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	Initial release
1.1.0	11/06/2007	vpodlozhnyuk	Merge histogram64 & histogram256 documents
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.

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com

Introduction

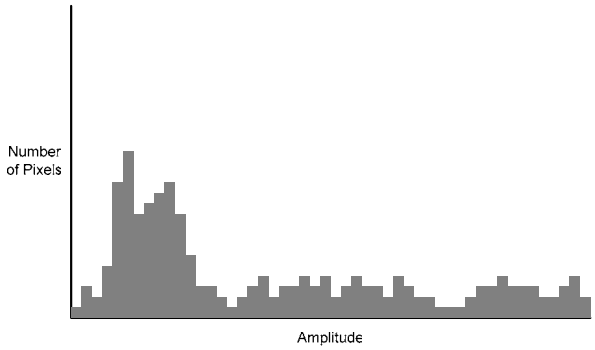

Figure 1: An example of an image histogram

An image histogram shows the distribution of pixel intensities within an image. Figure 1 is an example of a 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.

Motivation

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 predicatable, 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 subhistogram" 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* subhistograms 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* subhistograms, greatly relieving shared memory size pressure: 6 warps (192 threads) * 256 byte * 4 bytes per counter == 6KB

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

Implementation of histogram64

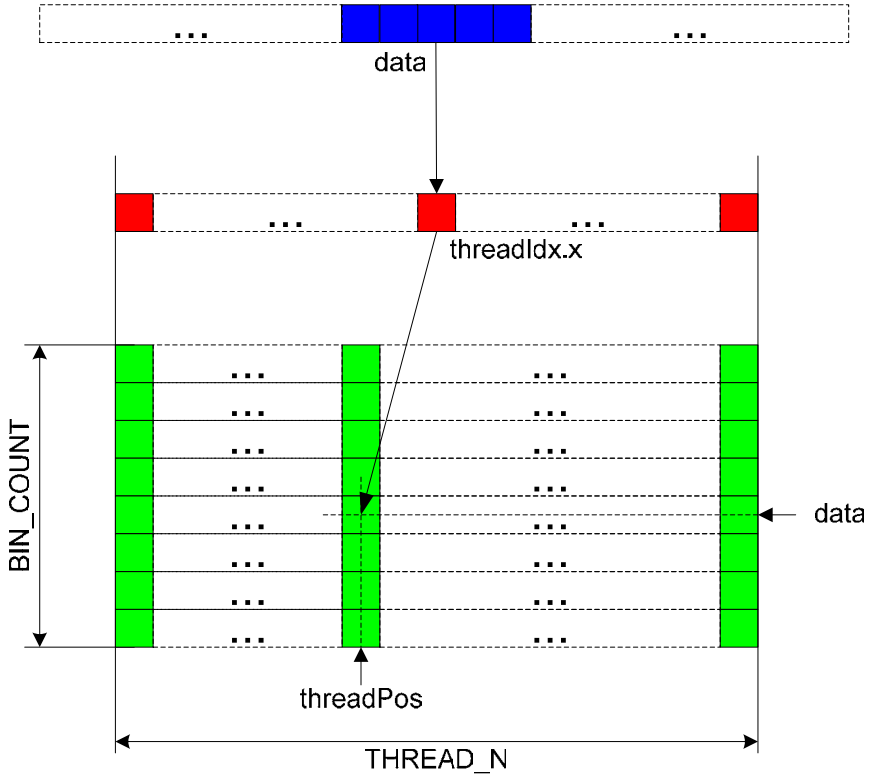

Figure 1. s_Hist[] array layout for histogram64.

s_Hist[] (per-block subhistogram storage) 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.

For each thread with its own `threadPos` and pixel color value (which may be the same for some other threads in the thread block), the shared memory bank number is equal to `(threadPos + color * 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 color value. `(threadPos / 4) % 16` is equal to bits [5 : 2] 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 double words, stored in 16 different banks, thus completely avoiding bank conflicts.

Since G80 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 63 double words, and the data size processed by the entire thread block is limited to `THREAD_N * 63` double words. (48,384 bytes for 192 threads)

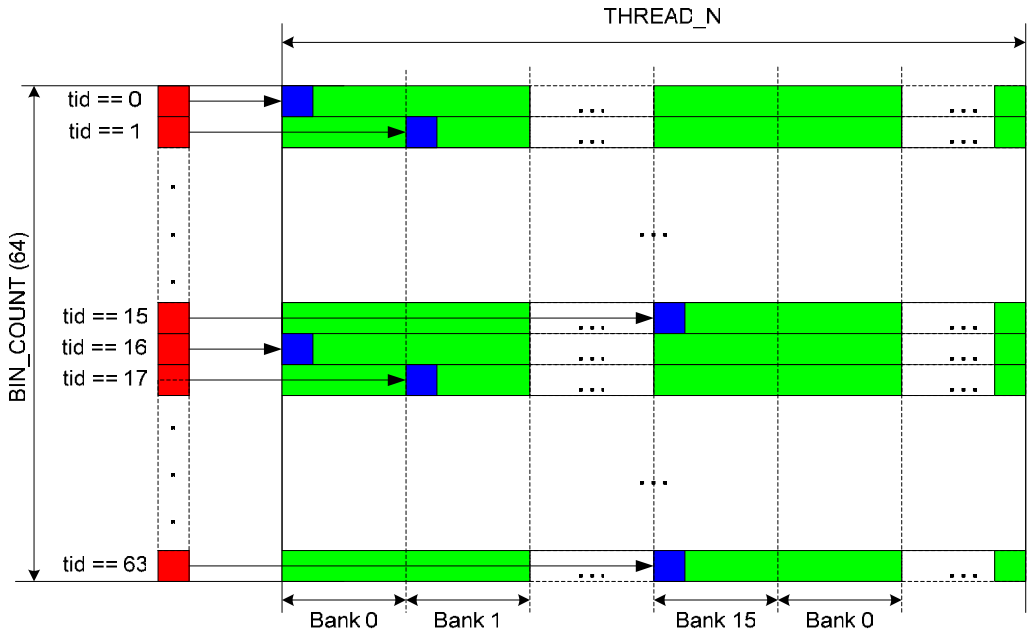

Figure 2. Shifting start accumulation positions (blue) in order to avoid bank conflicts during the reduction stage in histogram64.

The last phase of computations in `histogram64Kernel()` function is the reduction of per-thread subhistograms into a per-block subhistogram. At this stage each thread is responsible for its own pixel 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-1`, 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 demonstrated in Figure 2.

```
const int value = threadIdx.x;

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

Listing 2. Writing block subhistogram into global memory.

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 subhistogram, and a separate final reduction kernel `reduceHistogram64Kernel()` must be applied.

Implementation of histogram256

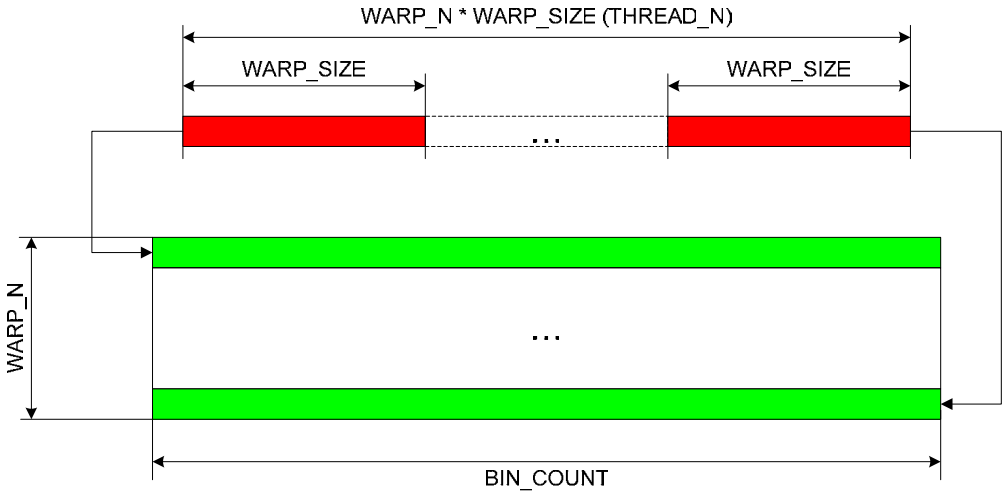

Figure 3. s_Hist[] layout for histogram256.

`s_Hist[]` (per-block sub histogram storage) is 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, processing dedicated row. 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.

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

    do{
        count = s_WarpHist[data] & 0x07FFFFFU;
        count = threadTag | (count + 1);
        s_WarpHist[data] = count;
    }while(s_WarpHist[data] != count);
}
```

Listing 3. Avoiding intra-warp shared memory collisions.

`addData256()` is the core of the 256-bin histogram implementation**.** Let's describe its logic in detail.

According to `data` value (lying within 0 .. 255 range), read from global memory, each warp thread must increment corresponding value in the `s_WarpHistp[]` array -- a "frame"(row) within `s_Hist[]` array, corresponding to current warp.

Each warp thread reads current warp counter `s_WarpHist[data]`, corresponding to `data` value, then locally increments, *tags* it by warp-local thread ID (equal to `threadIdx.x % 32`), and writes it back to the same `s_WarpHist[data]` position. In case each warp thread received unique `data` values (from global memory), there are no collisions at all and no additional actions need to be done. Otherwise, when two or more threads collide on the same bin counter, the hardware performs shared memory *write combining,* resulting in acceptance of the tagged counter from one thread and rejection from all other pending threads. After the write attempt each thread queries the shared memory count value (the same `s_WarpHist[data]`) and owing to the tag decides whether its pending increment made its way to shared memory. If true, it becomes idle (masked out by hardware) until the entire warp is done (all the collisions are resolved). Otherwise, some other thread has submitted its increment into `s_WarpHist[]`, and current thread needs to grab the new counter value and perform the same actions. Since each warp is isolated and warp threads are always synchronized we do not rely on warp scheduling order (which is undefined). Not more than after 32 loop iterations all the warp threads submit their increments into `s_WarpHist[]`.

```
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] & 0x07FFFFFU;

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

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

The last phase of computations in `histogram256Kernel()` is the reduction 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 reduction kernel `reduceHistogram256Kernel()` 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