# Histogram calculation in CUDA

Victor Podlozhnyuk
vpodlozhnyuk@nvidia.com

November 2007

# Document Change History

| Version | Date | Responsible | Reason for Change |
| --- | --- | --- | --- |
| 1.0 | 06/15/2007 | vpodlozhnyuk | First draft of histogram256 whitepaper |
| 1.1.0 | 11/06/2007 | vpodlozhnyuk | Merge histogram256 & histogram64 whitepapers |
| 1.1.1 | 11/09/2007 | Ignacio Castano | Edit and proofread |
|  |  |  |  |

# Abstract

Histograms are a commonly used analysis tool in image processing and data mining applications. They show the frequency of occurrence of each data element.

Although trivial to compute on the CPU, histograms are traditionally quite difficult to compute efficiently on the GPU. Previously proposed methods include using the occlusion query mechanism (which requires a rendering pass for each histogram bucket), or sorting the pixels of the image and then searching for the start of each bucket, both of which are quite expensive.

We can use CUDA and the shared memory to efficiently produce histograms, which can then either be read back to the host or kept on the GPU for later use. The two `CUDA SDK` samples: `histogram64` and `histogram256` demonstrate different approaches to efficient histogram computation on GPU using CUDA.
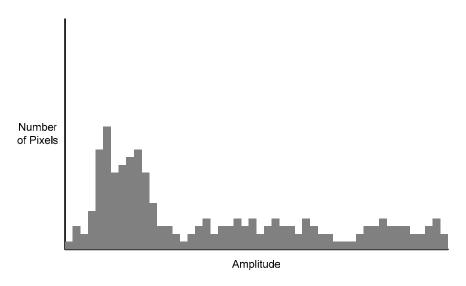
# Introduction



**Figure 1:** An example of an image histogram

An image histogram shows the distribution of pixel intensities within an image. Figure 1 is an example of an image histogram with amplitude (or color) on the horizontal axis and pixel count on the vertical axis.

`Histogram64` demonstrates a simple and high-performance implementation of a 64-bin histogram. Due to the current hardware resource limitations, its approach cannot be scaled to higher resolutions. 64-bin are enough for many applications, but it's not well suited for many image processing applications, like for example histogram equalization.

`Histogram256` demonstrates an efficient implementation of a 256-bin histogram, which makes it suitable for image processing applications that require higher precision than 64 bins can provide.

# Overview

Calculating an image histogram on a sequential device with single thread of execution is fairly easy:

```
for(int i = 0; i < BIN_COUNT; i++)
    result[i] = 0;

for(int i = 0; i < dataN; i++)
    result[data[i]]++;
```

**Listing 1.** Histogram calculation on a single-threaded device. (pseudocode)

Distribution of the computation process between multiple execution threads is possible. It amounts to:

1) Subdivision of the input data array between execution threads

2) Processing of the sub-arrays by each dedicated execution thread and storing the result into a certain number of sub histograms. In some cases it may also be possible to reduce the number of histograms by using atomic operations, but resolving collisions between threads may turn out to be more expensive.

3) Finally all the sub-histograms need to be merged into a single histogram.

When adapting this algorithm to the GPU several constraints should be kept in mind:

Access to the `data[]` array is sequential, but access to `result[]` array is data-dependent (random). Due to inherent performance difference between shared and device memory, especially on random patterns, shared memory is the most optimal storage for the `result[]` array.

On G8x hardware, the total size of the shared memory variables is limited by 16KB.

A single thread block should contain 128-256 threads for efficient execution.

G8x hardware does not have native support for atomic shared memory operations.

An immediate deduction from point 3 is to follow "one scalar thread per sub-histogram" tactic, implemented in `histogram64` CUDA SDK sample. It has obvious limitations: 16 KB per average 192 threads per block amount to max. 85 bytes per thread. So at a maximum, *per-thread* sub-histograms with up to 64 single-byte bin counters can fit into shared memory with this approach. Byte counters also introduce 255-byte limit to the data size processed by single execution thread, which must be taken into account during data subdivision between the execution threads.

However, since the hardware executes threads in SIMD-groups, called *warps (32 threads on G80)*, we can take advantage of this important property for manual (software) implementation of atomic shared memory operations. With this approach, implemented in `histogram256 CUDA SDK` sample, we store *per-warp* sub-histograms, greatly relieving shared memory size pressure: 6 warps (192 threads) * 256 counters * 4 bytes per counter == 6KB

Implementation details as well as benefits and disadvantages of these two approaches are described in the following sections.

# Implementation of histogram64

The per-block sub-histogram is stored in shared memory in the `s_Hist[]` array. `s_Hist[]`is a 2D byte array with `BIN_COUNT` rows and `THREAD_N` columns as shown in Figure 1. Although it is stored in fast on-chip shared memory, a bank-conflict-free access pattern needs to be ensured for best performance, if possible.
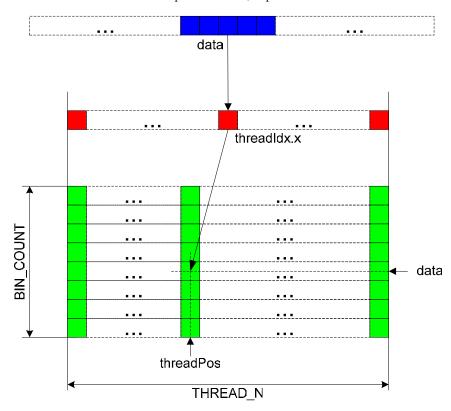


**Figure 1.** `s_Hist[]` array layout for histogram64.

For each thread with its own `threadPos` and data value (which may be the same for some other threads in the thread block), the shared memory bank number is equal to (`threadPos + data * THREAD_N) / 4) % 16`. (See section 5.1.2.4 of the Programming Guide.)

If `THREAD_N` is a multiple of 64, the expression reduces to (`threadPos / 4) % 16`, which is independent of `data` value. (`threadPos / 4) % 16` is equal to the [5 : 2] bits of `threadPos`. A half-warp can be defined as a group of threads in which all threads have the same upper bits [31 : 4] of `threadIdx.x`, but any combination of bits [3 : 0].

If we just set `threadPos` equal to `threadIdx.x`, all thread within a half-warp will access its own byte "lane", but these lanes will map to only 4 banks, thus introducing 4-way bank

conflicts. However, shuffling the [5 : 4] and [3 : 0] bit ranges of `threadIdx.x` will cause all threads within each warp to access the same byte within 4-byte words, stored in 16 different banks, thus completely avoiding bank conflicts.

Since G8x can efficiently work with arrays of only 4, 8 and 16 bytes per element, input data is loaded as four-byte words. For the reasons mentioned above, the data size processed by each thread is limited to 255 bytes or 63 4-byte words, and the data size processed by the entire thread block is limited to `THREAD_N * 63` words. (48,384 bytes for 192 threads)
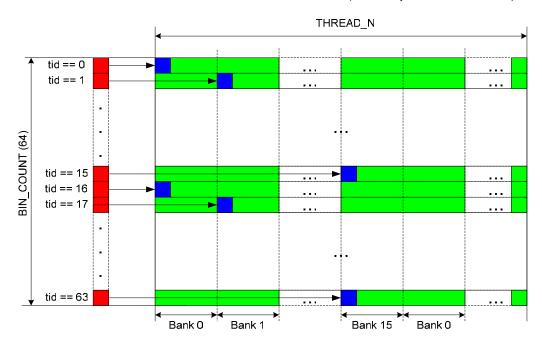


**Figure 2.** Shifting start accumulation positions (blue) in order to avoid bank conflicts during the merging stage in histogram64.

The last phase of computations in `histogram64Kernel()` function is the merging of per-thread sub-histograms into a per-block sub-histogram. At this stage each thread is responsible for its own data value (dedicated `s_Hist[]` row), running through `THREAD_N` columns of `s_Hist[]`. Similarly to the above, the shared memory bank index is equal to `((accumPos + threadIdx.x * THREAD_N) / 4) % 16`. If `THREAD_N` is a multiple of 64, the expression reduces to `(accumPos / 4) % 16`. If each thread within a half-warp starts accumulation at the same position [0 .. `THREAD_N`), then we get 16-way bank conflicts. However, simply by shifting the thread accumulation start position by `4 * (threadIdx.x % 16)` bytes relative to the half-warp base, we can completely avoid bank conflicts at this stage as well. This is shown in Figure 2.

```
const int value = threadIdx.x;

#if ATOMICS
    atomicAdd(d_Result + value, sum);
#else
    d_Result[blockIdx.x * BIN_COUNT + value] = sum;
#endif
```

**Listing 2.** Writing block sub-histogram into global memory.

If atomic global memory operations are available (exposed in CUDA via `atomic*()` functions) concurrent threads (within the same block, or within different blocks) can update the same global memory locations atomically, so thread blocks can merge their results within a single CUDA kernel. Otherwise, each block must output its own sub-histogram, and a separate final merging kernel `mergeHistogram64Kernel()` must be applied.

# Implementation of histogram256

The per-block sub-histogram is stored in shared memory in the `s_Hist[]` array. `s_Hist[]`is a 2D word array of `WARP_N` rows per `BIN_COUNT` columns, where each warp of a thread block is responsible for its own sub-histogram, as shown in Figure 3.
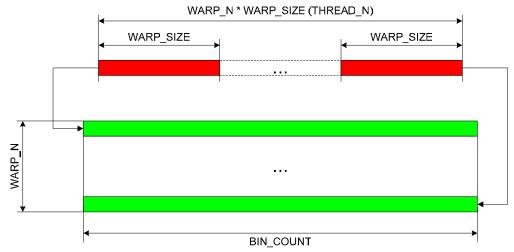


**Figure 3.** `s_Hist[]` layout for histogram256.

Compared to `histogram64,` threads no longer have isolated sub-histograms, but each group of 32 threads (warp) shares the same memory range, thus introducing intra-warp shared memory collisions. Since atomic shared memory operations are not natively supported on G8x, special care has to be taken in order to resolve these collisions and produce correct results.

The core of the 256-bin histogram implementation is the `addData256()` device function that's shown in Listing 3. Let's describe its logic in detail.

```
__device__ void addData256(
    volatile unsigned int *s_WarpHist,
    unsigned int data,
    unsigned int threadTag
){
    unsigned int count;

    do{
        count = s_WarpHist[data] & 0x07FFFFFFU;
        count = threadTag | (count + 1);
        s_WarpHist[data] = count;
    }while(s_WarpHist[data] != count);
}
```

**Listing 3. Avoiding intra-warp shared memory collisions.**

The data argument is a value that was read from global memory, and that lies in the [0, 255] range. Each warp thread must increment a location in the s_WarpHist[ ] array that depends on the input data. s_WarpHist[ ] is a section of the entire block sub-histogram s_Hist[ ], that corresponds to the current warp.

In order to prevent collisions between threads of a warp, the histogram counters are tagged according to the last thread that wrote to them. The tag is stored in he 5 most significant bits of the histogram counters. Only 5 bits are required, because the number of threads in a warp is 32 ($2^5$).

The first thing each thread does is to read the previous value of the histogram counter. The most significant bits of the count are masked and replaced with the tag of the current thread. Then each thread writes the incremented count back to the sub-histogram in shared memory.

When each thread in the warp receives unique data values, there are no collisions at all, and no additional actions need to be done. However, when two or more threads collide trying to write to the same location, the hardware performs shared memory *write combining,* that results in the acceptance of the tagged counter from one of the threads, and the rejection from all the other pending threads.

After the write attempt, each thread reads from the same shared memory location. The threads that were able to write their count, exit the loop and stay idle waiting for the remaining threads in the warp. The warp continues its execution as soon as all the threads exit the loop.

Since each warp uses its own sub-histogram and warp threads are always synchronized we do not rely on warp scheduling order (which is undefined). The loop won't be repeated more than 32 iterations, and that will only happen in case all the threads try to write the to the same location.

```
for(int pos = threadIdx.x; pos < BIN_COUNT; pos += blockDim.x){
    unsigned int sum = 0;

    for(int base = 0; base < BLOCK_MEMORY; base += BIN_COUNT)
        sum += s_Hist[base + pos] & 0x07FFFFFFU;

#if ATOMICS
        atomicAdd(d_Result + pos, sum);
#else
        d_Result[blockIdx.x * BIN_COUNT + pos] = sum;
#endif
}
```

**Listing 4.** Writing block sub-histogram into global memory.

The last phase of computations in `histogram256Kernel()` is the merging of per-warp sub-histograms into a per-block one. Similarly to `histogram64Kernel()`, the per-block histogram is written to global memory. If atomic global memory operations are available (exposed in CUDA via `atomic*()` functions) concurrent threads (within the same block, or within different blocks) can update the same global memory locations atomically, so thread blocks can merge their results within a single CUDA kernel. Otherwise, each block must output its own sub-histogram, and a separate final merging kernel `mergeHistogram256Kernel()` must be applied.

# Performance

Since `histogram64` is 100% free from bank conflicts and intra-warp branching divergence, it runs at extremely high data-independent performance rate, which reaches 10GB/s on G80.

On the other side, the performance of `histogram256` depends on the input data, and that causes bank conflicts and intra-warp branching divergence. When using a random distribution of input values, `histogram256` runs at 5.5GB/s on G80.

# Bibliography

1.  Wolfram Mathworld. "Histogram" http://mathworld.wolfram.com/Histogram.html

**Notice**

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent or patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. NVIDIA Corporation products are not authorized for use as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

**Trademarks**

NVIDIA, the NVIDIA logo, GeForce, NVIDIA Quadro, and NVIDIA CUDA are trademarks or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

**Copyright**