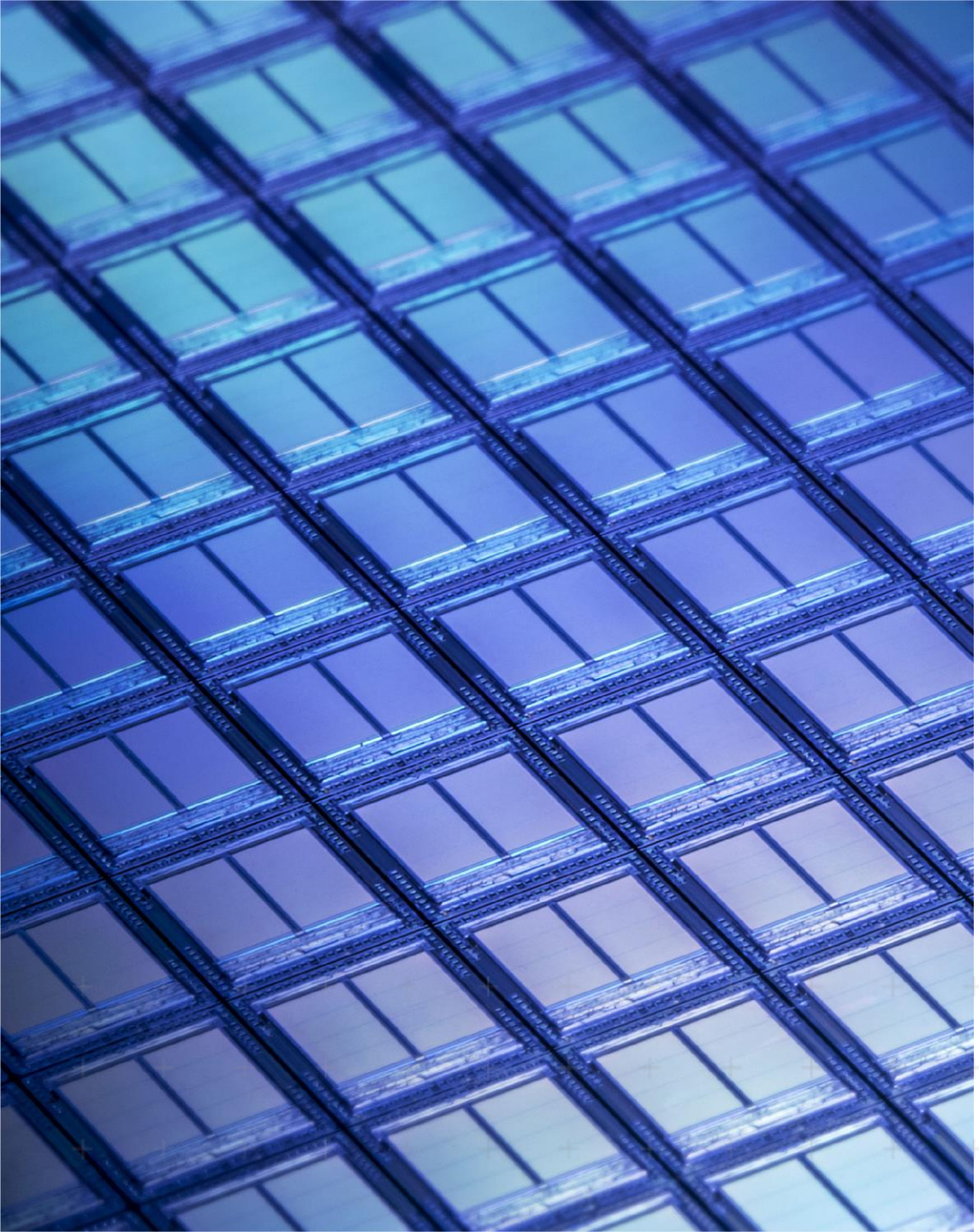


High Performance GPGPU Implementation of a Large 2D Histogram (S9734)

Mark Roulo

Principal Software Engineer

Wed, March 20, 2019

2:00PM

The Problem

1. Create a "large" (2M bins) 2D histogram
2. ~1M input values
3. The histogram data "clusters"
4. We can 'cap' the bins. 1-byte bins are okay.
5. This is a throughput, not a latency problem.

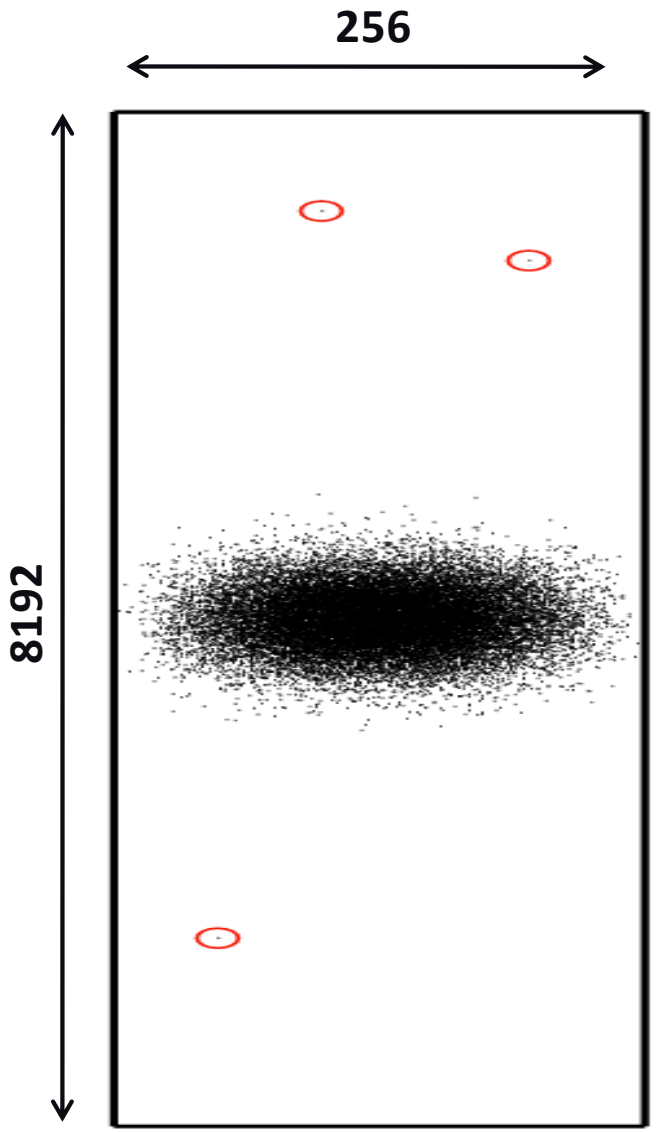
Caution

1. Be careful NOT to run out of cache when benchmarking!
2. Don't forget the "--optimize 2" option when compiling the C!

Hardware

1. 1 Volta
2. 2x20 2.25 GHz Broadwell Cores (E5-2698 v4)

Example Histogram



Basic “C” implementation

For each input …
If the bin is below the cap value
increment the bin

```
void histogram_cpu (input_value_t input[], bin_t histogram[])
{
    for (int i = 0; i < INPUT_VALUE_COUNT; ++i)
    {
        input_value_t myValue = input[i];

        if (histogram[myValue] < BIN_SOFT_CAP)
            ++histogram[myValue];
    }
}
```

Some Themes

This problem is going to require paying attention to:

1. **HBM Bandwidth**
2. **L2 Cache Bandwidth**
3. **L1/Shared Memory Bandwidth**
4. **Problem working set size**

V0 - Basic CUDA implementation

Same basic strategy, but threaded and we have to use atomics
Each block handles one histogram
80 SMs => 80 histograms at a time

```
__global__ void histogram_gpu (input_value_t input__[], int values_per_input, bin_t histogram__[], int bin_count)
{
    // We need a pointer to an unsigned int (rather than to a bin_t/byte so that we can
    // use the atomicAdd below.
    input_value_t *input  = input__ + blockIdx.x * values_per_input;
    bin_t        *my_histo  = histogram__ + blockIdx.x * bin_count;
    unsigned int *histo_int = (unsigned int*)my_histo;

    for (int i = threadIdx.x; i < values_per_input; i += blockDim.x)
    {
        const input_value_t myValue = input[i];

              unsigned int *p = histo_int + (myValue >> 2);      // Pointer to bin as unsigned int (not bye) for atomicAdd
        const unsigned int byteInInt = myValue & 3;              // 0, 1, 2, or 3
        const unsigned int shift     = 8 * byteInInt;            // 0, 8, 16, or 24
        const unsigned int add       = 1 << shift;               // 1, 256, 65536, or ...

        if (my_histo[myValue] < BIN_SOFT_CAP)
            atomicAdd(p, add);
    }
}
```

V0 - Basic CUDA implementation

	Histograms	Time	µs/histogram
1 C Thread	2,000	3.5 sec	1,750
40 C Threads	40x2,000 = 80,000	7.95 sec	99
80x1 CUDA Blocks	80x2,000=160,000	35.97 sec	225

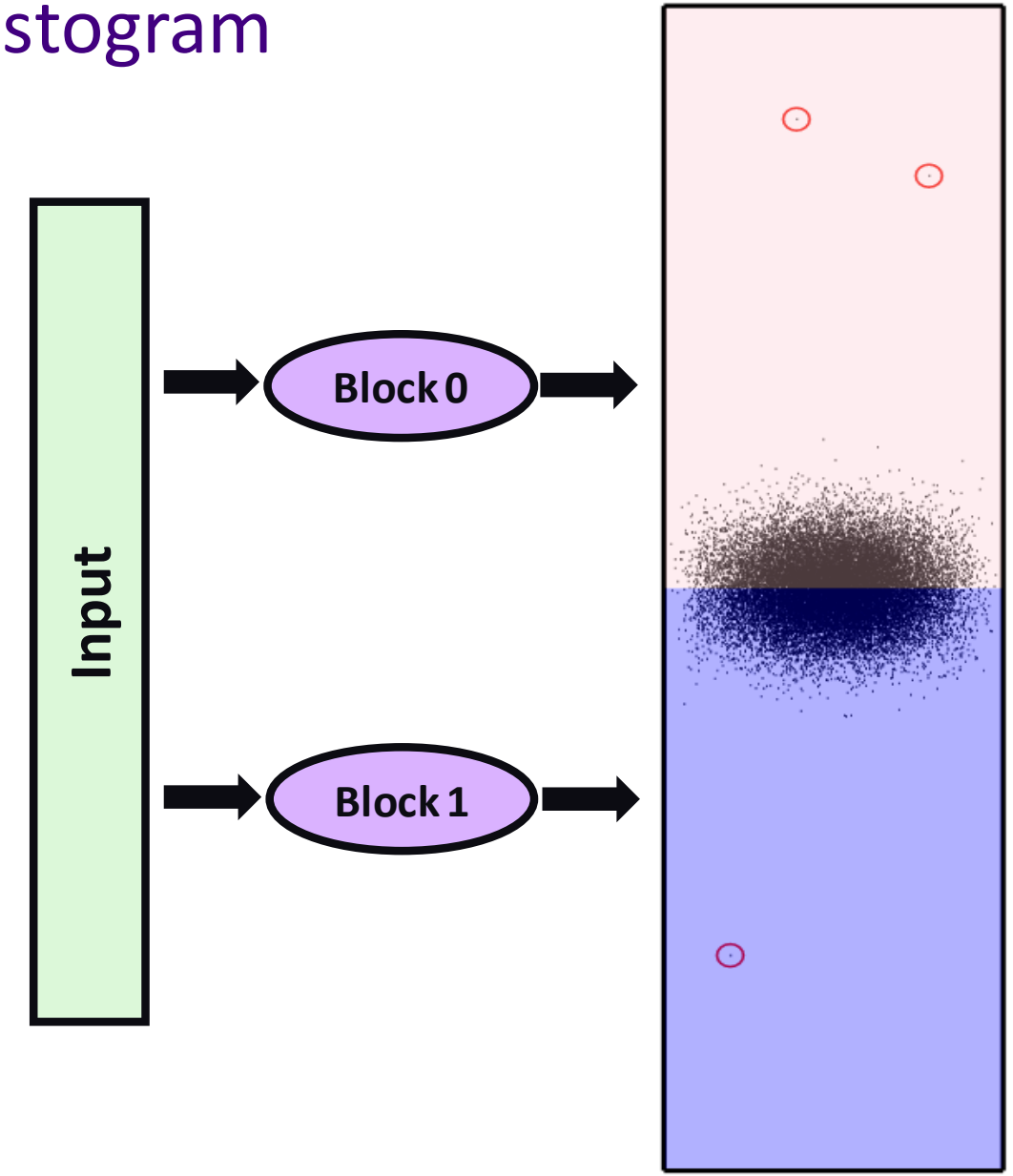
V0 - Basic CUDA implementation

Characteristic	Value	vs initial
Streaming Input B/W	~13 GB/sec	1.0x
Histogram Zero-ing B/W	~6.5 GB/sec	1.0x
Atomic Increments	~3.2×10^9/sec	1.0x
Streaming L2 Read B/W	~13 GB/sec	1.0x
Histogram Working Set Size	~14 MB	1.0x

CUDA V1 – Two Blocks/Histogram

Use two blocks per histogram

Each block sees ALL the input
Each block only writes ½ the histogram.



CUDA V1 – Two Blocks/Histogram

	Histograms	Time	µs/histogram
1 C Thread	2,000	3.5 sec	1,750
40 C Threads	40x2,000 = 80,000	7.95 sec	99
80x1 CUDA Blocks	80x2,000=160,000	35.97 sec	225
40x2 CUDA Blocks	40x2,000=80,000	6.62 sec	83

CUDA V1 – Two Blocks/Histogram

Characteristic	Value	vs initial
Streaming Input B/W	~47 GB/sec	~3.5x
Histogram Zero-ing B/W	~23.5 GB/sec	~3.5x
Atomic Increments	~11.8×10^9/sec	~3.5x
Streaming L2 Read B/W	~94 GB/sec	~7.0x
Histogram Working Set Size	~7 MB	~0.5x

CUDA V2 – 4 Blocks/Histogram

	Histograms	Time	µs/histogram
1 C Thread	2,000	3.5 sec	1,750
40 C Threads	40x2,000 = 80,000	7.95 sec	99
80x1 CUDA Blocks	80x2,000=160,000	35.97 sec	225
40x2 CUDA Blocks	40x2,000=80,000	6.62 sec	83
20x4 CUDA Blocks	20x2,000=40,000	2.12 sec	53

CUDA V2 – 4 Blocks/Histogram

Characteristic	Value	vs initial
Streaming Input B/W	~74 GB/sec	~5.5x
Histogram Zero-ing B/W	~36.9 GB/sec	~5.5x
Atomic Increments	~18.5×10^9/sec	~5.5x
Streaming L2 Read B/W	~300 GB/sec	~23x
Histogram Working Set Size	~3.5 MB	~0.25x

CUDA V3 – 8 Blocks/Histogram

	Histograms	Time	µs/histogram
1 C Thread	2,000	3.5 sec	1,750
40 C Threads	40x2,000 = 80,000	7.95 sec	99
80x1 CUDA Blocks	80x2,000=160,000	35.97 sec	225
40x2 CUDA Blocks	40x2,000=80,000	6.62 sec	83
20x4 CUDA Blocks	20x2,000=40,000	2.12 sec	53
10x8 CUDA Blocks	10x2,000=20,000	1.38 sec	69

CUDA V3 – 8 Blocks/Histogram

Characteristic	Value	vs initial
Streaming Input B/W	~57 GB/sec	~4.4x
Histogram Zero-ing B/W	~28 GB/sec	~4.4x
Atomic Increments	~14.2×10^9/sec	~4.4x
Streaming L2 Read B/W	~450 GB/sec	~34.5x
Histogram Working Set Size	~1.7 MB	~0.125x

Summarize

Working Set size is important. We want to fit into L2.
The L2 Cache has ~3x - ~4x the DRAM bandwidth, so 4x reads of the same data are fine.
At 20 Simultaneous Histograms, our Working Set fits in L2

At 20 Simultaneous Histograms, we may be L2 atomicAdd() limited. Can we address this?

But first …

CUDA V4 – Read 4 values at a time

Read 4 bin values at a time. Process all 4. Repeat.

```
const unsigned long *ipt = (unsigned long*)(input + threadIdx.x * INTS_PER_LONG);
const unsigned long *end = (unsigned long*)(input + values_per_input + blockDim.x * INTS_PER_LONG);

unsigned long bins  = *ipt;
ipt += blockDim.x;
unsigned long bins2 = *ipt;
ipt += blockDim.x;

while (ipt < end)
{
    const input_value_t bin_a = (input_value_t)(bins & 0xFFFFFFFF);
    const input_value_t bin_b = (input_value_t)(bins >> 32);
    const input_value_t bin_c = (input_value_t)(bins2 & 0xFFFFFFFF);
    const input_value_t bin_d = (input_value_t)(bins2 >> 32);
                :
                :
```

CUDA V4 – Read 4 values at a time

	Histograms	Time	µs/histogram
1 C Thread	2,000	3.5 sec	1,750
40 C Threads	40x2,000 = 80,000	7.95 sec	99
80x1 CUDA Blocks	80x2,000=160,000	35.97 sec	225
40x2 CUDA Blocks	40x2,000=80,000	6.62 sec	83
20x4 CUDA Blocks	20x2,000=40,000	2.12 sec	53
20x4 CUDA Blocks 4 reads	20x2,000=40,000	1.38 sec	34

CUDA V4 – Read 4 values at a time

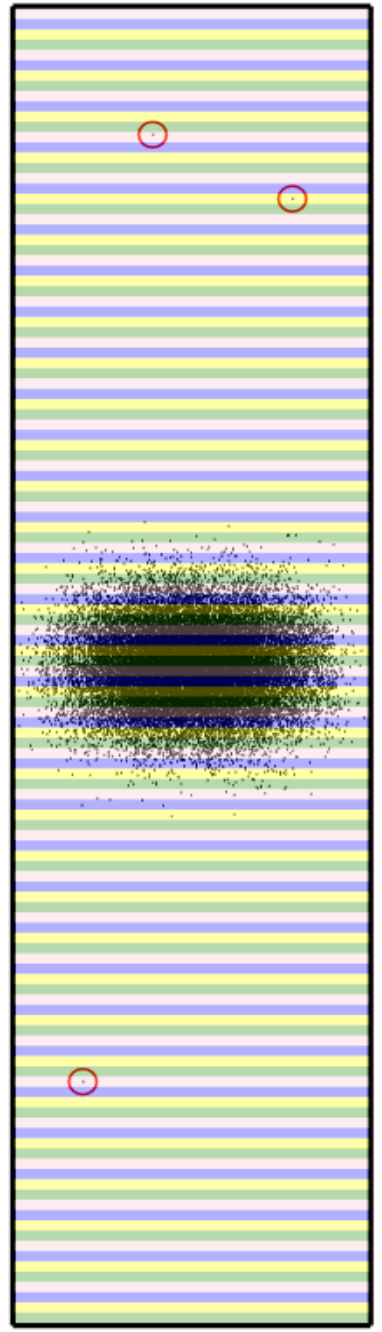
Read 4 bin values at a time. Process all 4. Repeat.

Characteristic	Value	vs initial
Streaming Input B/W	~115 GB/sec	
Histogram Zero-ing B/W	~58 GB/sec	
Atomic Increments	~29.0×10^9/sec	
Streaming L2 Read B/W	~460 GB/sec	
Histogram Working Set Size	~3.5 MB	

CUDA V5 – Interleave

Use shared memory for the most used bins.

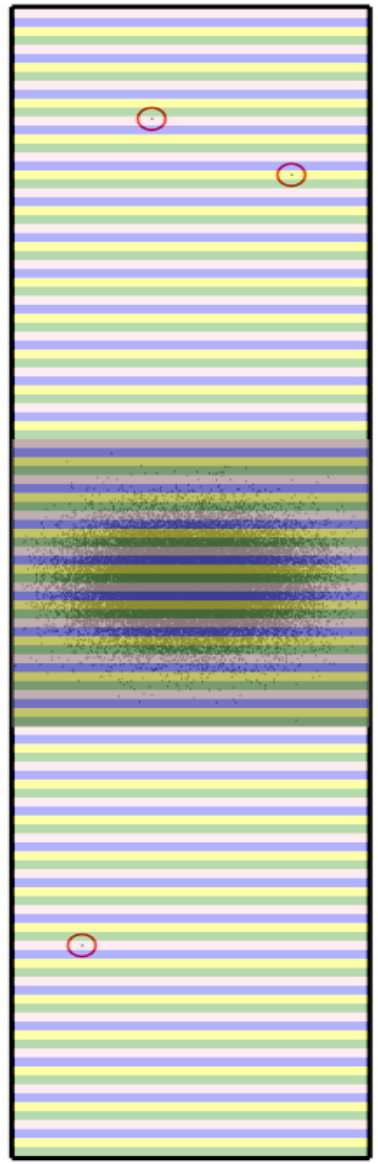
This lets each of the four blocks do about the same amount of work.



CUDA V5 – … And Use shared memory

Use shared memory for the most used bins.

Unlike CPUs, writes to L1 are not cached! Instead, they flow back to L2.



CUDA V5 – Interleave and use shared memory

Use shared memory for the most used bins.

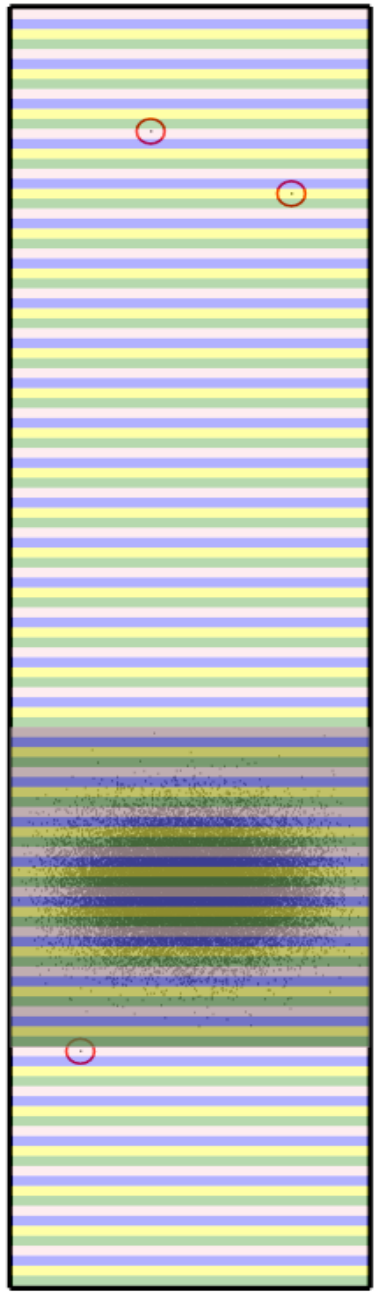
	Histograms	Time	µs/histogram
1 C Thread	2,000	3.5 sec	1,750
40 C Threads	40x2,000 = 80,000	7.95 sec	99
80x1 CUDA Blocks	80x2,000=160,000	35.97 sec	225
40x2 CUDA Blocks	40x2,000=80,000	6.62 sec	83
20x4 CUDA Blocks	20x2,000=40,000	2.12 sec	53
20x4 CUDA Blocks 4 reads	20x2,000=40,000	1.38 sec	34
20x4 CUDA Blocks 4 reads, use shm	20x2,000=40,000	0.88 sec	22

CUDA V5 – Interleave and use shared memory

Use shared memory for the most used bins.

Characteristic	Value	vs initial
Streaming Input B/W	~177 GB/sec	~13.5x
Histogram Zero-ing B/W	~88 GB/sec	~13.5x
Atomic Increments	~44.0×10^9/sec	~13.5x
Streaming L2 Read B/W	~708 GB/sec	~54.5x
Histogram Working Set Size	~3.5 MB	~0.25x

CUDA V6 – Now center the data



CUDA V6 – Now center the data

Center the histogram eye for maximum shared memory use.

	Histograms	Time	µs/histogram
1 C Thread	2,000	3.5 sec	1,750
40 C Threads	40x2,000 = 80,000	7.95 sec	99
80x1 CUDA Blocks	80x2,000=160,000	35.97 sec	225
40x2 CUDA Blocks	40x2,000=80,000	6.62 sec	83
20x4 CUDA Blocks	20x2,000=40,000	2.12 sec	53
20x4 CUDA Blocks 4 reads	20x2,000=40,000	1.38 sec	34
20x4 CUDA Blocks 4 reads, use shm	20x2,000=40,000	0.88 sec	22
20x4 CUDA Blocks, 4 reads, use shm, center	20x2,000=40,000	0.90 sec	23

CUDA V6 – Now center the data

Center the histogram eye for maximum shared memory use.

Characteristic	Value	vs initial
Streaming Input B/W	~177 GB/sec	~13.5x
Histogram Zero-ing B/W	~88 GB/sec	~13.5x
Atomic Increments	~44.0×10^9/sec	~13.5x
Streaming L2 Read B/W	~708 GB/sec	~54.5x
Histogram Working Set Size	~3.5 MB	~0.25x

Sum Up

CUDA Challenges

1. **Warps contend, so we want to use atomic add**
   - **There is no atomic add on bytes!**
2. **GPU L1 caches are read optimized**
3. **We want to spread the work evenly over the blocks (SMs)**

Summary and Lessons

Benchmarking

1. **Be careful NOT to run out of cache when benchmarking!**
2. **Don't forget the "--optimize 2" option when compiling the C!**
   No sandbagging the baseline.

Optimizing GPUs & Volta

1. **Volta L2 has ~3x to ~4x the DRAM Bandwidth. You can use it.**
2. **Volta L1 writes though to the L2, unlike x86 caches**
3. **Shared Memory is write friendly, but you have to manage it (duh!)**

Finally

1. **GPUs can do well on loads that are 'obviously' CPU friendly.**