Histogram calculation in OpenCL

Victor Podlozhnyuk
vpodlozhnyuk@nvidia.com

April 2009

Document Change History

Version	Date	Responsible	Reason for Change
1.0	06/15/2007	Victor Podlozhnyuk	First draft of histogram256 whitepaper
1.1.0	11/06/2007	Victor Podlozhnyuk	Merge histogram256 & histogram64 whitepapers
1.1.1	11/09/2007	Ignacio Castano	Edit and proofread
2.3	04/13/2009	Victor Podlozhnyuk	Adapted to OpenCL implementation

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 OpenCL and the fast local 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 `OpenCL SDK` samples: oclH`istogram64` and oclH`istogram256` demonstrate different approaches to efficient histogram computation on GPU using OpenCL..

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

Introduction

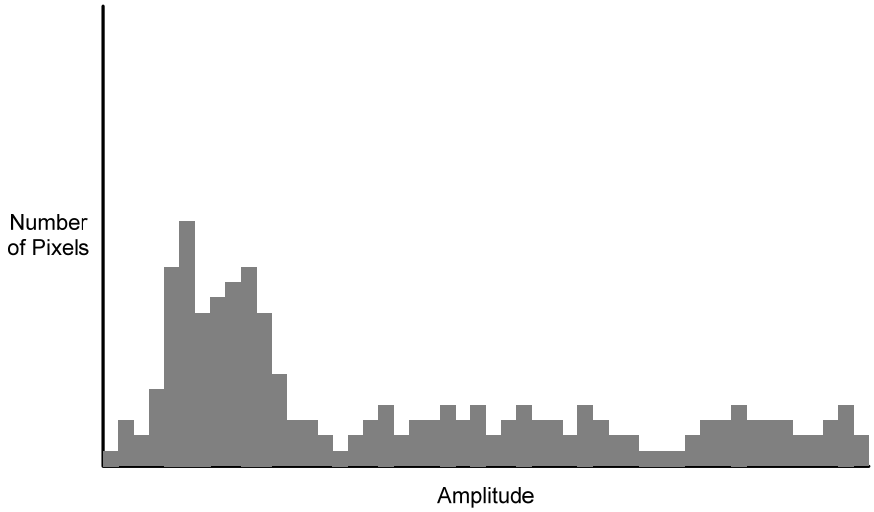

Figure 1: An example of an image histogram

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

`oclHistogram64` 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 resolution is enough for many applications, but it's not well suited for many image processing methods, like, for example, histogram equalization.

`oclHistogram256` 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(unsigned int i = 0; i < BIN_COUNT; i++)
    result[i] = 0;

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

Listing 1. Histogram calculation on a single-threaded device. (pseudo-code)

Distribution of the computation process between multiple work-items is possible. It amounts to three fundamental computation steps:

1) Map input data array to work-groups and work-items within a work-group. Generally, the exact mapping pattern doesn't matter for correctness. The only mandatory constraint is that each input data element must be counted exactly once.

2) Each work-item sequentially processes input data elements it is mapped, building up private work-item sub-histograms., which is largely identical to the pseudo-code in Listing 1. It may also be possible for subgroups of work-items within a work-group to build up common sub-histograms shared by these subgroups by using atomic operations (or otherwise resolve potential access collisions inevitable in parallel processing), thus considerably decreasing the size of required per work-group sub-histogram storage and the amount of work for step 3. But resolving collisions between work-items within the subgroups may vary from being expensive to impossible.

3) Finally all sub-histograms need to be merged into a single histogram. Each bin of the resulting histogram is merely a sum of corresponding bin values of sub-histogram(s) produced on previous stages of computation. If needed, this step is performed at multiple levels, i.e. first-level merge step would combine work-item sub-histograms to form a work-group sub-histogram; second-level merge step would combine work-group sub-histograms to form the histogram of the entire input data array.

When adapting these steps to the particular family of GPUs some important features and characteristics should be kept in mind.

- Local memory has approximately an order of magnitude higher bandwidth than global memory (loading/storing from/to local memory is generally as fast as reading/writing private register memory), tolerates many irregular access patterns, but is limited in size: maximum local storage size for G8x / G9x / G10x NVIDIA GPUs is 16KB.
- Only G10x NVIDIA GPUs provide built-in support for workgroup-wide atomic operations in local memory. But even on earlier G8x / G9x NVIDIA GPUs local-memory atomic operations can be emulated basing on the fundamental fact that work-groups are executed as subgroups of logically coherent work-items, called *warps* , though "consistency domain" of such manually-implemented atomic operations will also be limited by warp size, which is 32 work-items on G8x / G9x / G10x NVIDIA GPUs. But even with built-in support of atomic

operations we may want to limit the amount of work-items sharing the same sub-histogram, since the hardware still has to serialize colliding accesses, and the degree of contention increases as the amount of work-items competing for shared resource does.

- Depending on the utilization of local and private register memory, optimal work-group size typically varies in the range of 64..256 work-items.

With the intention to minimize the potential contention degree and avoid the need in expensive atomic operations (either built-in or emulated), we simply stick to "one sub-histogram per work-item" tactics, which is implemented in `oclHistogram64` OpenCL SDK sample. Such strategy however introduces some serious limitations: 16 KB per average 192 work-items in a group amount to the maximum of ~85 bytes of local memory per work-item. So this approach limits the histogram resolution to 64 bins on G8x / G9x / G10x NVIDIA GPUs. From the implementation perspective, byte counters also introduce 255-byte limit to the data size processed by single work-item, which must be taken into account during data subdivision between the execution threads. Also note that 8- and 16-bit loads/stores are not part of OpenCL 1.0 standart and currently available as `cl_khr_byte_addressable_store` extension on G8x / G9x / G10x NVIDIA GPUs.

`oclHistogram256` OpenCL SDK sample raises the histogram resolution limit by utilizing local-memory atomics and building up *per-warp* sub-histograms in local memory, greatly relieving local memory size pressure: 192 work-items per work-group / 32 work-items per warp * 256 counters per sub-histogram * 4 bytes per counter = 6KB per work-group.

Implementation details of these two approaches are described in the following sections.

Implementation of oclHistogram64

The per-work-item sub-histograms are stored in the local-memory `l_Hist[]` array, treated as a 2D byte array of `BIN_COUNT` rows by `WORKGROUP_SIZE` columns, as shown in Figure 1. For best performance a bank-conflict-free access pattern needs to be ensured, if possible.

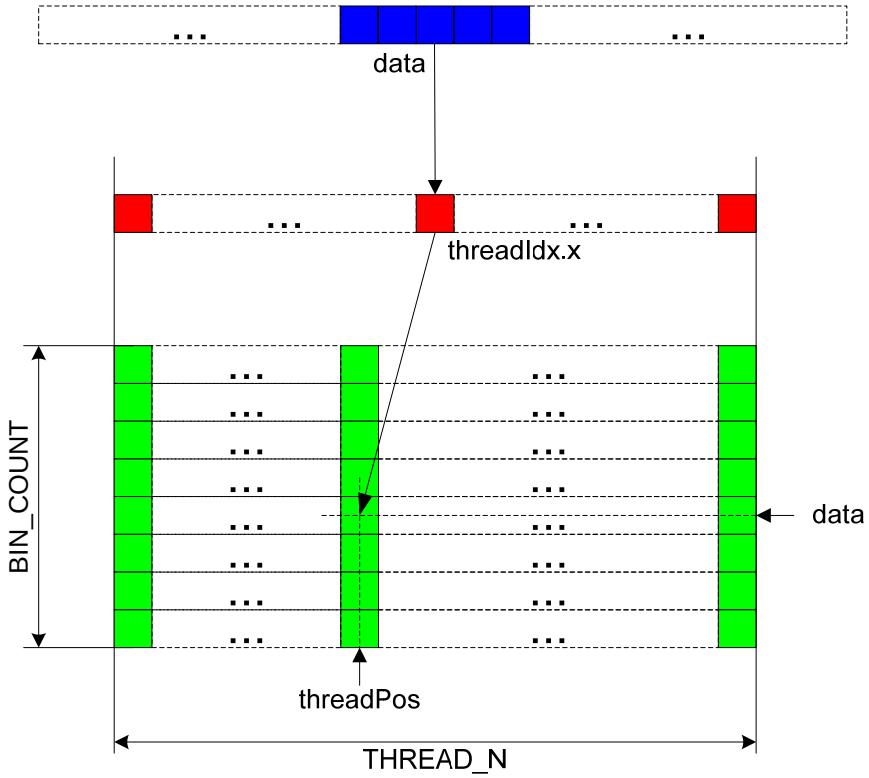

Figure 1. `l_Hist[]` array layout for `oclHistogram64`.

For each work-item of a work-group with its own coordinate `lPos` and data value (which may be the same for some or all other work-items in the work-group or not), local memory bank index is equal to `((lPos + data * WORKGROUP_SIZE) / 4) % 16`. (See section 5.1.2.4 of CUDA Programming Guide.)

If `WORKGROUP_SIZE` is a multiple of 64, the expression reduces to `(lPos / 4) % 16`, which is independent of `data` value. `(lPos / 4) % 16` is equal to the [5: 2] bits of `lPos.` A half-warp can be defined as a group of work-item in which all work-items have the same upper bits [31 : 4] of `get_local_id(0)`

If `lPos` is simply set equal to `get_local_id(0)`, all work-items 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, swapping the [5 : 4] and [3 : 0] bit ranges in the bit representation of `get_local_id(0)` will make bank index identical to lower four bits of get_local_id(0) thus completely eliminating bank conflicts.

Since G8x / G9x / G10x NVIDIA GPUs most efficiently work with global-memory arrays of 4, 8 and 16 bytes per element, input data is loaded from global memory as four-byte words. Since local-memory histogram bin counters are only 8-bit, the data size processed by single work-item is limited to 255 bytes or 63 full 4-byte words, and correspondingly data processed by the entire work-group is limited to `WORKGROUP_SIZE * 63 4-byte` words. (48,384 bytes for 192 work-items per work-group)

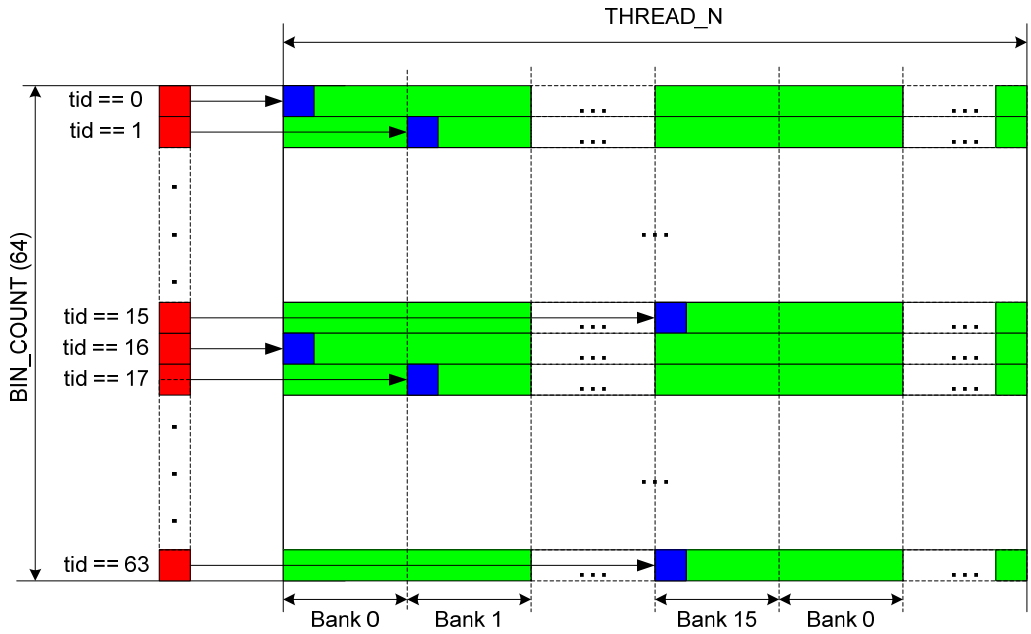

Figure 2. Shifting start accumulation positions (blue) in order to avoid bank conflicts during the merging stage in `histogram64`.

The second computational phase in the `histogram64()` kernel is merging of built per-workitem sub-histograms into a per-workgroup sub-histogram. At this phase each bin counter is assigned corresponding work-item, running through `WORKGROUP_SIZE` columns of `l_Hist[]`. Similarly to the above, the local memory bank index is equal to `((accumPos + get_local_id(0) * WORKGROUP_SIZE) / 4) % 16`. Still assuming `WORKGROUP_SIZE` to be 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 .. WORKGROUP_SIZE)`, then we get 16-way bank conflicts. However, simply by shifting the thread accumulation start position by `4 * (get_local_id(0) % 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.

After the workgroup-level merge step the produced subhistograms are written to global memory and passed down to the dedicated `mergeHistogram64()` kernel finalizing the merging. Although `mergeKernel64()`'s global memory loads are uncoalescable due to large stride of `BIN_COUNT` words between consecutive work-items, its running time nonetheless is only a fraction (< 5%) of the main histogram64() kernel, so may be largely ignored.

Implementation of oclHistogram256

The per-warp sub-histograms are stored in the local-memory `l_Hist[]`array, treated as a 2D word array of `WARP_COUNT` rows by `BIN_COUNT` columns, as shown in Figure 3.

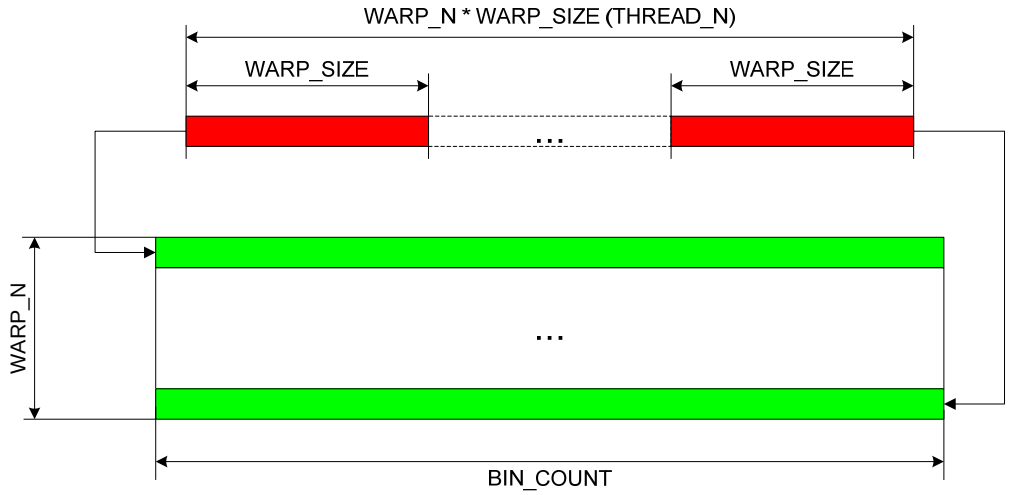

Figure 3. `l_Hist[]` layout for oclHistogram256.

As was already stated in the "Overview" section, since a sub-histogram is shared by more than one work-item, local memory collisions are inevitable. Also since atomic local memory operations are not natively supported on G8x / G9x NVIDIA GPUs, special care has to be taken in order to resolve the collisions and produce correct results.

The heart of the 256-bin histogram implementation is `addData256() function`

```
void addData256(
      volatile __local uint *l_WarpHist,
      uint data,
      uint workitemTag
){
    uint count;

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

Listing 3. Resolving intra-warp local memory collisions.

The `l_WarpHist` is a pointer to current warp sub-histogram.

The `data` argument is a value read from global memory lying in the [0, 255] range. Essentially, at function entry each work-item of a warp has an outstanding increment operation it has to commit. Obviously, the exact serialization patter in the case of collisions doesn't matter for correctness.

`workitemTag` is a unique id used by work-items to "sign" the data on write attempt and then determine whether the outstanding increments have finally committed to local memory. It is simply 5 lower bits of local ID (left-shifted by 27 bits to occupy 5 higher bits of `count`).

Let's consider what happens in a single iteration of the `do{…}while(…);` loop in Listing 3: Each active (e.g. not masked out bwork-item of a warp loads a corresponding value from the sub-histogram storage and produces a private tagged increment (in `count` variable),

then stores it back, all in lock-step (e.g. logically coherently) by the definition of warp. However, in the case when two or more work-items store to the same location, the hardware performs local memory *write combining,* that effectively results in rejection of all but one colliding stores.

The `while(…)` condition evaluated to FALSE by a work-item means that this work-item has committed it's increment and thus should exit the loop. At hardware level committed work-items are *masked out* (e.g. have all state updates disabled) until all warp work-items exit the loop, after which the warp continues its normal execution. Worst-case scenario for `addData256()` is 32 iterations per warp, in the case when entire warp receives the same value in the formal `data` parameter.

The second and the third computational steps of the discussed 256-bin histogram implementation are largely indentical to those of `oclHistogram64`: merging of per-warp sub-histograms into a per-workgroup subhistogram (second phase of `histogram256()` kernel) and separate `mergeHistogram256()` kernel merging the per-workgroup sub-histograms produced by `histogram256()` kernel. Similar to `mergeHistogram64()`, `mergeHistogram256()` has uncoalescable global loads, but its running time constitutes only a small fraction of the total running time.

Bibliography

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