DREXEL UNIVERSITY
Electrical and Computer Engineering
College of Engineering

**Drexel University**

**Electrical and Computer Engineering Dept.**

**ECEC-413**

# Counting Sort with CUDA

**Chris Kasper**
**Prof. Naga Kandasamy**
**DATE: 06/8/19**

## GPU Implementation (using CUDA)

The first part of the CUDA implementation is to initialize the variables for the device. There are also three different kernel functions for performing the counting sort process. One for the histogram generation, another for the inclusive prefix scan, and finally one for the sorting. The kernels for the histogram generation and the final sort use the same kernel configuration, while the kernel for the prefix scan has a different one.

The initialization code is as follows:

```
/* GPU implementation of counting sort. */
void
compute_on_device (int *input_array, int *sorted_array, int num_elements, int range)
{
    struct timeval start, stop;

    int *d_input = NULL;
    int *d_output = NULL;

    /* Histogram step variables */
    int *hist = (int *) malloc (sizeof (int) * HISTOGRAM_SIZE);
    memset (hist, 0, HISTOGRAM_SIZE);

    int *d_hist = NULL;

    /* Prefix scan variables */
    int *scan_out = (int *) malloc (sizeof (int) * HISTOGRAM_SIZE);
    memset (scan_out, 0, sizeof (int) * HISTOGRAM_SIZE);

    int *d_scan = NULL;

    /* Allocate space on GPU for the input */
    cudaMalloc((void**) &d_input, num_elements * sizeof(int));
    cudaMemcpy(d_input, input_array, num_elements * sizeof(int), cudaMemcpyHostToDevice);

    /* Allocate space on GPU for the output */
    cudaMalloc((void**) &d_output, num_elements * sizeof(int));
    cudaMemcpy(d_output, sorted_array, num_elements * sizeof(int), cudaMemcpyHostToDevice);

    /* Allocate space on GPU for histogram */
    cudaMalloc((void**) &d_hist, HISTOGRAM_SIZE * sizeof(int));
    cudaMemcpy(d_hist, hist, HISTOGRAM_SIZE * sizeof(int), cudaMemcpyHostToDevice);

    /* Allocate space on GPU for scan output */
    cudaMalloc((void**) &d_scan, HISTOGRAM_SIZE * sizeof(int));
    cudaMemcpy(d_scan, scan_out, HISTOGRAM_SIZE * sizeof(int), cudaMemcpyHostToDevice);
```

```
/* Kernel config */
dim3 thread_block (THREAD_BLOCK_SIZE, 1, 1);
dim3 grid (NUM_BLOCKS, 1);

gettimeofday (&start, NULL);

/* Histogram generation */
histogram_kernel<<<grid, thread_block>>>(d_input, d_hist, num_elements, HISTOGRAM_SIZE);
cudaDeviceSynchronize ();
check_for_error ("KERNEL FAILURE: Histogram");

/* Kernel config for scan */
grid.x = 1;
thread_block.x = HISTOGRAM_SIZE;

int shared_mem_size = sizeof (int) * HISTOGRAM_SIZE;

/* Inclusive prefix scan */
scan_kernel<<< grid, thread_block, 2 * shared_mem_size >>>(d_scan, d_hist, HISTOGRAM_SIZE);
cudaDeviceSynchronize ();
check_for_error ("KERNEL FAILURE: Prefix scan");

/* Kernel config for count sort */
grid.x = NUM_BLOCKS;
thread_block.x = THREAD_BLOCK_SIZE;

/* Count sort */
counting_sort_kernel<<<grid, thread_block>>>(d_output, d_scan, HISTOGRAM_SIZE);
cudaDeviceSynchronize ();
check_for_error ("KERNEL FAILURE: Count sort");
gettimeofday(&stop, NULL);

/* Copy the result back from the GPU and store. */
cudaMemcpy (sorted_array, d_output, num_elements * sizeof (int), cudaMemcpyDeviceToHost);

printf("Execution time = %f\n",stop.tv_sec - start.tv_sec + (stop.tv_usec - start.tv_usec)/(float)1000000);
```

Regarding the kernel functions, it is possible that all three functions could've been combined to be just one function. However, it was more ideal to keep them all separate, which helped testing wise in order to figure out possible bugs. The histogram generation kernel utilizes the 10240 threads to quickly compute the histogram, while using shared memory and atomics. The prefix scan kernel has only 256 threads (histogram size), and also uses shared memory. Finally, the counting sort kernel has the 10240 threads, and each thread is responsible for populating the final sorted array based on the difference between the current position and the one right before, unless the current position is zero.

The kernel code is as follows:

```cpp
/* Counting sort: histogram generation */
__global__  void histogram_kernel(int *input_data, int *histogram, int num_elements, int histogram_size)
{
    __shared__ unsigned int s[HISTOGRAM_SIZE];

    /* Initialize the shared memory area. */
    if(threadIdx.x < histogram_size)
        s[threadIdx.x] = 0;

    __syncthreads();

    unsigned int offset = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int stride = blockDim.x * gridDim.x;

    while (offset < num_elements) {
        atomicAdd (&s[input_data[offset]], 1);
        offset += stride;
    }

    __syncthreads();

    /* Accumulate the histogram in shared memory into global memory. */
    if (threadIdx.x < histogram_size)
        atomicAdd (&histogram[threadIdx.x], s[threadIdx.x]);

    return;
}
```

```cpp
/* Counting sort: inclusive prefix scan */
__global__ void scan_kernel (int *out, int *in, int n)
{
    /* Dynamically allocated shared memory for storing the scan array. */
    extern  __shared__  int temp[];

    int tid = threadIdx.x;

    /* Indices for the ping-pong buffers. */
    int pout = 0;
    int pin = 1;

    /* Load the in array from global memory into shared memory. */
    temp[pout * n + tid] = in[tid];

    for (int offset = 1; offset < n; offset *= 2) {
        pout = 1 - pout;
        pin  = 1 - pout;
        __syncthreads();

        temp[pout * n + tid] = temp[pin * n + tid];

        if (tid >= offset)
            temp[pout * n + tid] += temp[pin * n + tid - offset];
    }

    __syncthreads();

    out[tid] = temp[pout * n + tid];
    return;
}
```

```
/* Counting sort */
__global__ void counting_sort_kernel (int *out, int *in, int n)
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int stride = blockDim.x * gridDim.x;

    int i, j, diff, start;

    for (i = 0; i < n; i++) {
        if (i == 0) {
            diff = in[i];
            start = 0;
        } else {
            diff = in[i] - in[i-1];
            start = in[i-1];
        }

        for (j = idx; j < diff; j+=stride)
            out[start+j] = i;
        __syncthreads();
    }

    return;
}
```

## Speedup

**Table 1: Speedup with Counting Sort on xunil-05**

| Number of Integers | CPU | GPU |
|---|---|---|
| 1,000,000 | 3.05 ms | 0.53 ms |
| 10,000,000 | 24.66 ms | 2.14 ms |
| 100,000,000 | 201.30 ms | 20.48 ms |

*Note: xunil-05 has a Nvidia 1080 GTX GPU*