Łukasz Myśliński

Histogram calculation on Intel Xeon Phi

Introduction

An image histogram is a type of histogram that acts as a graphical representation of the tonal distribution in a digital image. It plots the number of pixels for each tonal value. By looking at the histogram for a specific image a viewer will be able to judge the entire tonal distribution at a glance. A single-threaded histogram computation is fairly simple:

```
void histogram_cpu(int *host_data, size_t data_size, int* histogram){
    for(int i=0;i<data_size;i++){
        histogram[host_data[i]]++;
    }
}
```

Parallel solution

A simple parallel implementation in OpenMP consist of parallelization of the for loop, while making sure the sum operation is performed atomically.

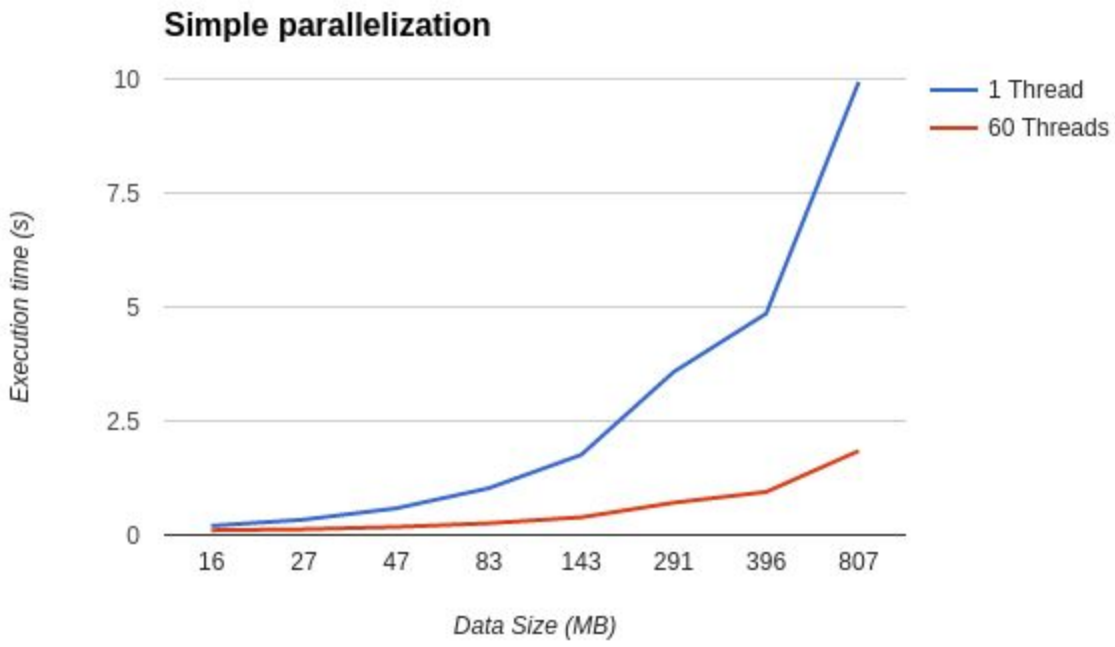
```
#pragma omp parallel for num_threads(60)
for(int i=0;i<dataSize;i++){
    #pragma omp atomic
    device_results[device_data[i]]++;
}
```

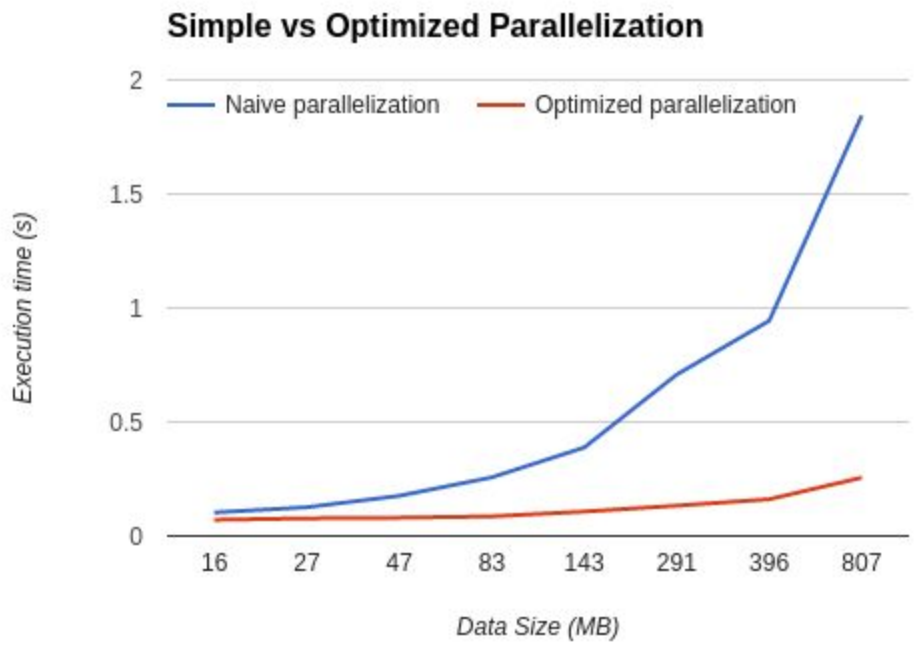
Optimized parallel solution

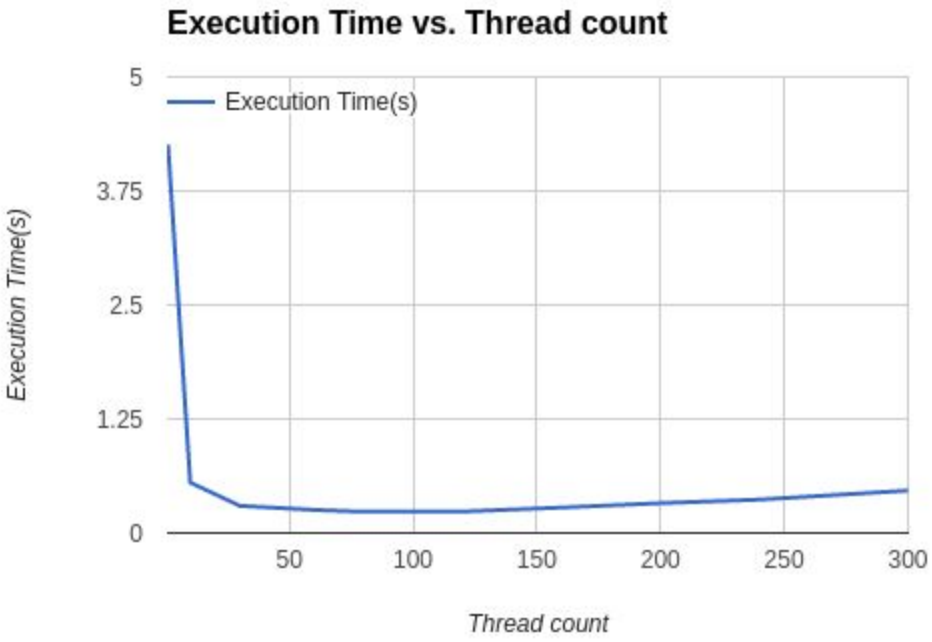
The solution above leaves much place for improvements. Instead of performing the reduction operation on one histogram, we divide it into N subhistograms, one for each thread. This allows each thread to operate on its own data chunks and update the histogram without race conditions. Furthermore, we align each of the histograms by 64 bits, to get rid of cache line invalidation. This optimization allows us to get rid of the *#pragma omp atomic* in the next parallel section - each thread can merge its subhistogram seamlessly.

```
__declspec (align(64)) int **histograms = new int *[THREAD_COUNT];
for (int i = 0; i < THREAD_COUNT; ++i)
    histograms[i] = (int *) calloc(256, sizeof(int));

#pragma omp parallel for num_threads(THREAD_COUNT)
for (int i = 0; i < dataSize; i++) {
    histograms[omp_get_thread_num()][device_data[i]]++;
}

#pragma omp parallel for num_threads(THREAD_COUNT)
for (int j = 0; j < 256; j++) {
    for (int i = 0; i < THREAD_COUNT; i++) {
        device_results[j] += histograms[i][j];
    }
}
```

Experiments







Conclusions

Although the speed-up due to optimization is very significant, the code does not scale well with increasing thread count. Once we surpass the number of physical cores present on the device,

the performance starts to decrease. This is likely due to overhead of spawning so many threads exceeding their computation time.