ECEC 413
Aleksandar Aleksandrov

CUDA – Counting Sort

Discussion

The goal of this assignment is to implement a counting sort algorithm using CUDA. The algorithm consists of three main parts – histogram generation, inclusive scan, and sorted array generation.

The algorithm takes an input array of random numbers and first generates a histogram of the frequency of occurrence of every number in the set range. Then, it performs an inclusive scan on the bins of the histogram to find the starting index of every number in the sorted array. Every number has a corresponding bin in the histogram – the number 1, corresponds to bin 1. By performing an inclusive scan, the algorithm finds out how many numbers have been written so far in the sorted array and know the starting index of the next number. For this assignment, every step of the counting sort algorithm can be parallelized using CUDA.

Parallel Implementation

The first step is to generate a histogram of the numbers in the array. The "compute_on_device" function starts off by allocating memory on the GPU for the input array, the sorted array, and the histogram. The input values are also copied to the GPU. A separate kernel is created for the histogram generation – "histogram_generation_kernel". The kernel is running on 256 thread blocks and a grid size of:

(Number of Elements)/(Thread Block Size)

The histogram generation kernel takes arguments of the input array, a pointer to the global memory allocated for the histogram, and the number of elements in the array and histogram respectively. The kernel uses shared memory to generate the histogram – it is allocated at the beginning:

```
__shared__ unsigned int shared_histogram[HISTOGRAM_SIZE];
```

Then the kernel sets every bin in the histogram to 0. The histogram has a fixed number of elements equal to the possible range of the numbers in the input array – 256 in this case. Then the algorithm proceeds to fill the shared memory for every thread block.

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

The threads are then synchronized and the first 256 threads then populate the global memory with values in shared memory using the atomic add operation.

The scan and generate operations are implemented in a second kernel – "counting_sort_kernel". The first part of the kernel generates the scanned array of the histogram in shared memory and then the shared array is used to sort the input array. For this kernel, the execution grid consists of only one thread block with 256 threads since the histogram and the scanned array have 256 elements. The kernel accepts the pointer to the global memory of the histogram, a pointer to the sorted array memory, and the number of bins in the histogram as arguments.

The kernel uses shared memory for the scanned array of size two times the size of the histogram – the two halves of the array act as ping pong buffers.

```
__shared__ int scan_shared[2*HISTOGRAM_SIZE];
```

The algorithm then proceeds to perform an inclusive scan by switching the ping-pong buffers for every iteration.

```
    scan_shared[pout * n + tid] = histogram[tid];

    /*Perform Inclusive Scan*/
    for (offset = 1; offset < n; offset *= 2) {
        pout = 1 - pout;
        pin  = 1 - pout;
        __syncthreads();

        //coppies array to the second half of scanned array
        scan_shared[pout * n + tid] = scan_shared[pin * n + tid];

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

The inclusive scan contains the sum of all previous bins, including the current one. Every bin is operated by its corresponding thread ID. After the scan operation is performed, the threads are synchronized again and the kernel proceeds to generate the sorted array.

For the generation of the sorted array, the initial start index is set to 0 and then for every thread ID after 0, the value of the starting index is taken from the previous bin in the scanned array (tid-1):

```
    int start_idx = 0;

    if(tid > 0)
        start_idx = scan_shared[tid-1];
```

Then for every thread, a for loop iterates the number of times set in the corresponding bin in the histogram and writes the thread ID to the sorted array. The scanned array contains the starting index, and the histogram contains the amount of times the number has to be written to the array:

```
    for (j = start_idx; j < (start_idx+histogram[tid]); j++)
        sorted_array[j] = tid;
```

The results are recorded to the global memory of “sorted_array” and then are copied back to the CPU.

Results

Number of Elements	CPU Time	GPU Time (Including CPU – GPU communication)
10^5	0.000708s	0.177347s
10^6	0.003418s	0.151208s
10^7	0.023836s	0.189970s
10^8	0.240763s	0.707570s

Conclusion

For all array sizes, the CPU implementation outperforms the GPU implementation. These results can be explained by the fact that the counting sort operation is memory bound rather than compute bound, therefore the GPU implementation spends more time allocating and transferring memory back and forth from the device to the CPU, rather than performing calculations. The calculations performed by the GPU are rather simple. Only a limited number of threads are used for the scan and generation operations. The parallel version could provide a significant improvement if there are other computation-heavy algorithms performed on the sorted array.