EECS 468: Lab 3 Journal
Scott Young, Kapil Garg

Purpose

This lab journal is used to capture all work done for EECS 468, Lab 3: Histograms. Here, we illustrate what each of the optimizations we tried were, any difficulties we had with them, and what the results of them were. We first begin with some "back-of-the-napkin" calculations to determine what the theoretical limits of our code likely are and then proceed to detail each of the optimizations we made.

Maximum Possible Speed

Assumptions: Our code will mostly be limited by bandwidth and not arithmetic. Thus we compute the lower bound to be the amount of time it takes for data to processed through the GPU. Furthermore, we assume that memory interactions and overhead for running 1000 iterations is minimal.

$$elements = INPUT\_HEIGHT * INPUT\_WIDTH * sizeof(uint32\_t) = 3984 * 4096 * 4\ bytes = 65273856\ bytes$$
$$680\_gpu\_max = 192.2\ GB/s$$

$$T_{1\ Iteration} = \frac{elements}{680\_gpu\_max} = \frac{65273856\ bytes}{192.2 GB/s} = 0.339614235\ ms$$
$$T_{1000\ Iterations} = T_{1\ Iteration} * 1000 = 0.339614235\ s$$

Reference Solution - Host CPU

11.0697 seconds @ 1000 iterations, 10 sample average

Optimization 1: Single Threaded Application

- Goal
  - To implement a very, very simple working version of a histogram generator that works on the GPU (this is not really an optimization)
  - Check if memory is copied correctly to the GPU
  - Execute histogram kernel on GPU
  - Check if histogram copied correctly back to host
- Difficulties
  - Copying the input file as a matrix was difficult. copied input file as a 1d array instead
- Time Spent
  - 8 hours (this includes setting up all the scaffolding code for copying matrices, freeing memory, etc.)
- Speedup
  - There was no speedup provided as this is effectively a benchmark version
  - 7720 seconds @ 1000 iterations (very bad)

```
__global__ void HistKernel(uint32_t *deviceImage, uint8_t *deviceBins, size_t height, size_t width) {
	for(size_t i = 0; i < height; i++) {
		for(size_t j = 0; j < width; j++) {
			const uint32_t value = deviceImage[i * width + j];
			if (deviceBins[value] < UINT8_MAX) {
				deviceBins[value]++;
			}
		}
	}
}

void opt_2dhisto(uint32_t *deviceImage, uint32_t *deviceBins32, uint8_t *deviceBins, size_t height, size_t
width) {
	HistKernel <<<1, 1>>> (deviceImage, deviceBins32, height, width);
	cudaThreadSynchronize();
}
```

Optimization 2: One Block, Full Occupancy

- Goal
  - Using multiple threads on one block
- Difficulties
  - Need to convert to uint8 for final output. We had to write another kernel to convert uint32 to uint8
- Time Spent
  - 6 hours
- Speedup
  - 38.807 seconds @ 1000 iterations, 10 samples
  - Iterations: [39.02, 39.016, 39.022, 39.013, 38.614, 38.691, 38.55, 39.015, 38.583, 38.547]

```
__global__ void HistKernel(uint32_t *deviceImage, uint32_t *deviceBins32, size_t height, size_t width) {
	size_t numThreads = (height * width)/HISTO_WIDTH;

	for (size_t j = threadIdx.x * numThreads; j < numThreads * (threadIdx.x + 1); j++){
		uint32_t value = deviceImage[j];

		if(deviceBins32[value] < UINT8_MAX) {
			atomicAdd(&deviceBins32[value], 1);
		}
	}
}

__global__ void HistKernel32to8(uint32_t *deviceBins32, uint8_t *deviceBins, size_t height, size_t width) {
	int index = blockIdx.x * blockDim.x + threadIdx.x;
	deviceBins[index] = (deviceBins32[index] < UINT8_MAX) ? (uint8_t) deviceBins32[index] : (uint8_t)
UINT8_MAX;
}

void opt_2dhisto(uint32_t *deviceImage, uint32_t *deviceBins32, uint8_t *deviceBins, size_t height, size_t
width) {
	cudaMemset(deviceBins32, 0, HISTO_HEIGHT * HISTO_WIDTH * sizeof(uint32_t));
	HistKernel <<<1, HISTO_WIDTH>>> (deviceImage, deviceBins32, height, width);
	HistKernel32to8 <<<HISTO_HEIGHT, HISTO_WIDTH>>> (deviceBins32, deviceBins, height, width);
	cudaThreadSynchronize();
}
```

Optimization 3: Uncoalesced Access

- Goal
  - Using uncoalesced access of elements
  - Normally, coalesced access of memory results in great speed-ups of code. However, in our case with images, we believe that non-coalesced memory can perform better because the overhead of atomicAdd serialization is much greater than the increased latency from using non-coalesced memory vs. coalesced memory. This will happen because images usually have similar values next to each other and using non-coalesced memory will let us reduce the amount of “hits” to the same bin when using atomicAdd.
- Difficulties
  - None
- Time Spent
  - 3 hours
- Speedup
  - 9.275 seconds @ 1000 iterations, 10 samples
  - Images tend to have similar values next to each other so by accessing them uncoalesced we can reduce atomicAdd serialization

```
__global__ void HistKernel(uint32_t *deviceImage, uint32_t *deviceBins32, size_t height, size_t width) {
        size_t globalTid = blockIdx.x * blockDim.x + threadIdx.x;
        size_t numThreads = blockDim.x;

        for (size_t j = globalTid; j < height * width; j  += numThreads){
                uint32_t value = deviceImage[j];

                if(deviceBins32[value] < UINT8_MAX) {
                        atomicAdd(&deviceBins32[value], 1);
                }
        }
}
```

Optimization 4: Shared Memory → Global Memory

- Goal
  - Using shared memory
- Difficulties
  - Checking for bank conflicts
- Time Spent
  - 8 hours
- Speedup
  - 7.940 seconds @ 1000 iterations
  - Reduces need to copy as much memory to global memory and improve memory latency

```
__global__ void HistKernel(uint32_t *deviceImage, uint32_t *deviceBins32, size_t height, size_t width) {
        size_t globalTid = blockIdx.x * blockDim.x + threadIdx.x;
        size_t numThreads = blockDim.x;
        __shared__ uint32_t partialHist[HISTO_WIDTH];
        partialHist[globalTid] = 0;
        __syncthreads();
        for (size_t j = globalTid; j < height * width; j += numThreads){
                uint32_t value = deviceImage[j];
                if(partialHist[value] < UINT8_MAX) {
                        atomicAdd(&partialHist[value], 1);
                }
        }
        __syncthreads();
        atomicAdd(&deviceBins32[globalTid], partialHist[globalTid]);
}

void opt_2dhisto(uint32_t *deviceImage, uint32_t *deviceBins32, uint8_t *deviceBins, size_t height, size_t
width) {
        HistKernel <<<1, HISTO_WIDTH>>> (deviceImage, deviceBins32, height, width);
        HistKernel32to8 <<<HISTO_HEIGHT, HISTO_WIDTH>>> (deviceBins32, deviceBins, height, width);
        cudaThreadSynchronize();
}
```

Optimization 5: Multiple Blocks

- Goal
  - Using multiple blocks
  - Calculated for 16 blocks via Occupancy Calculator
- Difficulties
  - None
- Time Spent
  - 1 hour
- Speedup
  - 0.776 seconds @ 1000 iterations, 10 sample average
  - Iterations: [0.779, 0.778, 0.777, 0.773, 0.78, 0.773, 0.78, 0.773, 0.776, 0.771}

```
void opt_2dhisto(uint32_t *deviceImage, uint32_t *deviceBins32, uint8_t *deviceBins, size_t height, size_t
width) {
	cudaMemset(deviceBins32, 0, HISTO_HEIGHT * HISTO_WIDTH * sizeof(uint32_t)); //zeros

	//8 multiprocessors * 2 blocks

	HistKernel <<<16, HISTO_WIDTH>>> (deviceImage, deviceBins32, height, width);
	HistKernel32to8 <<<HISTO_HEIGHT, HISTO_WIDTH>>> (deviceBins32, deviceBins, height, width);
	cudaThreadSynchronize();
}
```

Optimization 6: Overloaded __nv_min function

- Goal
  - Use builtin min function
- Difficulties
  - None
- Time Spent
  - 2 hours
- Speedup
  - 0.7676 seconds @ 1000 iterations, 10 sample average
  - Using built-in functions allows for better clarity in code
  - Min function is overloaded by nvidia, optimized code for CUDA
  - Iterations: [0.773, 0.768, 0.772, 0.764, 0.769, 0.764, 0.772, 0.763, 0.768, 0.763]

```
__global__ void HistKernel32to8(uint32_t *deviceBins32, uint8_t *deviceBins) {
        // convert int32 to int8; overloaded __nv_min function
        int index = blockIdx.x * blockDim.x + threadIdx.x;
        deviceBins[index] = (uint8_t) min(deviceBins32[index], UINT8_MAX);
}
```

Optimization 7: AtomicAdd check

- Goal
  - Adding a check if atomicadd is needed
- Difficulties
  - None
- Time Spent
  - 1 hour
- Speedup
  - 0.766 seconds @ 1000 iterations, 10 sample average
  - Allows for better protection against overflow and at no cost in terms of time
  - Reduces serialization during atomicadds
  - Iterations: [0.771, 0.774, 0.764, 0.764, 0.768, 0.764, 0.764, 0.765, 0.765, 0.764]

```
if (deviceBins32[threadIdx.x] < UINT8_MAX) {
        atomicAdd(&deviceBins32[threadIdx.x], partialHist[threadIdx.x]);
}
```

Optimization 8: Unrolling for loop

- Goal
  - Unroll the for loop
- Difficulties
  - None
- Time Spent
  - 4 hours
- Speedup
  - 0.720 seconds @ 1000 iterations, 10 sample average
  - Unrolling the for loop allows the compiler to better optimize the machine code behind the loop at the cost of larger binary size and higher register use
  - Would only work if number of elements is divisible by 4
  - Iterations: [0.720, 0.722, 0.722, 0.717, 0.717, 0.722, 0.725, 0.717, 0.721, 0.717]

```
#pragma unroll
        for (size_t j = globalTid; j < height * width; j += numThreads * 4) {
                if (partialHist[deviceImage[j]] < UINT8_MAX) {
                        atomicAdd(&partialHist[deviceImage[j]], 1);
                }

                if (partialHist[deviceImage[j + numThreads]] < UINT8_MAX) {
                        atomicAdd(&partialHist[deviceImage[j + numThreads]], 1);
                }

                if (partialHist[deviceImage[j + numThreads * 2]] < UINT8_MAX) {
                        atomicAdd(&partialHist[deviceImage[j + numThreads * 2]], 1);
                }

                if (partialHist[deviceImage[j + numThreads * 3]] < UINT8_MAX) {
                                atomicAdd(&partialHist[deviceImage[j + numThreads * 3]], 1);
                }
        }
        __syncthreads();
```

Optimization 9: More unrolling

- Goal
  - Unroll for loops even more
- Difficulties
  - None
- Time Spent
  - 1 hour
- Speedup
  - 0.711 seconds @ 1000 iterations, 10 sample average
  - Increasing the unrolling factor
  - Would only work with these number of elements (divisible by 12)
  - Iterations: [0.718, 0.713, 0.71, 0.71, 0.712, 0.71, 0.709, 0.709, 0.71, 0.713]

```
#pragma unroll
        for (size_t j = globalTid; j < height * width; j += numThreads * 12) {
                if (partialHist[deviceImage[j]] < UINT8_MAX) {
                        atomicAdd(&partialHist[deviceImage[j]], 1);
                }

                if (partialHist[deviceImage[j + numThreads]] < UINT8_MAX) {
                        atomicAdd(&partialHist[deviceImage[j + numThreads]], 1);
                }

                if (partialHist[deviceImage[j + numThreads * 2]] < UINT8_MAX) {
                        atomicAdd(&partialHist[deviceImage[j + numThreads * 2]], 1);
                }

                if (partialHist[deviceImage[j + numThreads * 3]] < UINT8_MAX) {
                                atomicAdd(&partialHist[deviceImage[j + numThreads * 3]], 1);
                }
                if (partialHist[deviceImage[j + numThreads * 4]] < UINT8_MAX) {
                                atomicAdd(&partialHist[deviceImage[j + numThreads * 4]], 1);
                        }
                if (partialHist[deviceImage[j + numThreads * 5]] < UINT8_MAX) {
                                atomicAdd(&partialHist[deviceImage[j + numThreads * 5]], 1);
                        }
                if (partialHist[deviceImage[j + numThreads * 6]] < UINT8_MAX) {
                                atomicAdd(&partialHist[deviceImage[j + numThreads * 6]], 1);
                        }
                if (partialHist[deviceImage[j + numThreads * 7]] < UINT8_MAX) {
                                atomicAdd(&partialHist[deviceImage[j + numThreads * 7]], 1);
                        }
                if (partialHist[deviceImage[j + numThreads * 8]] < UINT8_MAX) {
                                atomicAdd(&partialHist[deviceImage[j + numThreads * 8]], 1);
                        }
                if (partialHist[deviceImage[j + numThreads * 9]] < UINT8_MAX) {
                                atomicAdd(&partialHist[deviceImage[j + numThreads * 9]], 1);
                        }
```

```
        if (partialHist[deviceImage[j + numThreads * 10]] < UINT8_MAX) {
                        atomicAdd(&partialHist[deviceImage[j + numThreads * 10]], 1);
                }
        if (partialHist[deviceImage[j + numThreads * 11]] < UINT8_MAX) {
                        atomicAdd(&partialHist[deviceImage[j + numThreads * 11]], 1);
                }

}
__syncthreads();
```

Optimization 10: Coalesced global memory write

- Goal
  - Reduce atomicadd serialization to global memory by writing all elements instead
  - Sum up all partials in second kernel
- Difficulties
  - None
- Time Spent
  - 2 hours
- Speedup
  - 0.712 seconds @ 1000 iterations, 10 sample average
  - No speed up, trade off between 16 way atomicadd serialization and more global memory access
  - Iterations: [0.718, 0.713, 0.71, 0.71, 0.712, 0.71, 0.709, 0.709, 0.71, 0.713]

```
if (deviceBins32[globalTid] < UINT8_MAX) {
                deviceBins32[globalTid] = partialHist[threadIdx.x];
        }
}
__global__ void HistKernel32to8(uint32_t *deviceBins32, uint8_t *deviceBins) {
        // convert int32 to int8; overloaded __nv_min function
        int index = blockIdx.x * blockDim.x + threadIdx.x;
        uint32_t total = 0;
        for (size_t i=0; i < 16; i++) {
                total += deviceBins32[i*HISTO_WIDTH + threadIdx.x];
        }
        deviceBins[index] = (uint8_t) min(total, UINT8_MAX);
}
```

Optimization 11: Reducing Memory Reads using uint32 (in-progress)

- Goal
  - To reduce the amount of memory reads by combining 4 uint8 inputs into a single uint32
  - Should reduce computation time due to memory latency
- Difficulties
  - We originally wanted to use uint64 but found that it is not supported for our version of CUDA (requires compute capability 6.x or higher, see: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#ixzz4YzDf3GIi)
  - We currently do not have a working version of this
- Time Spent
  - 6 hours
- Speedup
  - NA, this is currently not functioning