University of Amsterdam

Parallelization with CUDA

Exploring heat dissipation, histogram and smoothing filter

Kian Paimani / 11622849
Jonas Theis / 12142964

Universiteit van Amsterdam
Vrije Universiteit Amsterdam
Course: Programming Multi-core and Many-core Systems
Course instructor: dr. Clemens Grelck

20th August 2018

1 Introduction

In this report we will demonstrate our findings from experimenting with programming NVIDIA GPUs with *CUDA*. For this reason, before explaining the results, we will briefly discuss the available CPUs and GPU on DAS worker nodes and examine their ability for multi and many-core execution [1].

As opposed to the previous assignments, where we discussed CPU programming with *OpenMP* and *PThreads*, we are now going to explore GPU programming (*SIMT* = Single Instruction Multiple Thread execution). *CUDA* proposes a low-level programming model of a GPU architecture, where a `thread` represents a core, a `block` represents a multiprocessor unit and the `thread grid` stands for the device.

To further elaborate on the concepts of CUDA, three different types of experiments have been conducted, namely:

- **Heat Dissipation**: An iterative matrix cell update process based on neighbouring cells.
- **Histogram**: One-shot iteration over a grayscale image, represented as 2 dimensional array, to compute its histogram.
- **Smoothing filter**: Convolution is a one-step modification applied to a matrix, namely images in most cases. Through the operation, a smaller matrix, mostly named a *kernel* is slided thorough the source matrix. In each sliding step, each cell value of the kernel is multiplied by its corresponding cell value in the source image. The result of these modifications are summed together and then divided by the number of cells in the kernel. Finally the result is replaced at the cell correspondent with the center of the kernel in the source matrix.

For the rest of this report, in section 2 we will explain the implementation details of each of these tasks. In section 3 we will evaluate the performance of each implementation and compute their speedup based on a specific baseline. The baseline for each part is, of course, different and it is explained in the respective section. Finally, in section 4 we conclude about CUDA and the explored concepts of optimizing CUDA code.

2 Implementation Details

In this section we will describe implementation details of all three parts of this assignment. We will focus on the high level concepts and omit negligible details.

2.1 Heat Dissipation

The heat dissipation is implemented as a combination of different kernels, each performing one specific part of the update operation. Nonetheless, it is worth mentioning that in our initial attempt, we have tried to fit the entire operations into one single kernel. This kernel was later decomposed into multiple ones for the following main reasons, namely:

- Lack of global synchronization: the entire input, except for very small cases, will never fit into a single block (with the maximum number of threads in each block being 1024). Hence, they should be divided into multiple blocks and this requires all threads in all blocks to be synchronized. Even though this global barrier can be implemented with manual *active polling* (e.g. waiting on a `while loop`), it is far from being efficient.
- The fact that launching kernels is relatively cheap in CUDA and it implicitly solves the problem of synchronization.
- Avoiding thread divergence. The different operations that need to be applied in each iteration are not homogeneous among threads. In other words, all of the threads will not be active at the same time during the execution of the kernel. As an example, all threads must perform the cell update. For mirroring the first and last column, only a small group need to be active. In the reduction of `maxdiff` each group needs to perform different operations. All of these variations are signs of a *potentially* high threads divergence which is a big performance killer in CUDA.

The only downside of this approach is that the outer `for loop` needs to be moved to the host code and kernels have to be spawned for each iteration. Nonetheless, we have seen in our tests that this approach is faster.

The kernels executed in each iteration are as follows:

- `cell_update_kernel()`, which updates the value of each cell in the result pointer.
- `mirror_kernel()`, which copies the elements of the first and last column in parallel to the extra columns in the sides of the matrix
- `maxdiff_kernel()`, which reduces `maxdiff`.

Based on the previous description, the gist of the host code is as follows:

```
/* setup, copy data to device */

for (it = 0; it < p->maxiter; it++) {
       /* All cells will be updated in d_dest */
```

```
    cell_update_kernel<<<...>>>(/* ... */)

    /* update first and last column */
    mirror_kernel<<<...>>>(/* ... */);

    /* calculate diff, */
   maxdiff_kernel<<<...>>>(/* ... */);
   /* Copy just one value from maxdiff kernel out */
   cudaMemcpy(global_maxdiff, d_maxdiff, sizeof(double));
    if ( *global_maxdiff < p->threshold ) { break; }

    if ( p->printreports ) { /* do report */ }

    /* swap pointers for next iteration, if exists */
    { double *tmp = d_src; d_src = d_dst; d_dst = tmp; }
}

/* copy results back, cleanup */
```

Next, we will go through some of the details of the above structure, focusing specifically on performance related aspects.

One of the most important *hyperparameters* of any CUDA implementation is finding an optimized dimension for the grid and blocks. We will elaborate more on this matter and fine-tune inputs to different dimensions [1] in section 3. Nonetheless, we have chosen the following simple approach to ensure that our code works for *all* possible inputs: The number of threads per block is assumed to be fixed to 1024, divided into a two-dimensional space with shape of 32x32. Next, we have to ensure that the total number of threads is bigger than or equal to the input size. Hence, the number of blocks is set to a value that covers to input size. As an example, for an input of 1000x1000, we spawn 1024x1024 threads, divided into a `blockDim` of `32x32` and `gridDim` of `32x32`.

Registers dedicated to each thread are the fastest possible way to access data inside the threads. These accesses can even be more optimized by using the `__constant__` preprocessor identifier of CUDA, if the data is known at compile time and is *the same* among *all* threads. The coefficients of diagonal and direct neighbours is a perfect match to such use case:

```
__constant__ __device__ double c_cdir = 0.25 * M_SQRT2 / (M_SQRT2 + 1.0);
__constant__ __device__ double c_cdiag = 0.25 / (M_SQRT2 + 1.0);
```

Furthermore, normal C variables defined inside the kernel are also stored on register-level, with almost zero fetch time. Hence, we have tried as much as possible to avoid index computation and store common sub-expressions to efficiently access different elements of the matrix[2]. An example

[1] using tools such as KernelTuner

[2] the matrix is stored as a contiguous one dimensional vector

of this can be seen in the `update_kernel()`. The following variables are stored as base pointers to different neighboring rows of the current row. Furthermore, a variable is stored that keeps the exact index of the current cell.

```
/* i and j are row/col indexes based on threads location on the grid */
const unsigned int prev_row_base = (i)*w;
const unsigned int row_base = prev_row_base+w;
const unsigned int next_row_base = row_base+w;
const unsigned int cell_base = row_base+j+1;
```

Using these base variables, the index calculation of all 8 neighbors can be reduced to just addition, which is much cheaper than expressions such as `a[row*width + col];`

After computing the new value of each cell, the `mirror_kernel` is launched. This kernel has the simple task of duplicating the first and last column. The dilemma regarding this kernel is that it *could* have been merged with the update kernel. The trade-off is that if it was a part of the update kernel, most threads of that kernel would have been idle near the end. On the other hand, despite being cheap, the cost of launching a new kernel, tailor-suited for this task is not *free*. Nonetheless, for a clear comparison between the cost of launching new kernels or avoiding divergence as much as possible, we have tried both and they are compared in section 3. In the dedicated server for mirroring, the dimension of the launched grid is much smaller. In fact, only two columns of threads are deployed to do the copying in parallel.

Lastly, one of the most challenging parts of the entire implementation is the reduction of the `maxdifff`. First, in the `update_kernel()`, each thread will write the calculated diff to a variable in the global memory.

```
double diff = fabs(v - v_old);
maxdiff[cell_base] = diff;
```

This matrix is then used in the third phase as the input to the `maxdiff_kernel()`. For the calculation of the actual maximum number in the matrix, we use a two phase reduction operation. First, each row is reduced to the first element of that row. Consequently, in the second phase, the first column is reduced to the first element of the matrix at index [0,0]. We have also implemented an optimized version of this operation that uses *shared memory*. In this version, first each thread in each block fetches and stores one element of the matrix. Next, the block is reduced horizontally and vertically and the result is stored in one extra row dedicated to this purpose in the matrix. Finally, the second kernel reduces the mentioned row and stores the global maxdiff at index [0,0]. These operations are depicted in Fig 1. Furthermore, the following snippet, as an example, shows the row/column wise reduction performed with collaborative shared memory. This outcome of this snippet is that maximum of each block being stored in the first cell of that particular block.

```
const unsigned int i = blockIdx.y * blockDim.y + threadIdx.y;
const unsigned int j = blockIdx.x * blockDim.x + threadIdx.x;
```

```
// 1-d index of cell
const unsigned int block_resp_index = blockDim.x*threadIdx.y + threadIdx.x;

/* Load the entire matrix in parallel and sync each thread */
extern __shared__ double shared_maxdiff[];
shared_maxdiff[block_resp_index] = maxdiff[_index(i,j,w)];
__syncthreads();

/* Reduce each row of the block horizontally */
for (unsigned int s=(blockDim.x)/2; s>0; s>>=1) {
   if (threadIdx.x < s) {
        if (shared_maxdiff[i,j+s] > shared_maxdiff[i,j]) {
            shared_maxdiff[i,j] = shared_maxdiff[i,j+s];
        }
    }
    __syncthreads();
}

/* Reduce each column vertically */
if ( threadIdx.x == 0 ) {
    for (unsigned int s=(blockDim.y)/2; s>0; s>>=1) {
        if (threadIdx.y < s) {
            if (shared_maxdiff[i+s,j] > shared_maxdiff[i,j]) {
                shared_maxdiff[i,j] = shared_maxdiff[i+s,j];
            }
        }
        __syncthreads();
    }
}

/* one thread writes the result back. They are stored in order in the first row */
if ( threadIdx.x == 0 && threadIdx.y == 0 ) {
    maxdiff[w+1+(blockIdx.x)+(gridDim.x*blockIdx.y)] = shared_maxdiff[0];
}
```

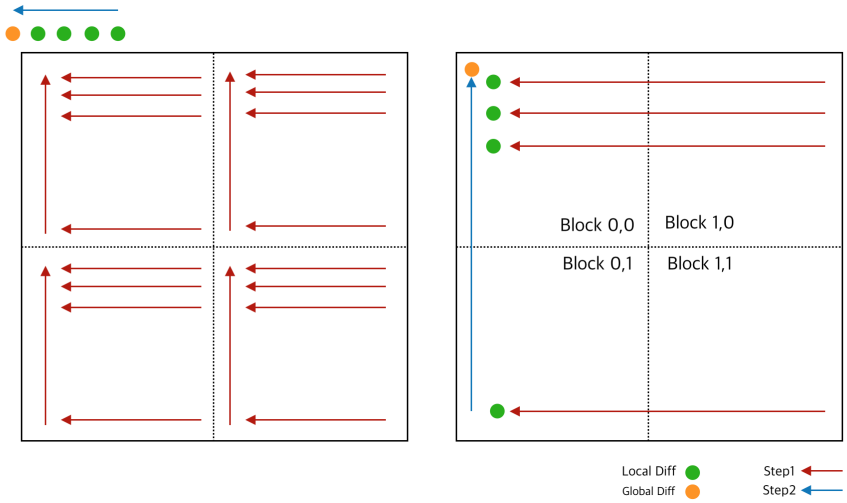

Figure 1: Demonstration of the maxdiff reduction in (left) shared memory and (right) normal mode.

Regarding the details of the reduction, we have used the approach suggested by Nvidia. This approach is limited to row/column-wise reduction. Essentially, in this iterative reduction each thread compares the value associated with its own index, `j` with a value `step` columns away, namely `j + step`. After each iteration `step` is halved. Consequently, in the last iteration, only the first thread of the row is active and it compares first and second column. Interested reader can refer to [2] for more information.

2.2 Histogram

A histogram of a grayscale image is essentially counting the number of occurrences of each pixel value in the image. Hence, for this assignment where each image is defined as a `NxM` matrix of values in range `[0, 255]`, the result of the computation is a vector of 255 elements where element `i` indicates how many times the pixel value `i` exists in the image. For this assignment we have implemented two different versions (kernels) of computing a histogram with CUDA.

- A *simple version* where each thread represents a pixel in the image, reads the value out of it and writes it to global memory.
- An *optimized version* where each thread represents a pixel in the image and each block creates a local histogram. In a next step all the local histograms get reduced to one final histogram in parallel.

For both versions, first, an image with random color distribution is created in the specified size. If the parameter `-b` is used, a black image – worst case – is created. Then, the histogram is

calculated sequentially. This enables us to verify the correctness of the parallel execution, which is done in the end of the application.

The kernel preparation and invocation in both implementations is very similar. Therefore we will describe the process once for the simple implementation and later on reference parts when describing the optimized version. First, the number of `blocks` to be run on the device is calculated based on the `threadBlockSize`. Then, the image and histogram are allocated on the device using `cudaMalloc`. The image is copied to the device and the histogram is set to `0` using `cudaMemcpy` and `cudaMemset`. Then the kernel can be executed and in the end the histogram is copied back to the host using `cudaMalloc`. Both kernels run with a block size of 1024 threads per block, which is the maximum supported of the hardware we tested on. On a GTX480 there can be `65535 blocks` launched, which limits the maximum image size to $1024 \times 65535 = 67107840$ pixels since each pixel is represented by a thread.

The following code snippet shows the simple kernel. Each thread gets its `tid` in the beginning and atomically adds its corresponding pixel value to the histogram in global memory. This is only done if `tid` is smaller than the image size to prevent additional threads in the last block from accessing uninitialized memory.

```
__global__ void histogramKernelSimple(unsigned char* __restrict__ image, long img_size,
    unsigned int* __restrict__ histogram) {
    int tid = threadIdx.x + blockDim.x * blockIdx.x;
    if(tid < img_size) {
        atomicAdd(&histogram[image[tid]], 1);
    }
}
```

The optimized version also calculates the number of blocks before allocating the image and a local histogram for every block. Then, the image is copied to the device using `cudaMemcpy`. The histogram kernel is executed, followed by multiple invocations of the reduce kernel. The kernel is recursively executed halving the number of blocks until only one block is left. In the end, the final histogram – the first in the memory structure of previously local histograms – is copied back to the host using `cudaMalloc`.

```
int reduce_blocks = (int) ceil(blocks / 2.0);
if(reduce_blocks % 2 != 0) {
   reduce_blocks++;
}
int last_reduction = blocks;

while(reduce_blocks >= 1) {
   reduceKernel<<<reduce_blocks, hist_size>>>(deviceHistos, reduce_blocks,
       last_reduction);
        cudaDeviceSynchronize();
```

```
   last_reduction = reduce_blocks;
    reduce_blocks = (int) ceil(reduce_blocks / 2.0);
   if(reduce_blocks % 2 != 0 && reduce_blocks != 1) {
      reduce_blocks++;
   }
}
```

The following code depicts the optimized histogram kernel. First, a histogram in shared memory is allocated and the first `hist_size` threads initialize it with `0`. With `__syncthreads()` it is made sure that all threads in the block see the changes to memory. Then, the thread representing a pixel in the image atomically adds its corresponding pixel value to the local histogram. Following a synchronization of the threads in the block the first `hist_size` threads copy the values of the local histogram to the global memory to make it persistent over multiple kernel invocations.

```
__global__ void histogramKernel(unsigned char* __restrict__ image, long img_size,
    unsigned int* __restrict__ histos) {
    __shared__ unsigned int shared_histo[hist_size];
    unsigned int tid = threadIdx.x;
    unsigned int i = tid + blockDim.x * blockIdx.x;

    // initialize shared memory to 0 in parallel (256 first threads in each block)
    if(tid < hist_size) {
        shared_histo[tid] = 0;
    }
    // make sure, that all writes to shared memory are finished
    __syncthreads();
    if(i < img_size) {
        atomicAdd(&shared_histo[image[i]], 1);
    }
    // make sure, that all writes to shared memory are finished
    __syncthreads();
    // write histogram of block back to global memory
    if(tid < hist_size) {
        // advance pointer to histograms to block specific one
        histos += blockIdx.x * hist_size;
        histos[tid] = shared_histo[tid];
    }
}
```

As described previously, the reduce kernel is called recursively with halving the number of blocks until only one block is left. Within the kernel basically each block reduces two local histograms to one on the position of the current block. If the number of blocks is not dividable by two, more blocks than to reduce histograms exist. Thus, it can happen that the last blocks practically do nothing.

```
__global__ void reduceKernel(unsigned int* __restrict__ histos, const int reduce_blocks,
    const int last_reduction_blocks) {
    unsigned int tid = threadIdx.x;
    if((blockIdx.x + reduce_blocks) < last_reduction_blocks) {
        // get current position
        int thread = blockIdx.x * hist_size + tid;
        // get position from block to reduce
        int thread_next = (blockIdx.x + reduce_blocks) * hist_size + tid;
        histos[thread] += histos[thread_next];
    }
}
```

2.3 Smoothing filter

The implementation of the smoothing filter consists of three consecutive steps. First, we have implemented a naive version that works for every possible input. Next, we have enhanced this version by using constant memory for filter values. Finally, *tiling* is introduced to further improve the performance of the filtering. We will now briefly explain the major keys to implement each of the steps mentioned above.

The initial version of the filtering can be implemented straightforward. Similar to other implementations of CUDA, the outer loop of the cell update is eliminated and a kernel is launched to perform it. Essentially, each thread will update one cell and this eliminates the loop to iterate through each pixel. Each thread only has to iterate through each filter value and perform the modification. The following gist depicts this process.

```
const unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;
const unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
float sum = 0.0;

// global index
const unsigned int based_index = y * image_width + x;

/*Ensures the kernel works for all inputs */
if (x >= image_width || y >= image_height ) { return; }

/* Iterate each filter value */
for (int i=0; i < filter_height; i++) {
    for (int j=0; j < filter_width; j++) {
        sum += input[(y+i)*input_width+x+j] * filter[i*filter_width+j];
    }
}
output[based_index] = sum / filter_sum;
```

One can easily notice from the above snippet that *all* threads use the `filter` variable, which is stored in the global memory at the moment. In our case, the size of the filter is relatively small `5x5` and the values remain constant in the entire execution. This makes it ideal to be stored in the constant memory. In order to do this, the variable should be defined as `__constant__` in the global scope. Next, a CUDA function should be called to copy the values to the constant memory. The values stored in constant memory are accessible from all threads and readily cached, which significantly reduces their access time. The following snippet demonstrates how to enable constant memory. Henceforth, in the kernel code, instead of `filter[]` and `filter_sum`, `dc_filter[]` and `dc_filter_sum`, can be used.

```
/* In global scope */
__constant__ float dc_filter[filter_width*filter_height];
__constant__ float dc_filter_sum = 0.0;

/* In main function */
cudaMemcpyToSymbol(dc_filter_sum, &filter_sum, sizeof(float));
cudaMemcpyToSymbol(dc_filter, filter, filter_width*filter_height*sizeof(float));
```

To further optimize the code, accesses to global memory need to be reduced. Right now every thread accesses global memory `filter_height x filter_width = 25` times. This can be done with a technique called *tiling*, which is shown in figure 2. Tiling describes the process of splitting global data to smaller local chunks. In essence, this means in our case, copying all data needed by a block from global to shared memory before starting the computation on the data in shared memory.

Figure 2 shows an image with size `10x10`, filter size `5x5` – therefore input size of `14x14` – and a block size of `3x3`. In row one of the grid there are two blocks shown. For block `B(0|0)` there is also the data needed for shared memory shown. The gray marked threads in `B(0|3)` are not executing, since they are out of the boundaries of the image. As can be seen every block needs data mapping to all threads plus `border_width = 4` columns on the right and `border_height = 4` rows at the bottom. With this data every read operation in the loop can be performed from shared memory.

The process of copying data from global memory to shared memory can be done in three phases, which can be executed in parallel, due to the fact that there are no data dependencies.

1. Each thread copies one value from global memory to shared memory at its corresponding shared memory place
2. Select threads to copy the last 4 values in each row
3. Select threads to copy the last 4 rows

Implementing this mapping from global to shared memory for many threads in parallel was quite a challenge and in the end it was not working as expected. Therefore we implemented a naive

version, where only the first thread copies all the values to shared memory. This obviously is a waste of resources and does not yield the expected speedup from using shared memory, because the time copying to shared memory is bigger than the gain by using shared memory. This can also be seen in our test results in section 3.4.

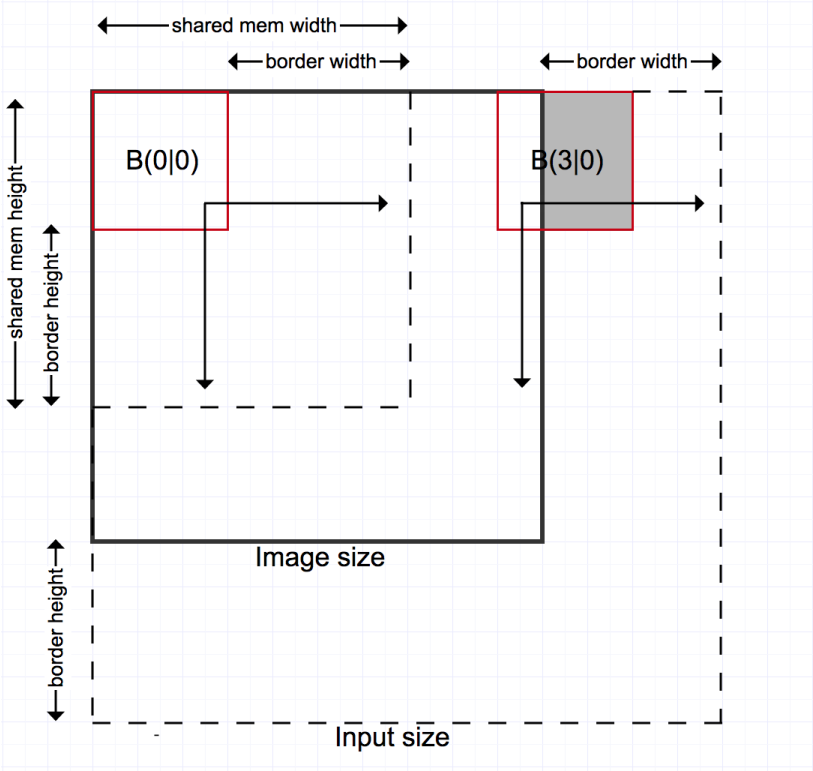

Figure 2: *Tiling*: splitting global memory in smaller chunks and making it accessible in shared memory for every block

3 Evaluation

This section first describes the experimental setup, the procedure of taking measurements and our expectations. Then, the results using the presented setup are displayed and analyzed.

3.1 Experimental setup and expectations

All the experiments were conducted on the DAS4 VU cluster. Each used node consists of a dual quad-core, 2x hyperthreading processor with 2.4GHz, 24 GB memory, a NVIDIA GTX480 GPU with 1.5GB onboard memory and 2*1TB HDDs interconnected with Gigabit Ethernet and InfiniBand. Measured was only the time of the actual computation without eventual initial time for

reading and preprocessing. For experiments on the GPU the time copying data to and back from the device is included in the total time. Furthermore, for the heat dissipation floating point operations per second (FLOPs) are calculated based on the measured time and the size of the problem. Measurement results marked as sequential, are truly conducted as sequential executions of the corresponding program, which means that there is no overhead for creating or managing threads. All applications were compiled using `GCC 4.8.5`, `CUDA Toolkit 8.0.61` and the optimization flag `-O3`.

Based on these hardware specifications of the DAS our expectations for the different applications are as follows:

- **Heat Dissipation**: Unlike other parallelization methods, CUDA is a *massively parallel platform* that uses thousands of cores. Albeit, expecting an ideal speedup is always out of reach because of various application specifications (e.g. synchronization costs). Based on this knowledge alone, our biggest expectation would be for the GPU to outperform other CPU-based approaches particularly on *large inputs*. Furthermore, we expect the performance gap w.r.t. the sequential implementation to be more significant. Furthermore, based on numerous best-practice articles, we expect using shared memory to be beneficial, up to an unknown extent.
- **Histogram**: While the histogram application itself can be deemed as an *embarrassingly parallel task*, the high data dependency and necessary synchronization when accessing the 256 bins, can be a performance kill. Hence, we form the following set of hypothesis:
  - In general we expect both CUDA versions to be faster than the sequential implementation even when including the time to copy to the device into the total time. It remains to be seen how a worst case image (black image) will effect speedup times. Since there is a lot of synchronization needed we expect run times to be slower than sequential.
  - For the *simple version* we expect it to be faster than sequential and slower than the optimized version. Especially in the worst case scenario we expect to see huge slow-downs due to necessary synchronization.
  - We expect the *optimized version* to outperform the simple version. In the worst case scenario, we expect the optimization using shared memory and local histogram to pay off and don't show slow-downs as big as for the simple version.
- **Smoothing filter**: We expect the constant memory usage to deliver a noticeable improvement to the speedup of the execution. Regarding the shared memory we expect the accesses within the loop to be much faster compared to global memory. The question here is whether copying the values to shared memory is fast enough to be worth doing. For our naive shared memory implementation we expect the time to copy values to shared memory to be too big and therefore slowing the execution down. Copying in parallel should yield

improvements, but unfortunately we were not able to test our copying in parallel version due to unexpected results.

3.2 Results: Heat Dissipation

Table 1 demonstrates the most important results obtained from the heat dissipation implementation with CUDA. First, we need to explain the details of this extensive table and then conclude from it. All tests are executed with the following parameters:

- 200 iterations
- `0.0001` threshold to ensure full execution

The three main header-columns of the table are based on the description given in section 2.1. To recap, they are as follows:

- *Global Mem / Sep. Mirror*: Maxdiff reduction is based on global memory and a separate kernel is used to mirror border columns.
- *Global Mem / Merged. Mirror*: Maxdiff reduction is based on global memory and border column mirroring is merged into the update kernel.
- *Shared Mem / Merged. Mirror*: Maxdiff reduction is based on shared memory and border column mirroring is merged into the update kernel.

The Speedup columns associated with each header-column, denoted by *SU-** are relative speedups based on previous implementations. As an example, the *SU-Omp* column is calculated as $\frac{Time_{OMP}}{Time_{CUDA}}$ for that particular size. For the two parallel versions from previous parts, **8** threads are used in all benchmarks. This is due to the fact that in previous conclusion, 8 was identified as the most optimal thread number based on the hardware available.

Note that for the 3rd item, a shared memory segment with the total aggregate size same as the input/output matrix must be created. This bottleneck becomes a runtime `out-of-memory` allocation error with big inputs. Hence, the shared memory version is only tested with values up to 1024x1024. Using 8196x8196 would cause the main memory of the device to suffocate and leads to the same error.

Furthermore, all kernels could also work with sizes such as 1000x1000, 2000x2000, etc. Albeit, we have mentioned in section 2.1 that the block dimension is fixed to 32 and enough blocks are created to fill the entire input. We have chosen sizes that are all powers of two to not waste any threads in this process.

Observing the values of the table, to following points can be concluded:

- As mentioned before, since CUDA is a *massively parallel* platform, the real speedup can only be observed in inputs that are large enough. This is indeed to the fact that in small inputs the cost of data transfer and synchronization is more significant than the speedup gained. This can be observed in all 3 sections of the table where the FLOP count increases substantially as the input size grows.
- Same argument is valid for comparison of CUDA as opposed to other parallelization approaches such as PThreads. We can see that the `SU-Omp/PTh` columns are smaller than **1** in small inputs (less than 1024x1024). This means that the core/cpu level parallelization approach was faster than CUDA for these sizes. As opposed to that we see the exact opposite trend for large inputs. For example, in 4096x4096 the CUDA implementation was almost twice as fast as both PThreads and OMP.
- Regarding the exact ranking of the 3 main parallelization approaches, namely PThreads, OMP and CUDA, we can see that for large inputs CUDA is the most dominant with speedups around 10 relative to sequential code, a speedup number that was never feasible to achieve with OMP/PThreads. Among the two, we can see that our OMP implementation was slightly better than PThreads and can be named as the main competent of our CUDA implementation.
- Shared memory brings a small degree of speedup. Nonetheless, the memory bottleneck didn't allow us to further test this. Aside from that, we can see that the speedup gain of shared memory is noticeable in only small inputs. This is due to the fact that shared memory approaches require another level of copying, from device main memory to block shared memory.
- We can confidently state that by splitting the kernel into multiple ones and placing the iteration loop in the host code, we have gained the benefit of having implicit synchronization. Nonetheless, we have seen that an excessive use of kernels can have a negative effect on the performance. This can be clearly seen by comparing the fist and the second header column of the table. Simply said, to conclude: *Launching a kernel is cheap, but not free.*

Lastly, we will emphasize on some of the performance blockers of this problem that kept us from reaching the maximum possible FLOPS count.

- The GPU used for this experiment has relatively small amount of shared and global memory. This kept us from testing anything larger than 1024x1024 with shared memory and 4096x4096 with global memory. Keeping that in mind, we can clearly observe a *weak scaling* pattern in the table. That is, the FLOP count increases significantly as the input size grows significantly. We can argue that we have **not** exploited the full computation power of the cheap because of memory constrains.
- Synchronization of all threads at the end step of the each iteration is expected to be a relatively important performance bottleneck.

Table 1: Overview of the Heat Dissipation results

	Global Mem / Sep. Mirror					*Global Mem / merged Mirror*					*Shared Mem / merged Mirror*				
Size	***Time(S)***	***GFLOPS***	***SU-Seq***	***SU-Omp***	***SU-Pth***	***Time(S)***	***GFLOPS***	***SU-Seq***	***SU-Omp***	***SU-Pth***	***Time(S)***	***GFLOPS***	***SU-Seq***	***SU-Omp***	***SU-Pth***
256x256	0.08	1.87	0.83	0.13	0.38	0.08	1.91	0.85	0.13	0.39	0.08	2	0.89	0.14	0.40
512x512	0.11	5.64	2.78	0.47	0.68	0.12	5.37	2.65	0.45	0.64	0.11	5.67	2.79	0.47	0.68
1024x1024	0.22	11.37	5.91	0.94	1.33	0.22	15.43	5.99	0.96	1.35	0.22	11.8	6.13	0.98	1.38
2048x2048	0.64	16.18	8.50	1.80	2.06	0.61	16.62	8.87	1.88	2.15	-	-	-	-	-
4096x4096	2.32	17.51	8.97	1.82	2.02	2.22	18.28	9.37	1.90	2.11	-	-	-	-	-

3.3 Results: Histogram

Tables 2 and 3 show the results for experiments with the different histogram implementations. The column *implementation* describes the program version which was tested. *GPU simple* and *GPU optimized* refer to the *simple* and *optimized version*, respectively, described in section 2.2. *par CPU* refers to the *image-size-parallel* implementation from the previous assignment with PThreads. It was run with 32 threads since tests in the previous assignment yielded the best outcome on the DAS4 for this implementation. Each thread gets a partition of the image and creates a local histogram. All of these local histograms get merged to one final histogram in the end. This implementation does not need any synchronization.

In table 2 the results with a random image pattern are shown. It can be seen that par CPU generally outperforms all the other implementations. Furthermore, the GPU simple version is slightly faster than the sequential implementation but slower than the optimized CUDA version. The memory time for both GPU versions are basically the same, since the exact same amount of data is copied to and from the device. So only the kernel execution times differ. For the optimized CUDA version the time to copy data to the device takes longer than the actual execution of the kernel. This result yielding speedup compared to the sequential version is as expected, but the speedup is unexpectedly low. Both GPU versions are only marginal faster than the sequential implementation and – for bigger inputs a lot – slower than the parallel CPU version with PThreads.

Table 3 presents the results with a black image, which is the worst case scenario for computing a histogram. Every access, in this case for every pixel, to a shared resource needs to be synchronized. Overall, it can be observed that run times are slower compared to the random image pattern. In detail, the GPU simple and GPU optimized yield negative speedups compared to the sequential implementation. This is due to high synchronization. It also can be observed that the simple version is much slower than the optimized version, which is caused again by too many concurrent memory accesses. In the simple version all launched threads compete for one histogram and in the worst case only for one address to write to, which leads to congestion and long run times. The optimized version does better since there are only 1024 threads in a block which compete to concurrent access to shared memory to write to the local histogram of the block. Furthermore, the par CPU undergoes almost no slowdown since there is no synchronization at all needed. Thus, this worst case does not affect the execution in that way but maybe through different caching behaviour. Again, the memory time for both versions are the same and equal to those of the random image since the same data needs to be copied to the device. We did expect a slowdown for the GPU versions for the worst case scenario, which is clearly shown in the result. It is also obvious how the

Table 2: Histogram results with random image pattern

Size	Implementation	Time	Memory Time	Kernel Time	Speedup
1000x1000	seq	0.00193	-	-	-
1000x1000	par CPU	0.00126	-	-	1.59
1000x1000	GPU simple	0.00158	0.00054	0.00104	1.22
1000x1000	GPU optimized	0.00123	0.00050	0.00073	1.57
4000x4000	seq	0.03118	-	-	-
4000x4000	par CPU	0.00806	-	-	3.87
4000x4000	GPU simple	0.01919	0.00613	0.01305	1.63
4000x4000	GPU optimized	0.01120	0.00598	0.00522	2.78
8000x8000	seq	0.12492	-	-	-
8000x8000	par CPU	0.03065	-	-	4.08
8000x8000	GPU simple	0.07487	0.02317	0.05171	1.67
8000x8000	GPU optimized	0.04255	0.02306	0.01949	2.94

Table 3: Histogram results with worst-case image pattern (black image)

Size	Implementation	Time	Memory Time	Kernel Time	Speedup
1000x1000	seq	0.00527	-	-	-
1000x1000	par CPU	0.00119	-	-	4.43
1000x1000	GPU simple	0.01912	0.00055	0.01857	0.28
1000x1000	GPU optimized	0.00734	0.00052	0.00682	0.72
4000x4000	seq	0.06774	-	-	-
4000x4000	par CPU	0.00851	-	-	7.96
4000x4000	GPU simple	0.29965	0.00610	0.29354	0.23
4000x4000	GPU optimized	0.10770	0.00601	0.10169	0.63
8000x8000	seq	0.26828	-	-	-
8000x8000	par CPU	0.03512	-	-	7.64
8000x8000	GPU simple	1.19672	0.02310	1.17362	0.22
8000x8000	GPU optimized	0.42859	0.02330	0.40529	0.63

optimization with the local memory improves the run time especially in the worst case.

3.4 Results: Smoothing filter

To test the smoothing implementation, we have chosen three distinct images sizes. All experiments are performed with the block size of *16x16*. We have empirically, through trial and error, realized that this block size is slightly faster for this application.

Table 4 demonstrates the results obtained from the smoothing experiment, comparing the first two versions. Despite the differences being small, the following can be concluded from the table: For the constant memory, the execution time is smaller comparing to that of the naive version. As

Table 4: Results of the smoothing experiments with the naive and constant memory version.

	Naive				*Constant Memory*			
Size	***Time(Seq)***	***Time(GPU-Exec)***	***Time(GPU-Mem)***	***Speedup***	***Time(Seq)***	***Time(GPU-Exec)***	***Time(GPU-Mem)***	***Speedup***
5000x5000	1.015	0.016	0.095	9.197	1.015	0.013	0.094	9.414
10000x10000	3.867	0.058	0.373	8.965	3.867	0.053	0.374	9.057
20000x5000	3.879	0.059	0.375	8.940	3.879	0.053	0.385	8.851

Table 5: Results of the smoothing experiments with the naive shared memory version.

Size	***Time(Seq)***	***Time(GPU-Exec)***	***Time(GPU-Mem)***	***Speedup***
5000x5000	1.015	0.160	0.093	4.018
10000x10000	3.867	0.636	0.368	3.850
20000x5000	3.879	0.636	0.368	3.863

opposed to that, the memory time of the constant version is slightly bigger. Nonetheless, we can see that the overall speedup of the constant memory version is bigger than the naive version. The third row is a counter example of this conclusion because of having a less optimized memory layout. Having narrower columns with more number of rows reduces the efficiency of cache performance. For this reason, despite having the same total number of elements, `20000x5000` is slightly slower.

Table 5 shows the results of measurements of the naive shared memory version, where only the first thread of each block copies all the needed values from global memory to shared memory. As expected the speedup is way worse in all three runs compared to the naive and the constant memory version. The time for the first thread to copy all the values to shared memory is too big compared to the speedup gained by using the shared memory. If this is done in parallel we are very confident that there is speedup to gain compared to accessing global memory over and over again.

4 Conclusion

We have seen that programming a GPU with CUDA and its programming model (SIMT) is similar to SIMD programming, at least in terms of thinking and unrolling loops. Especially this thinking is quite different from usual parallel programming and needs some time for the programmer to get used to. The model CUDA uses is quite close to the hardware, so a programmer has to write low-level code and be aware of the hardware opportunities and limitations to write efficient code.

Some other conclusions obtained from the tasks of this report can be enumerated as:

- While getting speedups better than CPU-based parallelization is quite easy to do with CUDA, utilizing the maximum capacity of the hardware is very difficult and requires programmer to take multiple aspects into account. Some of these aspects being: grid mapping, using constant and shared memory, having minimum thread divergence.
- CUDA is a *massively parallel platform*, this makes it suitable for applications that are large enough for this platform. In other words, it is likely to benefit more from the platform in

very large scale problems.

References

[1] CUDA, Nvidia Developer Zone

[2] Reduction in CUDA, Nvidia Developer Zone

[3] Full experiment table, Google Docs