Modestly Faster Histogram Computations on GPUs

Shawn Brown Jack Snoeyink
UNC Chapel Hill
201 South Columbia St
Chapel Hill, NC 27599-3175
shawndb@cs.unc.edu

ABSTRACT

We present TRISH, a 256-bin histogram method for byte data that runs up to 50% faster than previous GPU methods for random data and 2-4× faster for image data. The performance gains come from reducing total cycle counts. Reducing cycles comes from improving 1) thread level parallelism (TLP), 2) instruction level parallelism (ILP) and 3) software vector parallelism (VP). TLP is improved by increasing occupancy from 2 to 3 thread blocks, achieved by compacting 'per thread' histograms in shared memory, and by using register arrays. ILP is improved by increasing independent instructions via loop unrolling by a factor of $k = [1..63]$ and batching operations into groups of four. VP is supported by compacting bin counts into four 8-bit quads per 32-bit element and reducing binning & accumulating instructions by working with 32-bit elements as overlapping 16-bit pairs instead of 4 individual bytes. Note that TRISH is a deterministic algorithm that avoids atomic operations and gives performance that is data independent.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures

General Terms

Algorithms, Performance

Keywords

Histogram, CUDA, GPU, Parallel Processing

1. INTRODUCTION

Histograms, first defined by Pearson [8], are well-known; their computation takes as input a set of n data values, V, from a range $R = [min; max)$, and a number of bins. m. It subdivides the range R into m equal sized sub-ranges or bins, $r_i = [a_i; a_{i+1})$, covering the original range without overlap, partitions the data values V into the m bins, and maintains counts b_i for the number of data values that fall into each bin r_i, then outputs the frequency counts b_i for each bin. A histogram is often graphically rendered as a bar-chart of these counts. The shape of the chart gives information about the frequency distribution of the data.

The histogram methods discussed in this paper focus on the special case of 256-bin histograms for (8-bit) byte data, useful for image processing, text processing, and other applications. Although histogram methods are straightforward to implement on a sequential CPU (see Figure 1), it has proven difficult to adapt histograms for use on many-core processors such as GPUs. Previous GPU histogram methods give correct results, but achieve only $6 - 15\%$ of the theoretical peak throughput of modern GPU hardware, such as Nvidia Fermi or Tesla cards. Our method improves modestly on previous results, achieving ~21% of peak throughput.

1.1 Parallelism

Of the many forms of parallelism [2], we focus in this paper on data parallelism, instruction pipelining, and vector parallelism.

Input: V = array of n *bytes* to be binned
Output: *bins* = array of m= 256 *bin* counts

```
integer bins[256] = 0;    // zero counts
foreach idx in [0..n-1]   // count bytes
  bins[V[idx]]++;
end idx
```

Figure 1. 256-bin histogram method on a CPU.

Data Parallelism: Data parallelism partitions a large set of n data elements across p processors with each individual processor typically responsible for $O(n/p)$ elements [2]. This results in a speedup of p for linear $O(n)$ single CPU algorithms like computing histograms.

Instruction Pipelining: Instruction pipelining increases instruction throughput by overlapping the execution of instruction processing stages. The goal is to keep the hardware busy by keeping the pipeline stages full and to avoid hazards that cause stalls and wasted idle cycles [2]. On Fermi cards, pipelining is measured using the instructions per cycle (IPC) counter. We can use both TLP and ILP to improve instruction pipelining throughput.

Thread Level Parallelism (TLP): Each SM on a Fermi card contains 16 fully pipelined integer arithmetic logic units (ALU) and floating point units (FPU) [2]. The Fermi hardware uses thread level parallelism (TLP) to extract independent instructions across multiple threads. First, since each warp contains 32 threads but only 16 ALU's, the warps are partitioned into 2 batches of 16. This means each warp instruction is issued over 2 cycles. Second, multiple independent warps (up to 48 on Fermi class cards) are kept resident on each SM so that stalls occurring in one warp can be hidden by scheduling instructions from another independent warp. TLP is measured using the *occupancy* metric on GPUs.

Instruction Level Parallelism (ILP): Traditional CPU hardware pipelines use a combination of dynamic hardware and static software techniques to keep the ALUs and FPUs busy -- including loop unrolling, branch prediction, scoreboarding, etc. [2]. We will focus on providing long sequences of independent instructions for improved ILP.

Vector Parallelism (VP): Also known as vectorization is the idea of processing multiple data elements at once with a single instruction [2]. Because Nvidia GPU hardware is single-instruction-multi-threaded (SIMT), and not Vector parallel, we simulate this idea in software by working with four 8-bit bytes as a single 32-bit element.

1.2 GPU Performance

We assume that readers are comfortable with GPU architecture and programming concepts. To achieve good GPU performance, we use the standard ideas from Nvidia of coalesced memory access, avoiding bank conflicts, and minimizing branch divergence.

We use two primary measures for GPU performance. The first measure is *I/O throughput*, which measures the number of 8-bit

bytes processed per second. The second measure is *Total Cycles* (TC), which measures the total number of machine cycles to complete each method. $TC = II/IPC$, where II = total *instructions issued* and IPC = average *instructions* retired *per cycle*. Using the TC measure, we seek to improve performance by either reducing the total instructions issued (II) or by increasing the instructions per cycle (IPC).

To achieve better GPU performance in our TRISH histogram, we also use several ideas from other sources. Micikevicius [5, 6] talked about finding and removing GPU performance bottlenecks by carefully measuring and analyzing performance. Merrill [3, 4] used *register arrays* for better performance and memory utilization by partitioning array elements across the registers of multiple threads in a thread block. Harris [1] showed how *iterative improvements* to a reduction primitive eventually resulted in a 30× speedup. Volkov [11] showed how ILP can improve overall GPU performance even at low thread occupancy (reduced TLP.)

2. RELATED WORK

We aim for our GPU Histogram methods to be correct, fast, and predictable. We verify correctness by comparison against a baseline CPU method. We achieve faster performance by porting our original CPU methods onto GPUs to take advantage of the large number of multi-cores via data parallelism and by seeking to minimize the total cycles that our GPU method takes to complete. We also seek predictable methods where performance is not influenced by the underlying data distribution.

I will briefly describe two previous GPU histogram methods. There is also a third, by Yang et al. [12], but this histogram method is slow (0.7 GB/s throughput on G80 class GPUS).

2.1 Podlozhnyuk's Histogram Method

Victor Podlozhnyuk provides methods for 64-bin and 256-bin histograms in the most recent CUDA software development kit [9]. (Shams [10] has generalized Podlozhnyuk's method to support 32-bit inputs and a range of bin sizes.) These methods target older GPUs with 16 KB of shared memory per SM. Thus, 64-bin histograms can be created in shared memory for each thread. Since 256-bin histograms for each thread would not fit, he uses a novel 'per warp' method in which thread blocks with 6 warps (of 32 threads each) create 6 histograms in shared memory. After scanning the data, he then accumulates these per-thread or per-warp histograms into the final histogram. Using this 'per warp' memory layout, all 32 threads in each warp must compete to increment and update shared histogram bins. To maintain correct behavior during collisions between threads, the method must use either hardware atomics or the author's novel software tagging scheme: the top 5 bits of a counter are reserved for a thread ID, which each thread sets as it tries to increment a bin count. Since hardware guarantees that a single thread will always win despite collisions, each thread can check the tag to ensure that its increment happened, or try again. Of course, this means that worst-case behavior (many thread collisions) can be 8-32× times slower than best-case behavior (no thread collisions). This method uses 192 threads per block with 8 concurrent blocks per SM and thus achieves full occupancy ($100\% = 1536/1536$). Most of its performance comes from TLP but this is limited by the need to resolve intra-warp thread collisions.

Unfortunately, this 256-bin method performs at only 15.0% of peak throughput on modern GPUs (GTX 580) for random, uniformly distributed data. Moreover, performance is data dependent -- the data distribution determines the number of thread collisions, which determines overall performance. As our experiments show, performance can indeed degrade by factors of > 30.

2.2 Nugteren's Histogram Method

More recently, Nugteren et al. [7] reported an algorithm for a fixed-size (2048×2048 bytes) that is independent of the data distribution. They claimed that on a GTX 470 they achieved 56% better throughput than Podlozhnyuk's method, but we have not seen consistent improvement in our tests on the GTX 580, 480, and 560M. Two factors may explain this: in our tests, we replace Podlozhnyuk's software tagging with hardware atomics for a 50% speedup of his algorithm, and the uniform distribution that is our main test case is kinder to data dependent algorithms than images with correlated pixel values that were probably used by Nugteren et al. Our experimental results (e.g., Figure 7) are consistent if both factors are taken into account.

They use the larger shared memory (48 KB) of modern GPUs to implement 'per thread' histograms. Histogram bin counts are stored using two 16-bit bins per 32-bit word, so 128 words suffice to store an entire 256-bin histogram. With 32 threads per thread block, 16 KB of shared memory is sufficient to store individual histograms for each thread block. This enables 3 thread blocks per SM to run concurrently. Unfortunately, this results in low occupancy ($6.25\% = 96/1536$) and therefore their method does not fully utilize the GPU pipelines.

Their histogram GPU kernel works in a straightforward way: data is partitioned across all the threads in a cooperative thread array (CTA), each thread bins and counts all its assigned data. After all input data is binned, the 'per thread' histograms are summed to create a 'per block' histogram in global memory. A *tail* GPU kernel is then executed to sum the 'per block' histograms into the final histogram.

In our experiments, Nugteren's code achieved about 5.4% of peak throughput on the GTX 580 for a fixed image size of 2048×2048 bytes. Performance for this method is predictable, since the method is deterministic and not influenced by the underlying data distribution.

3. THE TRISH METHOD

Like these previous GPU histogram methods, our method also supports coalesced memory accesses for efficient I/O, stores intermediate results in shared memory, and uses simple indexing. The TRISH (Threaded Registers Independent Strided Histogram) method is similar to Nugteren's method with several simple ideas that improve overall performance. Where possible, we report on the improvements in throughput (GB/s) afforded by each idea to indicate their contributions to overall performance, although we defer the description of the experimental setup to section 4.

We achieve better throughput by reducing the total cycles required to complete our method ($TC = II/IPC$). We improve pipelining, as measured by instructions retired per cycle (IPC), via a combination of thread- and instruction level parallelism (TLP and ILP). We reduce instructions issued (II) by simplifying code and applying vector processing (VP) ideas. Better TLP is achieved by building compact per-thread histograms in shared memory. Better ILP is achieved by increasing independent instructions by loop unrolling and batching operations into groups of four. Better VP is achieved by working with 16-bit pairs or 8-bit quads as single 32-bit elements instead of separately as 8-bit bytes.

Like Nugteren's method, we use a deterministic 'per thread' histogram algorithm and have no need for the atomics to resolve intra-warp thread collisions of Podlozhnyuk's method. Thus our method is data independent.

We therefore introduce our Threaded Registers Independent Strided Histogram (TRISH) method.

3.1 Improving TLP

Building compact 'per-thread' histograms in shared memory improves thread-level parallelism at the cost of more frequent accumulation passes.

Compacting Histograms: We compact a histogram by storing four 8-bit bin counters per 32-bit word. Since each bin counter is a single byte, 64 words suffice to store an entire histogram for a thread (see Figure 2).

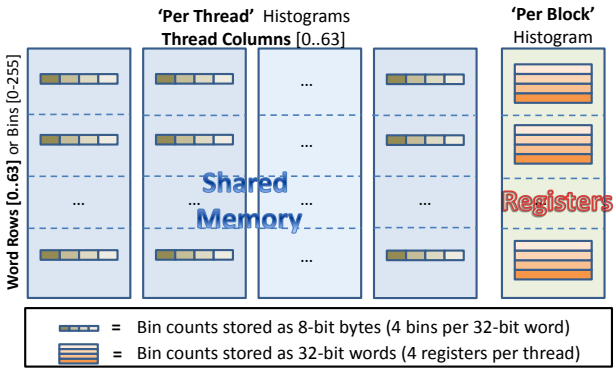

Figure 2: In TRISH, each thread has a 256-bin histogram stored as a column of 64 words (four 8-bit bin counts per word). Each thread maintains four 32-bit registers in the 'per block' histogram stored in registers.

Since we use 64 threads per thread block, 16 KB of memory suffices to store all `per thread' histograms for each thread block. Since each Fermi class SM has 48 KB of shared memory, this enables 3 concurrent thread blocks per SM, achieving a thread occupancy rate of 12.5% (192/1536). This is a much lower occupancy than Podlozhnyuk's method but double the occupancy of Nugteren's method. Furthermore, having 2 or more warps per SM enables the dual issue feature on Fermi class GPU cards, enabling the scheduling of two warps of instructions at a time onto GPU pipelines.

The small range [0..255] of bytes as bin counters means that we must act earlier to prevent overflow. Thus, we accumulate 'per thread' histograms into a 'per block' histogram on a regular basis. The 'per block' histogram uses 32-bit bin counters, so overflow within the 'per block' histogram is not a concern. We treat data elements as 32-bit words instead of individual 8-bit bytes. To be safe, we accumulate our 'per thread' results after binning at most 63 words per thread. That is, we assume all 252 bytes in those 63 words can bin into the same counter.

'Per thread' histograms are accumulated into the 'per block' row sum histograms by a simple reduction (linear serial scan) of the bin counts. As part of the reduction, we reset the 'per thread' bin counts to zero. Since there are 256 bins but only 64 threads per thread block, each thread must accumulate 4 bins.

Direct indexing during the `per block' row sum accumulation would result in a 32-way bank conflict per warp that would hurt performance, so we stagger starting indices by thread ID and index modulo block size (circular indexing), see Figure 3.

```
// Staggered Start
currIdx = threadID;
…
// Circular Indexing
nextIdx = mod(currIdx+1, BlockSize);
```

Figure 3. Staggered Indexing avoids bank conflicts.

Register Arrays: Since each bin counter is completely independent, we can safely store the entire 'per block' histogram as a register array. We partition the entire 256-bin histogram across all the 64 threads in each thread block. Each thread maintains its four 'per block' bin counters in registers.

3.2 Improving ILP

Since our method results in low occupancy (12.5%), we can't fully take advantage of TLP to keep the hardware pipelines busy. Instead, we fall back on well-known ILP techniques of 1) loop unrolling and 2) batching four runs of independent instructions.

Loop Unrolling: Define work per thread, k, as the number of 32-bit words that each thread counts in bins before looping to the next chunk of work and repeating. To prevent overflow we can process at most 63 words (252 bytes) before accumulating results, so k is chosen in the range [1..63]. Larger values of k amortize the cost of loop overhead across multiple elements and increase the pool of independent instructions that the compiler and the hardware can potentially harvest during instruction pipeline scheduling.

Batching Instructions: We further batch the k binning operations per loop into groups of four to help the compiler harvest even more independent instructions. We try to make the compiler's job easier by giving each run of independent instructions (within a batch of four) its own set of named variables. This batching does increase register pressure on our kernels; Using CUDA 4.0, The TRISH method uses 36 registers per thread, between Podlozhnyuk's 18, and Nugteren's, which spills out of 42. Since thread occupancy is already limited by shared memory constraints, this turns out not to be a problem for our method.

3.3 Improving VP

As much as possible we handle the input data and initial count arrays either as 32-bit quads of bytes or as two alternating pairs of bytes, because this reduces the arithmetic operations involved by up to factors of 4 or 2, respectively.

Accumulation Optimizations: We normally think of the four 8-bit bins in a 32-bit word as a quad layout [3,2,1,0]. However, we can also consider the layout as 2 alternating pairs [*,2,*,0] and [3,*,1,*]. We can extract and work with these alternating pairs using bit shift and mask operations to increase the arithmetic range from 8-bits to 16-bits before overflow becomes a problem. Changing the accumulation function to work with pairs instead of singletons during the 'per block' row sum operations improved throughput performance by about 5% on the GTX 580 (from 38.1 GB/s to 40.3 GB/s).

Binning Optimizations: It is natural to extract individual bin indices by multiplying the byte value by the thread block size and then adding in the thread offset. How-ever, using bit manipulation, we can multiply alternating pairs of bytes, and, assuming fixed powers of 2 for our table sizes, we can replace multiplies and divides by bit shift and mask operations, which in turn allows us to combine multiple shifts into a single instruction. See the accompanying source code for details.

3.4 Picking the best 'k' value

What k-value (work per thread) in the range [1..63] gives best performance? Recall that 63 is the maximum number of elements we can safely process before overflow becomes a possibility. Picking the best k-value is complicated by several limitations: Loop overhead, ILP, and the RowSum efficiency ratio.

Loop Overhead: We amortize the cost of loop overhead across the work of binning k elements per loop. This suggests that we should favor larger values of k.

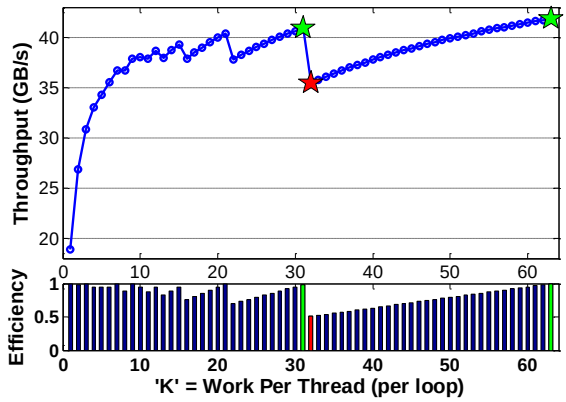

Figure 4: ***Upper:*** **Impact of work per thread on I/O throughput for $k \in [1, 63]$. Best results are for $k \in [31, 63]$.** ***Lower:*** **Efficiency for $k \in [1, 63]$. Max Efficiency = 1.0 (63 elements binned per RowSum) at $k = 1, 3, 7, 9, 21$, and 63.**

ILP: At low values of k, we don't unroll the loop enough to unlock sufficient independent instructions for the warp schedulers to take advantage of during instruction pipelining (see Figure 4, for $k = [1..4]$). This suggests we should favor larger values of k for improved ILP.

RowSum Efficiency: In order to avoid overflow, we frequently accumulate all 'per thread' counts into 'per block' counts and reset the 'per thread' counts to zero. This accumulation operation is expensive but fortunately is amortized across the cost of binning each individual element. Since, 252 bytes (63 elements) is the most we can bin before accumulating, the ideal ratio would be 63 bin operations per accumulation operation. However, since we test for overflow only at the top of each loop, the bin/RowSum efficiency ratio is actually the max number of iterations before overflow becomes possible, $Efficiency = (\lfloor 63/k \rfloor k)/63$.

In the worst case, $k = 32$, we process one loop of 32 elements (128 bytes) and at the top of the very next loop we have to perform a RowSum before binning and counting any more elements because overflow is now possible, $(32 + 32) > 63$, this results in an efficiency of only $32/63 = 0.508$. Compare this with $k = 31$, we process 31 elements and on the next loop, we can process another 31 elements before having to perform a RowSum at the top of the 3rd loop, $(31 + 31) \leq 63$, achieving an efficiency of $(2 \times 31)/63 = 0.984$. That is, we must accumulate almost twice as often for $k = 32$ than for $k = 31$. The ideal efficiency ratio of $1.0 = (63/63)$ is achieved at the values $k = 1, 3, 9, 21$, and 63, which are the values of k that divide 63 evenly.

We tested all values of $k = [1..63]$ to see how these limiting factors combine (see Figure 4.), and discovered that low values, $k \leq 4$, gave the worst performance because of high loop overhead and low potential for improving ILP, For example, at $k = 1$ throughput was only 18.92 GB/s. At higher values of k, there is a strong correlation between the RowSum efficiency and I/O throughput performance as accumulating is expensive. We found that $k = 31$ and $k = 63$ gave the best throughput results at 40.91 GB/s and 41.86 GB/s respectively. At $k = 32$, performance drops to 35.44 GB/s due to the frequent accumulation operations.

Though throughput for $k = 31$ is slightly slower than for $k = 63$, we recommend $k = 31$ as the better overall choice because its performance is faster overall for small to medium values of n.

3.5 Method Overview

Here is a brief high-level overview of the 256-bin TRISH code. A CPU host function implements our TRISH method by dealing with setup, invoking the GPU kernels, and cleaning up resources.

The actual GPU histogram method is a two-step process using two GPU kernels, a ***Count*** kernel and a ***BlockSum*** kernel. We briefly describe the ***BlockSum*** kernel first as it is so simple. The ***BlockSum*** kernel simply sums in global memory the results of the individual 'per block' histograms generated by the ***Count*** kernel to obtain the final histogram.

Count Kernel

Inputs: $\lfloor n/nBlocks \rfloor + [0,1]$
32-bit values to bin & count
Outputs: 'Per Block' counts ($256 \times nBlocks$)

I/O Summary:
$\lfloor n/32 \rfloor$ coalesced reads of input values
$\lceil (256 \times nBlocks)/32 \rceil$ coalesced writes of counts

Algorithm: (In Parallel, 64 threads)

```
Compute # of rows & left over row
while (more rows)
   if overflowCount ≥ (63 – K)
     Accumulate block counts
     overflowCount = 0;
   end if
   repeat K times  // fixed K in [1..63]
      Read 32-bit word of data
      Bin 32-bit word as 4 bytes
   end repeat
   overflowCount += K;
end while
if ('left-over' row)
  … Similar to main loop plus range checking
end if
if overflowCount > 0
   Accumulate block counts
end if
Write out block counts
```

Figure 5. GPU Count Kernel, which computes the 'per block' histograms.

The ***Count*** kernel (see Figure 5) partitions the data into multiple fixed-size chunks as described in section 5.2. Each thread block computes its own 'per block' histogram for its assigned data elements. The code has 4 main sections: The setup code figures out how many fixed size rows (via the stride) to process in the main loop and checks if we have any *left-over* data that needs to be processed with careful range checking. A *wrapup* section then

accumulates any extra 'per thread' counts and writes out the final 'per block' counts to global memory in a coalesced manner.

In the main loop, each thread in a warp (of 32 threads) reads in a 32-bit word from global memory in a coalesced manner and then bins each of the four bytes of that word into the 'per thread' histogram array in shared memory. An overflow counter is incremented after each inner loop of k elements is processed. On detecting possible overflow, the code accumulates and resets the 'per thread' counts into a 'per block' histogram stored in a register array. The accumulation function treats 32-bit quads as two alternating pairs for better performance via VP.

If there is any *left-over* data after the main loop is finished, then that data is also binned and accumulated but with more careful range checking to prevent access errors.

4. PERFORMANCE RESULTS

Due to space constraints, all reported tests were performed on the GTX 580 using the exact same computational environment. We repeated these tests on GTX 480 and GTX 560M cards with similar results; while the histogram code runs on a Tesla card, we had insufficient time to port our testing code to Linux.

Hardware: CPU = i7-920 @ 2.66 GHz, RAM = 12 GB, GPU = GTX 580 (16 SMs, 512 SPs, 1.5 GB memory, 192.4 GB/s peak throughput).

Software: GPU API = CUDA 4.0 (64-bit), LANG = C++, IDE = VS 2010 (64-bit), OS = Windows 7, SP1 (64-bit)

Data Size: n = [8192 – 268,435,456], in increasing powers of 2.

All timings are averages derived by using 100 runs of each test. I/O throughput is measured in gigabytes per second (GB/s) by counting the number of bytes processed in billions and dividing by the average time to process in seconds.

4.1 Direct Comparison

Figures 6 & 7 plot throughputs by input size for Podlozhnyuk's 'Per Warp,' Nugteren's 'Per Thread,' and our TRISH histogram methods. Because Podlozhnyuk's method is data dependent, Figure 6 shows results for three data sets: a *worst-case* dataset of "all zeros," which causes all threads in each warp to collide, a *best-case* linear dataset, carefully constructed to prevent any thread collisions, and a *uniform random* dataset. These dataset differences cannot be easily distinguished in the other two methods, confirming their data independence.

Test Conditions: For Podlozhnyuk's method we enabled hardware atomics, which improves performance by 50% over his software tagging approach. Nugteren's original method is statically compiled for a fixed 2048×2048 image size. We tweaked his code slightly too also allow sizes that are 4, 16, or 64 times larger. Generalizing further would add range checking that would decrease throughput. The TRISH method bins 31 elements per thread ($k = 31$) before looping, for a cooperative stride factor of 95,232 across all 48 thread blocks in the CTA. The grid size was set to 48 to give 3 concurrent blocks per SM on all 16 SMs on the GTX 580. A larger grid size resulted in a larger last row size, more range checking, and slightly slower overall performance.

Data Results: The TRISH method has better throughput than the previous methods for large n. For example for $n = 67{,}108{,}864$, TRISH achieves 40.53 GB/s. Nugteren's method achieves 11.25 GB/s. Unlike the other two, Podlozhnyuk's method is highly data dependent: for *best-case*, *uniform random*, and *worst-case* datasets achieves 43.61 GB/s, 27.43 GB/s and 0.48 GB/s, respectively. For uniform random data, TRISH is 3.6× and 1.48× faster than Nugteren's and Podlozhnyuk's methods, respectively.

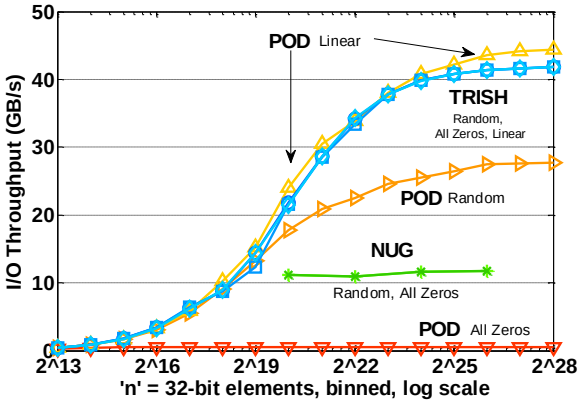

Figure 6: Throughput for Podlozhnyuk's (POD), Nugteren's (NUG), and our TRISH methods on three datasets -- *all zeros*, *uniform random*, and *linear*. We base comparisons on uniform random results, although these will be optimistic for POD. Performance measurements were taken on a GTX 580 card, all software was built using the CUDA 4.0 platform; results on other tested platforms were similar.

Images: In Figure 7, We compared all 3 methods on three standard greyscale images (Lena, Raft, MRI) obtained from sci-en.stanford.edu/pages/labsite/scien_test_images_videos.php. Images are tiled to make sizes from 256^2 through 4096^2. We included an RGB Raft Image for comparison, which was actually stored as RGBA with the alpha channel set to all zeros -- a near worst case for Podlozhnyuk's method. For all images, the coherence of nearby pixel values hurts Podlozhnyuk's method, but does not affect the other two, indicating that the uniform random tests are overly optimistic for his method's practical performance.

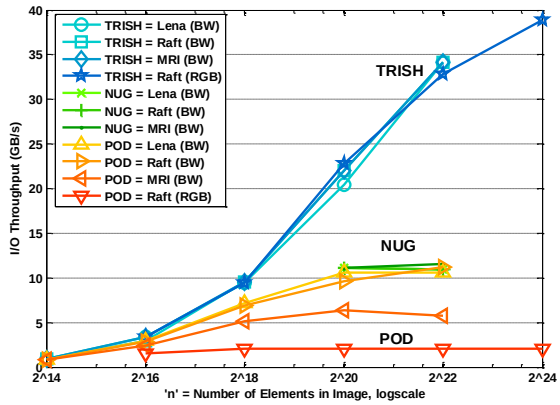

Figure 7: Throughput for Podlozhnyuk's (POD), Nugteren's (NUG) and our TRISH methods on 3 greyscale images (Lena, Raft, MRI) and one RGBA image (Raft). Each is tiled to sizes 256^2 - 4096^2. Performance measurements were taken on a GTX 580 card, software built on the CUDA 4.0 platform.

	Time (ns)	Through-put GB/s	Through-put Perf Ratio	Registers	Shared Mem	Occupancy	Warps/ Cycles	Ins/Byte
TRISH	6,545.41	40.02	**1.00**	36	16,384	192/1536	5.94	11.71
POD	9,745.44	27.03	**1.48**	18	6,144	1536/1536	44.47	23.42
NUG	22,631.20	11.25	**3.56**	42	16,384	96/1536	2.98	6.11

	IPC (2.0 max)	II	Bank Conflicts	Cycles II/IPC Ratio	Cycles Perf Ratio	Branches	Divergent Branches	Instruc-tion Re-plays
TRISH	1.18	6,140,046	0	**5,203,429**	**1.00**	8,844	0	0.00%
POD	**1.58**	**12,292,604**	2,469,684	7,780,130	**1.50**	1,663,119	940,361	20.09%
NUG	**0.34**	**6,039,095**	7,680	**17,762,045**	**3.41**	1,677	0	8.32%

Table 1: CUDA Profiler performance results for TRISH, Podlozhnyuk's (POD), and Nugteren's (NUG)

We found that our TRISH method was from 1.05× faster to 4.4× times faster than Podlozhnyuk's method for black & white images, depending on the image and image size. Our TRISH method was 2-3× faster than Nugteren's method for the images and image sizes tested. The RGB results underscore the potential danger of a data dependent method.

4.2 Degraded Self-Comparison

We ran a short series of tests to determine the impact of ILP and TLP on the overall performance of TRISH. First, we turned off accumulation completely, which makes the results incorrect, but gives an upper bound on performance if we could reduce the frequency of accumulation. Performance increased about ~20% (from 40.9 to 49.4 GB/s, a difference of 8.6). Next, we degraded ILP by reducing the amount of work per thread (k). We measured 18.9 and 40.9 GB/s for $k = 1$ and $k = 31$ respectively. To do this comparison correctly, we must ignore the cost of accumulation. Adding in our accumulation difference of 8.6 from step 1, we get 27.5 and 49.5 GB/s respectively, so our use of ILP makes our code run roughly two times as fast (49.5/27.5). Finally, we degraded TLP by reducing occupancy from 3 to 2 to 1 by padding shared memory. We measured 40.9, 28.5, and 14.6 GB/s for 3, 2, or 1 concurrent blocks per SM, respectively. So we are getting slightly under a three times speedup (40.9/14.6) in performance by using TLP.

4.3 Profiler Comparison

We also ran all three methods under the Nvidia Compute Visual Profiler (CUDA 4.0) on a GTX 580 card with $n = 67{,}108{,}864$ on uniform random data to gather performance metrics. The results are summarized below (see Table 1).

All 3 histogram methods are compute bound with an *instruction:byte* ratio (Ins/Byte) above the 4.00:1 "balanced" instruction:byte ratio recommended for a GTX 580 [5,6].

We notice right away that Podlozhnyuk's method excels at TLP. With low register and shared memory usage and 192 threads per block, it achieves full occupancy of 48 warps ($100\% = 1536/1536$) threads per SM. This high occupancy allows the hardware to keep the instruction pipelines busy retiring 1.58 instructions per cycle (IPC) on average (compared to a hardware maximum of 2.0). However, on the negative side the code issues many more instructions, has many unavoidable bank conflicts, and is involved in a lot of divergent branching, all of which slow down overall performance. These performance issues are all directly a result of the intra-warp thread collisions which are resolved using atomics.

In contrast, Nugteren's method is straightforward code issuing only half as many total instructions as Podlozhnyuk's method. On the negative side, Nugteren's method does a poor job keeping the hardware pipelines busy with only 0.34 instructions retired per cycle (IPC). This is primarily due to poor TLP caused by low occupancy ($6.25\% = 96/1536$) but also because of a small number of bank conflicts. This method doesn't do anything special to take advantage of ILP.

The TRISH method also has poor TLP due to low occupancy ($12.5\% = 192/1536$), which is double that of Nugteren's method. But by taking advantage of ILP via loop unrolling and batching, it manages to retire 1.18 instructions per cycle (IPC). This result is not quite as good as Podlozhnyuk's method but ~3.5 times better than Nugteren's. Despite the extra overhead of having to accumulate regularly, the total instructions issued (II) is competitive with Nugteren's method. This is due to streamlining code and using software VP to reduce the number of arithmetic operations required during binning and accumulation. The TRISH method runs efficiently with no bank conflicts or divergent branching requiring instruction replays.

Total Cycles: So why is the TRISH method faster overall despite being in second place for both for instructions per cycle (IPC) and Instructions Issued (II)? This is because the *Total Cycles* required for the method to complete is lower.

The TRISH method beats the Nugteren method by using some TLP and a lot of ILP to increase the hardware pipeline utilization for more efficient IPC, allowing it to retire 3.5x more instructions per cycle. The TRISH method beats Podlozhnyuk's method by simply issuing many fewer instructions (II). This is because there are no thread collisions causing instruction replays and we reduced overall computations via VP tricks.

5. CONCLUSION

The 256-bin GPU histogram TRISH method presented here modestly outperforms previous GPU histogram methods. Previous methods focused on minimizing I/O via coalesced memory accesses, storing bin counts in shared memory, and simple straight forward binning code. Our method improves compute performance by reducing the Total Cycles required, by improving pipelining and reducing instructions issued via a combination of TLP, ILP, and VP.

The final best results of our GPU TRISH method are up to 3.6× times faster than Cedric's Nugteren's 'per thread' method and up to 1.5× times faster than Victor Podlozhnyuk's 'per warp' method for random data. And up to 4× and 3× faster than Podlozhnyuk's

and Nugteren's methods respectively for image data. More importantly, like Nugteren's method, our performance is data independent.

Our 256-bin histogram GPU TRISH method achieves a throughput of 41.87 billion bytes binned per second (GB/s) in the best case.

5.1 Future Directions

Our method falls well short of peak I/O throughput (~21%), so clever algorithmic improvements, better I/O access patterns, or leveraging of other GPU hardware features may lead to yet more performance gains.

Generalizing: Our method is hardcoded for 256-bin histograms on 8-bit data. It should be possible to generalize it for smaller numbers of bins [1..254] and to support different data types (for instance floats) at an additional arithmetic cost required to map values into bins. However, because of shared memory pressure it will be difficult to generalize the TRISH method to support larger numbers of bins (> 256).

Improving IPC: Perhaps a different version of the binning code could improve ILP even further achieving a higher IPC, perhaps equaling or exceeding Podlozhnyuk's IPC of 1.58. If so, we could see overall performance increase by up to 33% or higher.

Leveraging FPU: Each SM contains 16 FPU pipelines. This hardware is completely unused with the current TRISH method. It may be possible to use the FPU hardware for additional work during binning and accumulation. For example binning 4 additional counts using floats (FPU) per 16 counts binned using integers (ALU) and VP.

6. ACKNOWLEDGMENTS

We thank the reviewers for their helpful comments on presentation and suggestions for additional experiments. One in particular helped us find a copy/paste error that artificially restricted throughput of Podlozhnyuk's method in our initial analysis.

Our thanks to Nvidia, Victor Podlozhnyuk and Cedric Nugteren for providing their histogram source code online. In that same spirit, we provide a link to our source code below.

TRISH source code: https://github.com/shawndb/demoTRISH

7. REFERENCES

[1] Harris, M., *Optimizing Parallel Reduction in CUDA*, URL: http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf

[2] Hennessey, J. and Patterson, D., 2012, *Computer Architecture: A Quantitative Approach*, 5th Edition, Elsevier Inc, Waltham, MA

[3] Merrill, D. and Grimshaw, A., 2010, *Parallel Scan for Stream Architectures.*, Technical Report, CS2010-02, University of Virginia, Dept. of Computer Science, Charlottesville, VA

[4] Merrill, D. and Grimshaw, A., 2010, *Revisiting Sorting for GPGPU Stream Architectures.*, Technical Report CS2010-03, University of Virginia, Dept. of Computer Science, Charlottesville, VA

[5] Micikevicius, P., 2010, *Analysis-Driven Optimization*, Nvidia GPU Technology Conference 2010, URL: http://www.nvidia.com/content/GTC-2010/pdfs/2012_GTC2010v2.pdf

[6] Micikevicius, P., 2010, *Fundamental Optimizations*, Nvidia GPU Technology Conference 2010, URL: http://www.nvidia.com/content/GTC-2010/pdfs/2011_GTC2010.pdf

[7] Nugteren, C., van den Braak, G.J., Corporaal, H., and Mesman, B., 2011. 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* (GPGPU-4). ACM, New York, NY, USA, , Article 1 , 8 pages. DOI=10.1145/1964179.1964181 http://doi.acm.org/10.1145/1964179.1964181

[8] Pearson, K., 1895, Contributions to the Mathematical Theory of Evolution II. Skew Variation in Homogenous Material, *Philosophical Transactions of the Royal Society A: Mathematical, Physical, and Engineering Sciences* v. 186 pp. 343-326

[9] Podlozhnyuk, V., 2007, *Histogram calculation in CUDA*, Technical Report, NVIDIA, URL: http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/histogram256/doc/histogram.pdf

[10] Shams, R. and Kennedy, R., 2007, Efficient Histogram Algorithms for NVIDIA CUDA compatible devices, in *ICSPCS: Proceedings of the International Conference on Signal Processing and Communication Systems*

[11] Volkov, V., 2010, Better Performance at Lower Occupancy, *Nvidia GPU Technology Conference 2010*, URL: http://www.nvidia.com/content/GTC-2010/pdfs/2238_GTC2010.pdf

[12] Yang, Z., Zhu, Y., and Pu, Y., 2008, Parallel Image Processing Based on CUDA, In *Computer Science and Software Engineering, 2008 International Conference on*, v3, pp. 198-201