buffer}$][$N_{warp} \times 32$];
// clear the buffer
for i=0; i < $N_{buffer}$ do
    BufferValCount[i][tid] = 0;
    BufferLambda[i][tid] = 0.0f;
end
// fill in the buffer (Step1.1)
for i=0; i < $N_{buffer}$ do
    int key = Value[blockIdx.x][tid + i * blockDim.x];
    float lambda = $\lambda$[tid + i * blockDim.x];
    int pos =(key %$N_{buffer}$ + tid)%$N_{buffer}$; // hash
    int val = BufferValCount[pos][tid] >> 16;
    int count = BufferValCount[pos][tid] & 0xFF;
    while count>0 and key≠val do
        pos = pos + 1; //another hash function
        pos %= $N_{buffer}$;
        val = BufferValCount[pos][tid] >> 16;
        count = BufferValCount[pos][tid] & 0xFF;
    end
    if count==0 then
        BufferValCount[pos][tid] = key<<16 + 1;
    else
        BufferValCount[pos][tid] += 1;//(key==val)
    end
    BufferLambda[pos][tid] = lambda;
end
__shared__ int HistCount[$N_{bin}$];
__shared__ float HistLambda[$N_{bin}$];
// clear the histogram
for i=tid; i < $N_{bin}$; i+=blockDim.x do
    HistCount[i] = 0;
    HistLambda[i] = 0.0f;
end
// build the histogram (Step1.2)
for i=0; i < $N_{buffer}$ do
    int bin = BufferValCount[i][tid] >>16;
    int count = BufferValCount[i][tid] & 0xFF;
    float lambda = BufferLambda[i][tid];
    if count>0 then
        atomicAdd(HistCount + bin, count);
        atomicAdd(HistLambda + bin, lambda);
    end
end
// copy to global memory (Step2)
for i = tid; i < $N_{bins}$; i+=blockDim.x do
    SumCount[blockIdx.x][i] = HistCount[i];
    Sum$\Lambda$[blockIdx.x][i] = HistLambda[i];
end
```

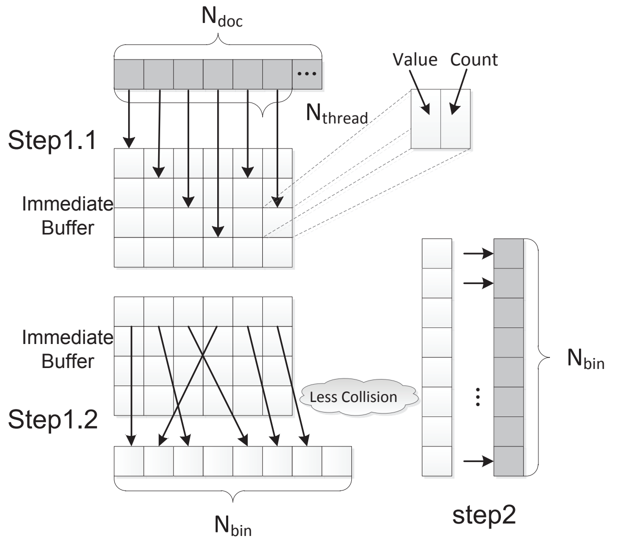

Figure 3: Collision-reduced method kernel

(4 Bytes) of $BufferValCount$ is divided into two 2-Byte fields, one for the value fetched from the input data (referred to as *value-field*), and the other for the count of this value (referred to as *count-field*). $BufferLambda$ is used for storing the λ.

In Step 1.1, each thread iteratively fills its intermediate buffer according to the hash function:

$$index = (key\%N_{buffer} + threadIdx)\%N_{buffer}$$

where *key* is the value of input data fetched from the global memory, and *threadIdx* is the index of the thread. If the calculated index is empty, the *value-field* will be filled with *key*, and the *count-field* will be set to 1. Otherwise, that is, the position is occupied (identified by nonzero *count-field*), if the *key* in the*value-field* equals to the input data, we will simply increase the *count-field* by 1; If not (the hash-collision occurs), we will recalculate the index using *another hash function*, until an unoccupied slot is found. For simplicity, we adopt the *linear probing* method as the another hash function, that is, when hash-collision occurs, we just add 1 to the index.

Obviously, the buffer can reside up to N_{buffer} different values, and the total number of fetched data (the sum of *count-fields*) may be larger than N_{buffer}. However, in order to minimize the divergent warps, we limit iteration time between consecutive buffer flushes to N_{buffer}, so as to ensure that there are always slots available for every input. That means the N_{doc} in this method need to be restricted to $N_{buffer} \times N_{thread}$. Therefore, the output of this kernel is actually several sub-histograms for each feature, and another kernel for merge is needed. The merge kernel costs far less time than the main kernel.

After the intermediate buffers are filled, we update the shared histogram in parallel (Step 1.2). The manner of this step is just like that of the Collision method, Step 1. But in this time, the collisions intensity is much lower than that in previous method. This is because that by Step 1.1, data with the same *key* is placed in different row due to different *threadIdx*s. Thus in Step 1.2, the concurrent threads are less likely to update one single container.

The collision-reduced method performs more stable than the collision method when the data is not so uniformly distributed. Besides, the sequence of the input data also influence the performance of these two method. This is because that on GPUs, it is actually the consecutive 32 threads that are executed simultaneously, so collisions will not occur when two documents with long distance have the same feature value. The collision-reduced method prefers the input data with clustering characteristic (like 1, 1, 1, 1, 0, 0, 2, ...)than that is independently distributed (like 1, 0, 3, 2, 0, 5, 7, ...), since in this case the potentiality of the rearrangement can be full played (see Section 6.2.4).

4.4.2 Data-Specific Optimization

For the latter two methods (collision method and collision-reduced method), we can further reduce their collision intensities if we know where most collisions comes from, which is the case in our data-mining applications. Specifically, since most document features in search engine data are zero (which means most documents do not match the query), we can skip the zeros when updating the histogram, and get the total amount of zeros by subtracting other bin-counters from the document size. This trick, which brings about 2x speedup to the GPU programs (Section 6.2.5), however, is not suitable for the CPU program, since the branches may cause the whole pre-issued instruction pipeline being wasted.

4.5 Hybrid Method

The above three methods have their own advantages in different scenarios. Collision-free method can achieve peak global memory bandwidth when N_{bin} is small. When N_{bin} is large, the collision is inevitable. And collision-reduced method is better than collision method when data is not uniformly distributed. Since in data mining applications, the statistic of data are diverse and the data can be either uniformly distributed or not, three methods needs to be integrated for the best performance.

In our hybrid method, we first determine whether we use the collision-free method, since N_{bin} for each feature is preset. Next, as is mentioned in Section 2, histograms for one set of data have to be recalculated for many times due to the iterative nature of the boosting algorithm. Therefore, we can choose from the collision method and collision-free method by simply comparing the running time of them at the first round (can be seen as preprocessing) with very low cost. The method selection process can be explained by Table 1. We will define the "small-N_{bin}" and the "large-N_{bin}" by experimental results in Section 6. Since shared memory usage must be specified when compiling the kernel code, we need build several kernels, each for one solution. The method-selection information is organized as an array for each kernel to find their corresponding features.

5. EXPERIMENT METHODOLOGY

This section provides basic methodology of our experiment, including the test environment, the performance measurement, the baseline, the experiment procedure and the data.

	"sparse" feature	"dense" feature
small N_{bin}	collision-free method	
large N_{bin}	collision method	collision-reduced Method

Table 1: Method Selection

5.1 Environmental Setup

Our test platform includes Intel Xeon 2.40GHz CPU with 24GB RAM and NVIDIA GTX 580 GPU. The GPU device specification is listed in Table 2. The operating system is Windows 7 64-bit, and we write the code using C/C++ and CUDA. Our code is compiled with the -O2 option. Using CUDA SDK 4.0[developers.nvidia.com], the global memory bandwidth achieved by test kernel is 142 GB/s.

Model	NVIDIA GTX 580
# of Multiprocessor	16
# of cores per Multiprocessor	32
Global memory	1GB
Global memory bandwidth	192 GB/s
Shared memory per Multiprocessor	48kB
Max # of threads per Multiprocessor	1536

Table 2: Device Specification

5.2 Performance Measurement

The exact performance we want for histogram calculation is the *Document Throughput*, which is defined as the number of documents that are processed per second (Gdoc/s). It can be used to compare with the performance of the CPU implementation, which is listed in Section 5.3.

However, the common measurement for memory-bandwidth limited GPU programs is the *Data Throughput* in GB/s [10] [11]. So in Section 6, the performance is represented by the data throughput rather than the document throughput. For convenience, we list the conversion between these two measurements in Table 3. For example, the feature whose value is smaller than 16 is stored in one *half-byte*, and the 98.3 GB/s data throughput equals to 49.2 Gdoc/s. This data organization, which aims at efficiently use the the memory bandwidth, has been completed in prior, and is also applied in the CPU program. The amount of processed data we count is the amount of total feature values (the $Value$ matrix in Algorithm 1), and does not include the λs. Since all the λs form an array, and each element is read for $N_{feature}$ times, the read of λs can be cached and shared by all thread-blocks, and needs far less global memory bandwidth than the read of feature values do.

N_{bin}	Size of $Value$	1(GB/s) means
$2 \sim 16$	0.5 B	2 Gdoc/s
$32 \sim 256$	1 B	1 Gdoc/s
$256 \sim 1024$	2 B	0.5 Gdoc/s

Table 3: Measurement Conversion from Data Throughput to Sample Throughput

Another important measurement for the bandwidth limited kernel is the *achieved Global Memory Throughput* (achieved GMT, in GB/s), which is an indicator of the GPU hardware utilization. For example in the performance analysis for the collision-free method (Section 6.1.2), whose bottleneck is the global memory bandwidth, we will show the achieved GMT to illustrate the effectiveness of our kernel. We get the achieved GMT by NVIDIA Compute Visual Profiler 4.0. The gap between the data throughput and the achieved GMT stems from the read of λs and the write of the histogram result.

The time (the denominator in these three measurements) is the kernel running time. As is stated in Section 2.1, the overhead of data transfer between host CPU and device GPU through PCI-e bus can be considered separately since once feature value is placed on GPU, histogram need to be calculated many times, with only the λs updating.

5.3 Performance Baseline

With our single-thread CPU implementation, the document throughput is about 0.28 Gdoc/s, 0.25 Gdoc/s and 0.3 Gdoc/s for 0.5B, 1B and 2B feature value respectively, independent of the exact number of bins. Multi-core CPU version can achieve linear speedups.

Size of $Value$	Document Throughput (Gdoc/s)	Data Throughput (GB/s)
0.5 B	0.49 Gdoc/s	0.25GB/s
1 B	1.43 Gdoc/s	1.43GB/s
2 B	1.25 Gdoc/s	2.50GB/s

Table 4: CPU Performance for reference

It is difficult to compare our result with previous GPU implementations directly, since in our application, the shared memory usage is doubled due to the existence of λs. Instead, we compare our tuned kernel with the straightforward implementation to investigate the effectiveness of our optimization.

5.4 Experimental Procedure and Data

Our experiments have two goals. First, we evaluate the effectiveness of the our three separate implementations on GPUs. Second, we use these performance data as a guide to the hybrid method. Specifically, we will complete a quantitative version of Table 1.

The evaluation of the histogram calculation on GPUs is complex since the performance potentially has to do with the N_{bin} and the data. For small-N_{bin} histogramming (the collision-free method), the performance is independent of data, since all operations are predictable. First, we show the tuning progress for a specific N_{bin} (namely 64) considering the optimization strategies such as loop-unrolling and saturating the instruction pipeline. Then we give the performance for the collision-free method with all possible N_{bin} and corresponding configurations, while omitting the exhaustive tuning progresses.

For large-N_{bin} histogramming, we need to consider the input data when comparing the performances of the collision method and the collision-reduced method, since both of the methods involve atomic operations, which brings unpredictability to the performance. The exact characteristic of input data that affects the GPU performance is its ability to cause the occurrence that *concurrent threads* updating the same bin-counter. However, this characteristic is difficult to manipulate and is device-dependent, since the range of concurrent threads is not confined to the threads in one warp, but includes all the threads that are active simultaneously, which is determined at the runtime. So instead, we use a more straightforward characteristic, i.e. the standard deviation, to have a glimpse at the predictability of these two methods.

Specifically, first we set N_{bin} to 1024, which is the maximum number of bins required in our application. We tune the two methods separately in Section 6.2.1 and Section 6.2.2 and use normal distributed data with changeable standard deviation as input to investigate effect of data statistic in Section 6.2.3. Then we use a set of real-world data, which is web page features from a commercial search engine, to further compare the effectiveness of these two methods in Section 6.2.4. Finally we change the N_{bin}, and provide the optimal configurations and performances in Section 6.2.5.

In the last, we propose the hybrid method, which can be seen as a summary of the three separate methods, and give its performance in Section 6.3.

6. EXPERIMENTAL RESULTS

6.1 Small-N_{bin}: Collision-Free Method

6.1.1 Tuning for 64-bin Collision-Free Method

Achieve better TLP and ILP: For 64-bin histogram, the size of shared memory for one warp (32 threads) is $64 * 32 * (2B(bin\,counter) + 4B(\lambda)) = 12kB$. So one multiprocessor can accommodate $48kB$ $/12kB = 4$ warps, and thus the theoretical occupancy is $4/48 = 8.3\%$, which is too low to hide the latency of memory access in this memory-intensive application. As is mentioned in Section 4, we further use the GPU registers to achieve better instruction level parallelism. We denote the degree of manual loop unrolling as N_{unroll}. In Figure 4, the y-axis is the measurement of performance (the data throughput; higher is better) and the size of registers used by a thread. The x-axis is N_{unroll}. We can see that the loop unrolling brings up to 2.2x speedup over traditional GPU implementation. But over-unrolling (see when N_{unroll} is 32) leads to performance drop due to the spilling of registers.

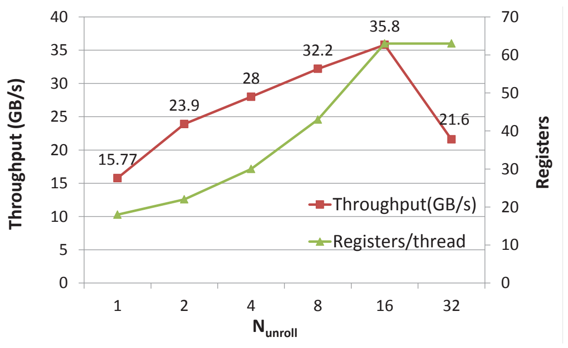

Figure 4: Collision-Free Method, 64-bin, 100k documents, 1k features

Saturating instruction pipelines: Since the size of data affects the effectiveness of instruction pipeline, we also investigate the influence of the grid size (i.e. the number of thread-blocks, or $N_{feature}$ in Algorithm 1) and the load of each thread (i.e the number of documents, or $N_{document}$ in Algorithm 1) on the throughput (Figure 5, Figure 6). We can see that the pipeline becomes saturated when $N_{document}$ is larger than 100k and $N_{feature}$ is larger than 1k. Note that the marks of x-axis in both Figure 5 and Figure 6 increases exponentially.

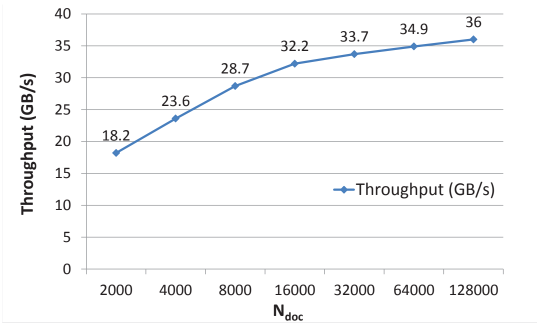

Figure 5: Collision-Free Method, 64-bin, 16 unrolling, 1k features

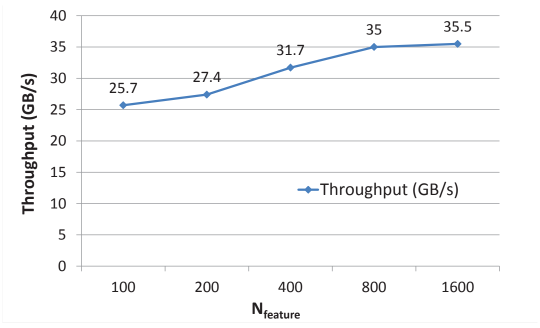

Figure 6: Collision-Free Method, 64-bin, 16 unrolling, 100k documents

6.1.2 *Performance for small-N_{bin} Collision-Free Method*

Using a similar tuning process, we derived the best performances under different N_{bin} in Figure 7. Both the data throughput (the solid line) and the achieved global memory (the dashed line) bandwidth are provided. We can see that with the N_{bin} decreasing, the achieved GMT approaches to 144 GB/s, which is equal to the GMT of the memory copy example kernel provided by CUDA SDK.

The optimal configuration and corresponding occupancy analysis is listed in Table 5. With N_{bin} decreasing, the limiter of occupancy shift from the shared memory usage to the register usage. It is because that when N_{bin} decreases, the number of threads than can be concurrently executed increase, but the consumption of registers by each thread remains unchanged, given the degree of unrolling. So when N_{bin} is large, we can use the register more aggressively by loop unrolling to achieve higher ILP. In contrast, when N_{bin} is small, with experiments we found that we should trade the ILP for better occupancy (TLP), namely cut off the N_{unroll}.

N_{bin}	N_{unroll}	Registers /Thread	SM[1] /Warp (B)	Occupancy Limiter
2	1	19	384	Register
4	2	21	768	Register
8	4	29	1.5k	SM
16	16	63	3k	SM
32	16	63	6k	SM
64	16	63	12k	SM
128	16	63	24k	SM

[1] Shared Memory. Similarly hereinafter.

Table 5: Collision-Free Method, Optimal Configuration

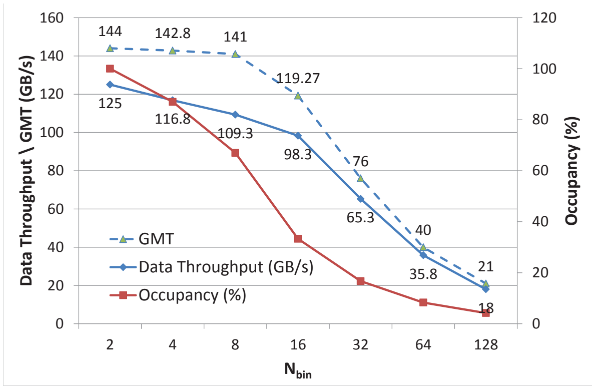

Figure 7: Collision-Free Method, 1k features, 100k documents

6.2 Large-N_{bin}: Collision Method and Collision-Reduced Method

6.2.1 *Tuning for 1024-bin Collision Method*

In the collision method for 1024-bin histogram, the size of shared memory for one full histogram (shared by all threads in one thread-block) is $1024 * (4B(bin\,counter)^2 + 4B(\lambda)) = 8kB$. Thus one multiprocessor can reside 6 thread-blocks concurrently. To achieve 100% occupancy (1536 threads per stream multiprocessor), the number of threads in one thread-block need to be larger than (1536 / 6 =) 256 (8 warps). We also try to use loop unrolling to further improve the parallelism. As you can see in Figure 8, the effect of loop unrolling is more subtle and complex than that in the collision-free method. The reason is that in collision method, we can easily achieve high TLP. So the slight benefit of ILP improvement may be overwhelmed by the overhead such as registers spilling. From figure 8 we can see that the optimal performance is achieved when we set 8 warps per block, and $N_{unroll} = 1$. Note that we use uniformly distributed input, which is the best case scenario of the collision method, and we achieve up to 45.8 GB/s data throughput.

To illustrate the bottleneck of the collision method, we further use the degenerate distributed data (i.e all the input elements are set to the same value), and get only 0.38 GB/s data throughput, which is even lower than that of the CPU

[2]Since the finest-grained atomic operation supported by CUDA 4.0 is 4Byte Add, we use the integer bin counter.

program. Then we replace the shared histogram *atomicAdd* with common addition operation. In this way the serial execution introduced by the atomic operation is removed, but the result is incorrect. The modified kernel achieves 117 GB/s data throughput and 160 GB/s GMT. Therefore we can safely conclude that the bottleneck of the collision method is the overhead of atomic operation (more than two order of magnitude drop in performance).

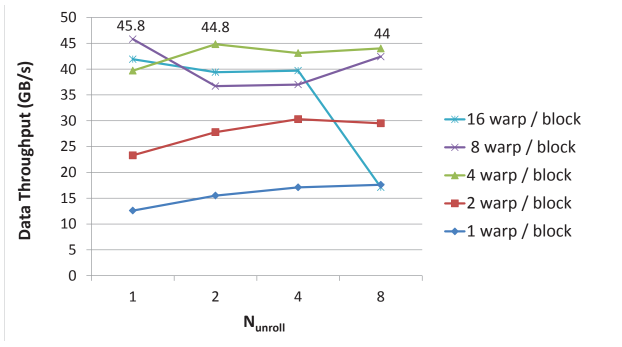

Figure 8: Collision Method, 1k features, 100k documents

6.2.2 *Tuning for 1024-bin Collision-Reduced Method*

The tuning for the collision-reduced method aims at improving the occupancy while keeping enough work load. The changeable parameters involved in this method include the number of warps per thread-block (N_{warp}) and the size of the extra buffer (N_{buffer}). For 1024-bin histogramming, the kernel performs best when $N_{buffer} = 10$ and $N_{warp} = 6$. The size of shared memory used by one thread-block is $32 * N_{warp} * N_{buffer} * (4B(Value + Count) + 4B(\lambda)) + N_{bin} * (4B^3 + 4B(\lambda)) = 23kB$. The achieved occupancy is $\frac{6*[\frac{48kB}{23kB}]}{48} = 25\%$, and for uniformly random data, the throughput is 9.86 GB/s, which is far less than that of collision method. But remember the input is uniformly distributed, favoring the collision method. for degenerate distributed data the throughput is 5.57 GB/s, 14 times faster than that of the collision method.

6.2.3 *Performance on Normal Distributed Data, 1024-bin*

To get a glimpse at the effects of the data characteristic, we generate the input data with a independent normal distribution with 512-mean and σ-standard deviation (σ is changeable). When $\sigma = 0$, the input is degenerated distributed, and when $\sigma = \infty$ the input is uniformly distributed. In Figure 9, the x-axis is the σ of the input, and y-axis is the data throughput and the histogram of the standard deviation of the real-world data. We can see that the performances of both the collision method and the collision-reduced method improves with the increase of σ. When $\sigma = 0$, the data throughput is 0.38GB/s and 5.57 GB/s for the collision method and the collision-reduced method respectively. And their throughput approaches to uniformly random case (i.e. 45.8 GB/s and 9.86 GB/s). The collision-reduced methods outperforms the collision method when $\sigma < 10$. Note that the real-world data is not *independently gaussian* distributed, so the input with large standard deviation does not necessarily favor the collision method, which will be shown later.

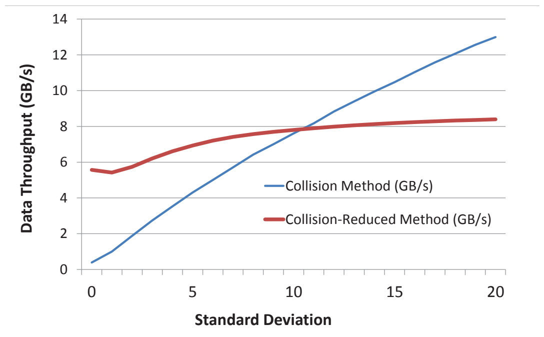

Figure 9: Collision Method and the Collision-reduced Method for Gaussian distributed data with different standard deviation

6.2.4 *Performance on the Real Data, 1024-bin*

Figure 10 compares the collision method and the collision free method for real-world data from commercial search engine and also with $N_{bin} = 1024$. The x-axis is the feature index. The y-axis is the data throughput and standard deviation of the data. For clarity, in Figure 10 we sort the features according to their running time of the collision method. The performance for a single feature is obtained by all thread-blocks calculating the same feature. The collision-reduced method outperforms the collision method for 80% of the features, and above 50% features achieve about one order of magnitude performance gain. The average performance can be found in Section 6.2.5.

We can also see that for real data, the collision method performs poor even when the σ of data is larger than 10. This phenomenon has to do with the sequence of data, since real data is not independent Gaussian distributed. Instead, the feature values from search engines are more likely to be clustered (not mixed up well). This characteristic potentially takes full advantage of the data rearrangement in the collision-reduced method.

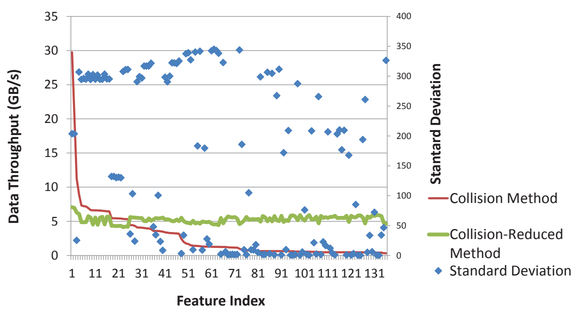

Figure 10: Comparison of the Collision Method and the Collision-reduced Method for single features

6.2.5 *Performance for large-N_{bin} on the Real Data*

In practice, different thread-blocks process different features. Figure 11 shows the data throughput of the collision method,

[3]ditto

the collision method and their hybrid method with different N_{bin} as dashed-border columns. An important characteristic of the feature values is that the number of zeros are large. Skipping the zeros when updating the shared histogram will eliminate much collisions. We denote the throughput of the zero-skipped collision method, the zero-skipped collision-reduced method and their hybrid method by solid-border columns. From Figure 11 we can see that for real data (no zero-skipping), collision-reduced method outperforms the collision method by a factor of 2 to 4.5, and skipping zeros is more beneficial for the collision method. The hybrid method gains 10% to 20% speedup compared to the best homogeneous method. The optimal configuration and corresponding occupancy is listed in Table 6.

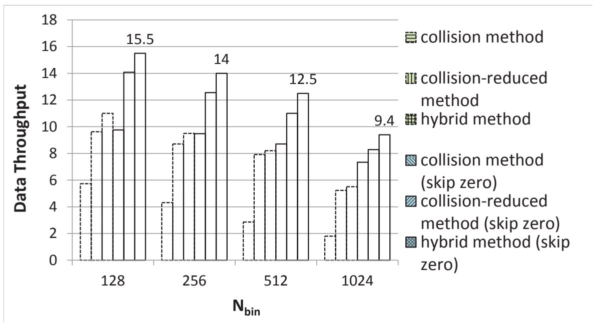

Figure 11: Collision Method and Collision-reduced Method and their Hybrid Method in practice

N_{bin}	collision-reduced Method			collision method
	N_{warp}	N_{buffer}	Occupancy	N_{warp}
128	2	10	33.3 %	2
256	2	8	33.3 %	2
512	3	5	75 %	2
1024	6	10	25 %	4

Table 6: Collision-Reduced Method & Collision Method, Optimal Configuration

6.3 Hybrid Method

Table 7 summarizes the performance under different N_{bin}. The percentage is from the real data set from a commercial search engine. When N_{bin} is from 2 to 128, we use the collision-free method (denoted as 1), and when N_{bin} is from 256 to 1024, we use the hybrid method of the collision method and the collision-reduced method (denoted as 2). The test of such hybrid method on this data set shows 13 GB/s data throughput, and gains 13x speedup compared to the single thread CPU program (0.95 GB/s).

7. CONCLUSION

In this work, we propose a new method for weighted histogramming on GPUs using CUDA. It shows 2x speedup over traditional method on data set from a commercial search engine. We also optimize the traditional methods and propose a hybrid method which dynamically chooses the best implementation for each input. Such a method benefits histogram calculation with both large and small number of bins while preventing the worst case, and shows 13GB/s data throughput and 13x speedup over single-core CPU implementation.

N_{bin}	2	4	8	16	32
DT[1](GB/s)	125	116.8	109.3	98.3	65.3
Percentage (%)	6.9	0.1	6.4	4.3	1.0
Method	1	1	1	1	1
N_{bin}	64	128	256	512	1204
DT (GB/s)	35.8	18	14	12.5	9.4
Percentage (%)	1.7	15.4	0[2]	5.8	58
Method	1	1	2	2	2

[1] Data Throughput.
[2] We get the performance by generating fake features from 1024-bin data.

Table 7: Method Selection and Performance Summary

8. REFERENCES

[1] T. Brosch and R. Tarn. A self-optimizing histogram algorithm for graphics card accelerated image registration. In *Medical Image Computing and Computer Assisted Intervention (MICCAI) Grid Workshop*, pages 35–44, 2009.
[2] C. Burges. From ranknet to lambdarank to lambdamart: An overview. Technical report, Technical Report MSR-TR-2010-82, Microsoft Research, 2010.
[3] C. Burges, R. Ragno, and Q. Le. Learning to rank with nonsmooth cost functions. *Advances in neural information processing systems*, 19:193, 2007.
[4] Y. Freund, R. Iyer, R. Schapire, and Y. Singer. An efficient boosting algorithm for combining preferences. *The Journal of Machine Learning Research*, 4:933–969, 2003.
[5] P. Glaskowsky. Nvidia fermi: The first complete gpu computing architecture. *NVIDIA Corp*, 2009.
[6] A. Jörgensen. Adaboost and histograms for fast face detection. 2006.
[7] J. Nickolls and W. Dally. The gpu computing era. *Micro, IEEE*, 30(2):56–69, 2010.
[8] C. Nugteren, G. van den Braak, H. Corporaal, and B. Mesman. High performance predictable histogramming on gpus: exploring and evaluating algorithm trade-offs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, page 1. ACM, 2011.
[9] C. Nvidia. Compute unified device architecture programming guide. *NVIDIA: Santa Clara, CA*, 83:129, 2007.
[10] V. Podlozhnyuk. Histogram calculation in cuda. *NVIDIA Corporation White Paper*, (November), 2007.
[11] R. Shams and R. Kennedy. Efficient histogram algorithms for nvidia cuda compatible devices. In *International Conference on Signal Processing and Communication Systems*, 2007.
[12] T. Sharp. Implementing decision trees and forests on a gpu. *Computer Vision–ECCV 2008*, pages 595–608, 2008.
[13] P. Viola and M. Jones. Robust real-time object detection. *International Journal of Computer Vision*, 57(2):137–154, 2002.
[14] V. Volkov. Better performance at lower occupancy. In *GPU Technology Conference (GTC)*, 2010.