CPP - Optional assignments

Maxim van den Berg (11867000), Sam van Kampen (11874716) and Robin Wacanno (11741163)

December 2018

Lessons 3-6

Assignment 1: Histogram

Introduction The computation of a histogram for the analysis of an image's colour values lends itself very well to parallelization, as the image can be easily split up into multiple independent chunks with which every thread can work simultaneously. However, data races can occur easily if all threads can write to the same memory locations, namely the histogram data.

We made use of the standard low-level multi-threading model pthreads, which gave us a lot of control at both the implementation- and design-level. We will only use the darkness-values in a grayscale image instead of all three RGB values. As one can imagine, the pthreads implementation for such an analysis will be analogous.

The worker threads An optimal choice for the total number of worker threads is dependent on multiple factors, such as the total workload of the application, as well as the amount of cores of the given machine the application is run on. Additionally, there is often a specific balance between the lost computation time from the overhead and gained efficiency from the parallelism. For this implementation, we chose to divide the workload over all available CPU cores, which should give a somewhat consistent high performance when running on multiple kinds of hardware. With n cores, the data per thread will then be $\frac{\text{total}}{n}$, where *total* indicates the full size of the image. When the image is represented as a 1-dimensional array, as we assume in our implementation, each thread will take consecutive elements of the array in order to make full usage of the CPU caching. Furthermore, the computation of the histogram has no dependencies on other parts of the image at all, and thus each thread can simply iterate through their part of the array and update the histogram appropriately.

Synchronization Pthreads allows us to keep both the image array and the histogram as global variables, and therefore accessible by all threads. For the image this is perfect, as each thread will only access their respective sections of the array. But keeping just a single histogram will introduce data-races, since it is likely that multiple threads will find a pixel with the same value at the same time. To solve this issue, each thread will keep their own histogram, which will be merged when all threads are finished with their computation. This will be not be done in parallel, since we have reasoned that the computation time of a fixed iteration size of 256 will be negligible in comparison to larger images.

Performance We can expect the performance to increase almost linearly with the amount of threads, but we also have to account of the time spend on dividing the workload, waiting for all threads (in fact the slowest thread) to finish and the reduction of all histograms into one. For this application we actually get minimal overhead, as all threads can work without having to communicate (wait for each other) at all. The division of the workload will also be small, since they can already access the data immediately and only the assignment of pointers is required. Therefore we can expect the performance to increase almost linearly with the amount of threads, especially when the image gets larger and larger.

A mostly complete implementation as described above can be found in the following pseudo-code.

Listing 1: Parallel histogram calculation

```
byte Image[]; // one dimensional image array

struct worker_data_t {
    int id;
    int num_threads;
    int* histogram;
};

main(int argc, char *argv)
{
    int num_threads = NUM_CPU_THREADS;

    // Allocate arrays
    int *histograms[num_threads];
    int final_histogram[255] = { 0 };
    pthread_t threads[num_threads];

    // Start the worker threads
    int size = ceil(Image.size / data->num_threads);
    for (int n = 0; n < num_threads; n++)
    {
        *histogram[n] = calloc(sizeof int * 255);
        worker_data_t data = { n, size, histogram[n] };
        pthread_create(threads[n], &function_worker, &data);
    }

    // Join threads
    for (int n = 0; n < num_threads; n++)
    {
        pthread_join(threads[n], NULL)
    }

    // Reduction
    for (int i = 0; i < num_threads; i++)
    {
        for (int j = 0; j < 256; j++)
        {
            final_histogram[j] += histograms[1][j]
        }
    }
}


function_worker(worker_data_t* data)
{
    int end = max(data->size*(data->id+1), Image.size)
    for (int i = data->size*data->id; i  < end; i++)
    {
        histogram[image[i]]++;
    }
}
```

Assignment 2: Conceptual computational models

Farmer/workers

(a) No. OpenMP handles most aspects of the parallelization process, and as a consequence leaves little room for control by the programmer. The openMP version we worked with during this course does not have functionality for more complex features such as this, where we specify different courses of action for the *master* and *worker* threads.

(b) Yes. A simple dummy-pogram is as follows. One can see that this kind of program can be useful for a lot of different applications.

```
worker(data) {
    for (t in range)
        data_new = compute_sum(data)
        send_to_master(data_new)
        data = receive_from_master();
}
master() {
    while (active_threads > 0)
        data = receive_from_thread(a);
        send_to_thread(b, data);
}
main() {
    pthread workers[n]
    pthread_create(workers, data)
    pthread_create(master)

    pthread_join(workers and master);
    export(data)
}
```

(c) Yes. A simple dummy program shows how a master-process would be used with MPI. The master will be the process with an id of 0, while the others will act as workers.

```
main() {
    id = get_my_id();

    if (id is 0)
        send_data_to_processes(data)
        data = receive_data_from_processes()
        export(data);
    else
        data = receive_from_master()
        data = compute(data)
}
```

Divide and conquer

(a) Yes. The histogram could be easily computed with openMP.

(b) Yes. The histogram as described in assignment 1 is an example of this.

(c) Yes.

Domain decomposition

(a) No. You can keep the complete data present while performing computation on it. Therefore domain decomposition does not necessarly have to occur.

(b) No. The same as with openMP applies.

(c) Yes.

Functional decomposition

(a) No, structures like for loop are parallelised and each thread will execute the same instructions inside these structures.

(b) Yes (see: multiple threads where one does input, the other does processing, etc). For pseudo-code, see the example for farmers/workers.

(c) Yes, in principle you could do the same thing as described with pthreads, though it is more difficult.

Assignment 4: Performance

1. Superlinear speedup may, for instance, be possible in case the data used does not fit in the cache of a single processor, but does fit in the cache of multiple processors. In this case, the overhead of loading things into the cache and discarding them that is present in the version of the program that runs on a single processor (sequential or not) can be eliminated when running the program on multiple processors, and the speedup can thus be superlinear in case the impact of the cache size is not dwarfed by the synchronisation and initialisation of a parallel version of the program.

2. In comparison to OpenMP, MPI gets more efficient when the number of samples increases. OpenMP has less of an impact when the number of samples is low, however. The impact that MPI has when the number of samples is low increases when the number of processes increases.

3. You could measure the execution time of the parallelisable part of your sequential program, and use Amdahl's law to figure out what the theoretical speedup of the execution of the whole task is.
4. In the same way that Amdahls law is taken into account, but uh, something about increasing the problem size?
5. It greatly depends, but generally 2x more processes/cores. Initially no data is read from file system, but generated by the algorithm. Writing the data to disk after computation is not vary significant. In case the amount of data wouldn't entirely fit in main memory, faster disk storage would be highly beneficial.

GPU

Assignment 6: Advanced CUDA programming - Heterogeneous computing

Listing 2 shows a heterogeneous implementation of the vector addition algorithm for *CUDA*. It tries to evenly balance the addition computation between the GPU and CPU, thus ensuring they continue to be as active as possible. The improved results in table 1 can also be attributed to the fact that only the parts of the arrays which are needed by the GPU are copied to the memory of the GPU. Homogeneous execution time is barely affected by the increase of vector size thus showing that the memory overhead is highly significant in the total execution time. Additionally, since this test was run on a super computer, the highly performant CPU present in the system attributed to the increased performance.

Vector size	Homogeneous GPU execution time	Heterogeneous GPU+CPU execution time
131072	457 ms	6.42 ms
262144	453 ms	12.2 ms
524288	480 ms	22.2 ms

Table 1: Execution time for homogeneous and heterogeneous vector add implementations

Listing 2: CUDA Heterogeneous Computing

```
__global__ void vectorAddKernelImp(float* deviceA, float* deviceB, float*
        ↪ deviceResult, int indexBegin, int size)
{
    unsigned index = blockIdx.x * blockDim.x + threadIdx.x;

    if (indexBegin + index < size)
        deviceResult[indexBegin+index] = deviceA[indexBegin+index] +
        ↪ deviceB[indexBegin+index];
}

struct cpu_data {
    int* cur_index;
    int size;
    float* a;
    float* b;
    float* result;
};

pthread_mutex_t lock_wait = PTHREAD_MUTEX_INITIALIZER;


void* CPUworker(void *data_in)
{
    cpu_data* data = (cpu_data*) data_in;
    int* i = data->cur_index;

    while (true)
    {
        pthread_mutex_lock(&lock_wait);
        if (*i < data->size)
        {
            data->result[*i] = data->a[*i] + data->b[*i];
            *i++;
            pthread_mutex_unlock(&lock_wait);
        }
        else
        {
            pthread_mutex_unlock(&lock_wait);
            break;
        }
    }

    return NULL;
}

void vectorAddCudaImp(int n, float* a, float* b, float* result)
{
    int threadBlockSize = 512;

    // memory allocation
    [...]

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    // copy the original vectors to the GPU
    checkCudaCall(cudaMemcpy(deviceA, a, n*sizeof(float), cudaMemcpyHostToDevice));
    checkCudaCall(cudaMemcpy(deviceB, b, n*sizeof(float), cudaMemcpyHostToDevice));

    // execute kernel
    cudaEventRecord(start, 0);

    int kernelSize = 1024; // Dit veranderen
    int cur_index = 0;
    int prev_index;
    pthread_mutex_lock(&lock_wait);
    pthread_t cpu_worker;
    cpu_data data = { &cur_index, n, a, b, result };
    pthread_create(&cpu_worker, NULL, &CPUworker, (void*) &data);

    while (cur_index < n)
    {
        prev_index = cur_index;
        cur_index += kernelSize;
        if (cur_index > n)
            cur_index = n;

        vectorAddKernelImp<<<kernelSize/threadBlockSize, threadBlockSize>>>(deviceA,
        ↪ deviceB, deviceResult, prev_index, n);

        // check whether the kernel invocation was successful
        checkCudaCall(cudaGetLastError());

        // wait for kernel execution, and start the cpu worker.
        pthread_mutex_unlock(&lock_wait);
        checkCudaCall(cudaMemcpy(&result[prev_index], &deviceResult[prev_index],
        ↪ kernelSize * sizeof(int), cudaMemcpyDeviceToHost));
        pthread_mutex_lock(&lock_wait);
    }

    cudaEventRecord(stop, 0);

    checkCudaCall(cudaFree(deviceA));
    checkCudaCall(cudaFree(deviceB));
    checkCudaCall(cudaFree(deviceResult));

    // print the time the kernel invocation took, without the copies!
    float elapsedTime;
    cudaEventElapsedTime(&elapsedTime, start, stop);

    cout << "kernel invocation took " << elapsedTime << " milliseconds" << endl;
}
```