Exploiting Computer Architecture Characteristics

Jerry Schonenberg (s2041022) and Dennis Buurman (s2027100)

December 21, 2018

Introduction

For the third and final assignment of Computer Architecture, we had to optimize a given program. This program can perform four operations on a PNG file. The assignment consists of four parts:

- CPU Caching effects
- SIMD
- Multi core
- GPU

For each of the tasks and sub-tasks, the file name will be given in the report together with the command to compile the file. We used perf to test the program. The perf command used is mentioned in every section. The repetitions of each program is set to 15 with `-r 15` in the perf command.
The baseline programs can be found in `ops-baseline.c` and `ops-cuda.cu`. These can be compiled with the command `make base`.
The command `make` will compile all the available programs.

The images `test4096.png`, `test512.png` and `bricks512.png` used for testing can be found on Blackboard [2].
To check whether the baseline output matches out output image, we used a compare utility which can be found on Blackboard [3].

The computer which is used for the test results is a Dell Precision T1650 located in Snellius room 302/304. The specifications of this machine are as follows:

- CPU: Intel core i7-3770 (@3.40GHz, 4 cores, 8 threads, 8MB cache: L1 Cache: 4 x 32 KB instruction caches, 4 x 32 KB data caches; L2 Cache: 4 x 256KB; L3 Cache: 8MB)
- RAM: 16GB DDR3
- GPU: Quadro K2000
- Machine ID: 0009763 from 304

1 CPU Caching effects

1.1 Transpose

File name: `ops-version1.c`
Command: `make version1`
PERF: `perf stat -D 500 -d -d -r 15 -e cache-misses:u,cycles:u ./imglab-version1 transpose test4096.png`

Function analysis of `run_op_transpose`:
The loops access the memory in the wrong order (width to height). This results in additional cache-misses. The pixels of `src` (source image) are processed vertically. This is not optimal for caching, this is because the CPU will prematurely store the pixels after the current one is in it's cache. But these pixels are the ones in the same row as the current pixel. So each time, there will be a cache-miss because the next pixel after the current one is in the row beneath the current one and thus not stored in the cache.
However, with `dst` (destination image) they are processed horizontally. So in the case of the `dst`, there will not be

many cache-misses in comparison with `src`.

Optimization's:
The first optimization in transpose was a loop interchange. This increased the speed of transpose a bit. That is because the program now goes through `src` horizontally and vertically through `dst` instead of the other way around. The results of this optimization are in table 1.

	`ops-baseline.c`	`ops-version1.c`
Time (sec):	0,955123	0,747119
Speedup:	1,0	1,278408
Cache-misses:	66.382.104	60.451.537
LLC-loads:	83.843.272	6.756.350
Cycles:	3.531.144.006	2.646.915.537

Table 1: Results of the first optimization with image `test4096.png`.

From table 1, we can conclude that interchanging the loops decreases the amount of LLC-loads drastically. This also causes the amount of cycles to decrease. All in all, the time it takes to process the image decreases by several tenths for `test4096.png`. The amount of cache-misses is also decreased by millions, because we read the `src` in the right order.

The second and final optimization is adding a third loop as the innermost loop. For this optimization, we assume that the dimensions of the input image are a multiple of 64.
This third loop ensures that both `src` and `dst` are processed partially horizontal and partially vertical. Performance improved significantly with this optimization. Cache-misses, LLC-loads and cycles were greatly reduced. The results of this optimization are in table 2.

	`ops-baseline.c`	`ops-version1.c`
Time (sec):	0,955123	0,228949
Speedup:	1,0	4,171771
Cache-misses:	66.382.104	12.202.280
LLC-loads:	83.843.272	40.761.516
Cycles:	3.531.144.006	2.471.912.879

Table 2: Results of the third optimization with image `test4096.png`.

From table 2, we can conclude that our optimization works. The amount of cache-misses, LLC-loads and cycles has decreased massively in comparison to `ops-baseline.c` and also in comparison to the first optimization.

Correctness:
According to the compare utility the resulting image of our optimized program is identical to the original programs output image. We tested this with `bricks512.png`, `test512.png` and `test4096.png`.

The picture `bricks512.png` was inconsistent with file-load times because it took very little time. Therefore we only checked the speedup while using `bricks512.png`. This is also the case for all other tests done with `bricks512.png`
Final speedup: factor 4,17 for `test4096.png` and factor 1,59 for `bricks512.png`.

1.2 Transgram

Loop blocking:
File name: `ops-version2.c`
Command: `make version2`
PERF: `perf stat -D 500 -d -d -r 15 -e cache-misses:u,cycles:u ./imglab-version2 transgram test4096.png`
Loop merging:
File name: `ops-version3.c`
Command: `make version3`
PERF: `perf stat -D 500 -d -d -r 15 -e cache-misses:u,cycles:u ./imglab-version3 transgram test4096.png`

Function analysis of `run_op_transgram`:
Once again the loops access the memory in the wrong order (width to height). The remainder of `run_op_transgram` seems fine as to caching effects. To further speed up transgram, we have used loop blocking and loop merging (which also uses loop blocking) in addition to the optimization's done in `ops-version1.c`.

Optimization's:
The first optimization was loop blocking on `op_histogram` and `op_transpose`. We added a constant for the block size. The loop blocking creates blocks that are processed by transpose and histogram. By creating smaller blocks the amount of cache-misses is greatly reduced. This is because a block with the right size fits in the cache. To implement this we added two for-loops to `run_op_transgram`. The loops are incremented with the block size. Within each iteration the functions `op_transpose` and `op_histogram` process the block of the image they received from `run_op_transgram`. For histogram, we had to move the loop which zeros out the `bins` array from `op_histogram` to `run_op_transgram`. Otherwise `bins` will be zeroed out with every block. `run_op_histogram` and `run_op_transpose` needed the changes done to `run_op_transgram` to work after implementing loop blocking. We chose a block size based on the L1 cache, which is 32KB or 256.000 bits. `image_get_pixel(image, x, y)` is of type long unsigned int, which is 4 bytes or 32 bits. With the calculation: $32 * (blocksize)^2$ the cached bits can be calculated. We also assumed that the dimensions of the image are a multiple of 64. Here is a table of the calculations with a given block size:

block size	cached bits
8	2.048
16	8.196
32	32.768
64	131.072
128	524.288

Table 3: Block size calculations.

As seen in the table, a block size of 128 exceeds the available L1 cache. Therefore a block size of 64 is theoretically optimal. We tested our loop blocking with block size 8 to 64, the results are in the table below. The tests were conducted on `run_op_transgram`.

	ops-baseline.c	ops-version2.c			
Blocksize:	-	8	16	32	64
Time (sec):	8,138001	1,295078	0,884682	0,885414	0,941616
Speedup:	1,0	6,283792	9,198786	9,191181	8,642589
Cache-misses:	124.670.411	24.875.757	18.675.091	12.583.756	16.213.421
LLC-loads:	174.279.998	14.627.794	12.849.824	15.998.869	47.920.017
Cycles:	32.624.449.403	4.847.568.273	3.241.646.513	3.151.014.848	3.362.274.523

Table 4: Results of the optimization with different block sizes with image `test4096.png`.

From table 4 we can conclude that block size 16 and 32 give the fastest results for test4096.png. We picked block size 32 by weighing up the LLC-loads, cache-misses and cycles of block size 16 and 32. As seen in table 4 the LLC-loads of block size 32 are higher, but the cache-misses and cycles are lower. The difference between the cache-misses is higher so we chose block size 32.

Correctness:
The images produced by the baseline and our program remain identical. All values of histogram are also identical.

From the table above we can conclude that a block size of 32 results in the best performance.
Final speedup: factor 9,19 for `test4096.png` and factor 1,65 for `bricks512.png`.

With `ops-version3.c` we added loop merging to `run_op_transgram`. The operations of `op_transpose` and `op_histogram` are merged and inserted into `run_op_transgram`.

	`ops-baseline.c`	`ops-version3.c`			
Blocksize:	-	8	16	32	64
Time (sec):	8,138001	0,968640	0,937517	0,850876	0,857309
Speedup:	1,0	8,401471	8,680376	9,564262	9,492494
Cache-misses:	124.670.411	18.958.233	12.433.183	12.809.911	11.595.400
LLC-loads:	174.279.998	7.935.684	5.426.591	6.516.121	8.072.358
Cycles:	32.624.449.403	3.613.195.871	3.507.578.466	3.180.961.911	3.130.405.935

Table 5: Results of the optimization with different block sizes with image `test4096.png`.

This time the fastest results come from block size 32 and 64. There is a difference between the two block sizes. The LLC-loads of 32 are significantly lower, but the cache-misses and cycles are higher. Because the difference is minor we still pick block size 32 because of the speed.

When we compare loop blocking and loop merging, loop merging is definitely faster and also has a lot less LLC-loads in comparison to loop blocking. This is because with loop merging, the pixel gets reused and there are a lot less jumps.

Correctness:
The produced images and histograms still match with the baseline.

From the table above we can conclude that a block size of 32 results in the best performance.
Final speedup: factor 9,56 for `test4096.png` and factor 1,43 for `bricks512.png`.

2 SIMD

For SIMD to work `#include <immintrin.h>`, `<mmintrin.h>`, `<xmmintrin.h>`, `<emmintrin.h>` are needed.
We used the Intel intrinsics website [1] to program in SIMD.

2.1 Histogram

File name: `ops-version4.c`
Command: `make version4`
PERF: `perf stat -D 500 -d -d -r 15 -e cache-misses:u,cycles:u ./imglab-version4 histogram test4096.png`

SIMD programming:
We implemented histogram with SIMD using `__m128` single-precision floating-point vectors. We used four vectors to work on four pixels at the same time. For each value of the RGBA-values we had one vector. Furthermore, we translated each line of code from the baseline into SIMD-code. `op_histogram` and `compute_intensity` are fully translated into SIMD. We also had vectors for the multipliers of `compute_intensity` and `N_BINS - 1`.

Only `RGBA_unpack` is not translated into SIMD, because we did not know how to shift with `__m128`.

	`ops-baseline.c`	`ops-version4.c`
Time (sec):	7,198685	0,278652
Speedup:	1,0	25,833961
Cache-misses:	60.621.328	121.856
LLC-loads:	86.159.215	357.336
Cycles:	27.530.213.126	1.025.045.935

Table 6: Results of histogram implemented with SIMD with image `test4096.png`.

The SIMD implementation greatly sped up the process. The cache-misses, LLC-loads and cycles are reduced massively.

Correctness:
We checked if the histogram of our SIMD-implementation matched with the baseline. We tested this with `bricks512.png`, `test512.png` and `test4096.png`.

Final speedup: factor 25,83 for `test4096.png` and factor 4,18 for `bricks512.png`.

2.2 Selective Grayscale

File name: `ops-version5.c`
Command: `make version5`
PERF: `perf stat -D 500 -d -d -r 15 -e cache-misses:u,cycles:u ./imglab-version5 selgray test4096.png`

SIMD programming:
We implemented SIMD selgray with `__m128` single-precision floating-point vectors. We used four vectors to work on four pixels at a time. For each value of the RGBA-values we had one vector. `op_selective_greyscale`, `compute_hue` and `compute_intensity` are fully translated into SIMD. We also had vectors to store constants for `compute_hue` and `compute_intensity`. The only thing we did not translate to SIMD was the `RGBA_unpack`, `RGBA_pack` and `RGBA`, because we did not know how to shift `__m128`.
Furthermore, to implement the multiple if-statements in `compute_hue`, we computed all the possible results of the if-statements. Then, with a compare-function of SIMD and with the AND-operator of SIMD, we used the result of the compare-function to AND (logical) it with the computed result. If the compare-function returns false, then the result gets AND'ed with 0. Otherwise, it gets AND'ed with ones, so the outcome remains unchanged. When this is done for every condition of the if-statement, all the results get added to the end result.
At last, `compute_hue` and computing the absolute value of `hue - compute_hue` return an integer. In our SIMD implementation, we did not convert the floats to integers. Instead we floored the floats.

	`ops-baseline.c`	`ops-version5.c`
Time (sec):	22,318626	1,029898
Speedup:	1,0	21,670714
Cache-misses:	61.701.839	207.521
LLC-loads:	86.566.574	445.933
Cycles:	86.369.149.987	3.892.129.205

Table 7: Results of selgray implemented with SIMD with image `test4096.png`.

From the results of table 7 we can conclude that our SIMD implementation is a lot faster than the baseline. The cache-misses, LLC-loads, cycles are greatly reduced in comparison to the baseline.

Correctness:
We tested if the selgray images are equal to the selgray images of the baseline with the compare utility. We tested this with `bricks512.png`, `test512.png` and `test4096.png`.

Final speedup: factor 21,67 for `test4096.png` and factor 14,69 for `bricks512.png`.

2.3 SIMD worthwhile?

Yes, because the speedup factors are high. Beside the speedups, the amount of LLC-loads, cache-misses and cycles also decreased massively. It does take a while to code and debug in SIMD, but it is worth the time you invest in it. The readability of the code also gets worse in comparison with `ops-baseline.c`.

3 Multi-core

For OpenMP to work `#include <omp.h>` is needed. Use `omp_set_num_threads(n)` to set the number of threads. We used `#pragma omp parallel for` to activate multithreading on for-loops.

3.1 Histogram

File name: `ops-version6.c`
Command: `make version6`
PERF: `perf stat -D 500 -d -d -r 15 -e cache-misses:u,cycles:u ./imglab-version6 histogram test4096.png`

For the multi-core implementation on histogram we set the number of threads to 8 and only executed the nested for-loop that calls `op_histogram` in parallel. The increment of the `bins` array in `op_histogram` needs `#pragma omp atomic update`. This ensures that the increments of `bins` don't overlap. This way the right amount is incremented to each `bins[idx]`.

	`ops-baseline.c`	`ops-version6.c`
Time (sec):	7,198685	0,605640
Speedup:	1,0	11,886079
Cache-misses:	60.621.328	644.930
LLC-loads:	86.159.215	17.107.372
Cycles:	27.530.213.126	16.377.209.652

Table 8: Results of histogram implemented with OpenMP with image `test4096.png`.

The difference in speed is big. The multi-core variant does better in every category. Each core has its own L1 and L2 cache. The L3 cache is shared (Intel i7-3770). This explains why the cache-misses and LLC-loads are much lower with the multi-core variant.

Correctness:
We checked whether the values of the histogram were correct by comparing it to the baseline. And it all matched.

Final speedup: factor 11,88 for `test4096.png` and factor 1,25 for `bricks512.png`.

3.2 Selective Grayscale

File name: `ops-version7.c`
Command: `make version6`
PERF: `perf stat -D 500 -d -d -r 15 -e cache-misses:u,cycles:u ./imglab-version7 selgray test4096.png`

Just like histogram, the number of threads is set to 8. But this time, there is no shared memory where two threads can write to the same pixel. This is because each thread processes a different pixel. So because of that

there was no `#pragma omp atomic update` necessary. Furthermore, we placed the `#pragma omp parallel for` in the first for-loop of `op_selective_grayscale`, this is done to make the workload of each thread smaller.

	`ops-baseline.c`	`ops-version7.c`
Time (sec):	22,318626	3,119626
Speedup:	1,0	7,154263
Cache-misses:	61.701.839	59.776.593
LLC-loads:	86.566.574	85.742.924
Cycles:	86.369.149.987	90.277.163.713

Table 9: Results of selgray implemented with OpenMP with image `test4096.png`.

As with histogram, the difference in speed is big. However, this time the differences between LLC-loads, cycles and cache-misses is negligible. We suspect that this is because selgray uses the shared L3 cache for a large part.

Correctness:
To check whether the output of OpenMP is correct, we used the compare utility to compare it to the baseline output. And it all matched.

Final speedup: factor 7,15 for `test4096.png` and factor 5,67 for `bricks512.png`.

3.3 Best amount of cores

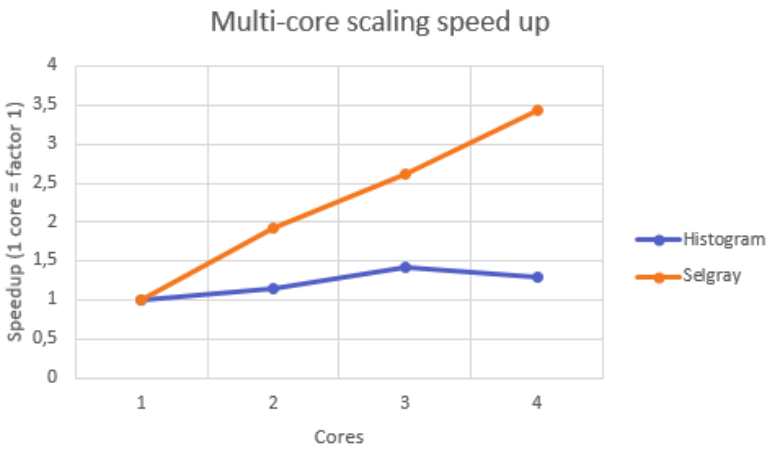
We investigated what the amount of cores is to get the optimal speed. See the tables and graph below for the results.

	`ops-baseline.c`	`ops-version6.c`			
Cores:	-	1	2	3	4
Time (sec):	7,198685	0,785204	0,677652	0,549908	0,605640
Speedup:	1,0	9,167916	10,62298	13,09070	11,886079
Cache-misses:	60.621.328	266.910	291.985	475.272	644.930
LLC-loads:	86.159.215	9.557.643	14.747.679	16.104.698	17.107.372
Cycles:	27.530.213.126	5.658.072.300	8.849.807.718	12.583.424.383	16.377.209.652

Table 10: Results when using different amount of cores with histogram and `test4096.png`.

	`ops-baseline.c`	`ops-version7.c`			
Cores:	-	1	2	3	4
Time (sec):	22,318626	10,703136	5,557125	4,0941148	3,119626
Speedup:	1,0	2,085241	4,016218	5,451392	7,154263
Cache-misses:	61.701.839	59.720.856	59.694.362	59.472.336	59.776.593
LLC-loads:	86.566.574	85.492.227	85.193.616	85.456.041	85.742.924
Cycles:	86.369.149.987	82.449.471.535	81.262.506.271	89.325.643.381	90.277.163.713

Table 11: Results when using different amount of cores with selgray and `test4096.png`.



As seen in the graph above, selgray scales much faster than histogram. An explanation for this is dependency. The `bins` array in histogram is accessed by multiple threads at a time. Because `bins[i]` can only be incremented by one thread at a time, there is a chance other threads have to wait to increment `bins[i]`. The operations of selgray are completely independent of each other (each thread a different pixel). This ensures better parallel performance and better scaling. With histogram the scaling even decreases with 4 cores compared to 3 cores. In conclusion, independency results in better multi-core performance because the threads can work in parallel without (potentially) having to wait on other threads.

4 GPU

4.1 Baseline

File name: `ops-cuda.cu`
Command: `make cuda`

We have written a baseline GPU implementation for transpose, histogram, transgram and selective grayscale. The CPU code of `ops-baseline.c` is directly translated into CUDA code which utilises the GPU of the machine.

4.1.1 Transpose

PERF: `perf stat -D 500 -d -d -r 15 -e cache-misses:u,cycles:u ./imglab-cuda transpose test4096.png`

In `run_op_transpose` a buffer is allocated to contain the initial image and result. This is allocated to `result_D` and `image_D` with `cudaMalloc`. After this is done, the block size and number of thread blocks is calculated. Then the input image is copied into `image_D` (host to device) and the `op_cuda_transpose` kernel is called with the calculated block size and number of thread blocks. `op_cuda_transpose` first checks if the indexes are in bounds. Then the transpose operation is executed. After all pixels are handled the transposed image is copied to `result->data` (device to host). At last the allocated memory is freed.

	`ops-baseline.c`	`ops-cuda.cu`
Time (sec):	0,955123	0,0380126
Speedup:	1,0	25,12648
Cache-misses:	66.382.104	372.966
LLC-loads:	83.843.272	2.602.278
Cycles:	3.531.144.006	132.639.390

Table 12: Results when using CUDA for transpose with image `test4096.png`.

Correctness:
The images produced by `ops-baseline.c` and `ops-cuda.cu` are identical according to the compare utility.

Final speedup: factor 25,12 for `test4096.png`.

4.1.2 Histogram

PERF: `perf stat -D 500 -d -d -r 15 -e cache-misses:u,cycles:u ./imglab-cuda histogram test4096.png`

We allocate a buffer for the initial image and the resulting `bins_D` array. Then we zero out the `bins` array (because of `n_repeat`). Then the block size and number of thread blocks is calculated. The input image is copied to `image_D` (host to device). Now the kernel `op_cuda_histogram` is called. This kernel checks if an index is out of bounds. The only difference (in code) with `op_histogram` from `ops-baseline.c` is the `atomicAdd` increment of `bins[idx]`. This is necessary because it is possible that multiple threads try to increment `bins[idx]` at the same time. `atomicAdd` prevents this from happening. `compute_intensity` is now a device. Functionality stays the same compared to the `compute_intensity` from `ops-baseline.c`. After all pixels are handled the resulting `bins_D` array is copied to the `bins` array (device to host). `image_D` and `bins_D` are freed and `bins` is printed.

	`ops-baseline.c`	`ops-cuda.cu`
Time (sec):	7,198685	0,063047
Speedup:	1,0	114,1796
Cache-misses:	60.621.328	109.365
LLC-loads:	86.159.215	1.963.863
Cycles:	27.530.213.126	91.429.388

Table 13: Results when using CUDA for histogram with image `test4096.png`.

Correctness:
Histogram with `ops-cuda.cu` and `ops-baseline.c` produce slightly different values. This is because of rounding errors with floating point operations. The CPU calculates the floats differently compared to the GPU. This sometimes leads to a different value. The `idx` of `bins[idx]` may differ by one due to the rounding errors. The differences are minor though. Only 4 values differ.

Final speedup: factor 114,17 for `test4096.png`.

4.1.3 Transgram

PERF: `perf stat -D 500 -d -d -r 15 -e cache-misses:u,cycles:u ./imglab-cuda transgram test4096.png`

For transgram, we need the buffers which are necessary for transpose and histogram. So we allocate buffers for the initial image, result image and the `bins` array. It is done the same way as is done for the histogram and transpose CUDA implementation.

	`ops-baseline.c`	`ops-cuda.cu`
Time (sec):	8,138001	0,101524
Speedup:	1,0	80,15776
Cache-misses:	124.670.411	370.003
LLC-loads:	174.279.998	2.046.991
Cycles:	32.624.449.403	209.696.604

Table 14: Results when using CUDA for transgram with image `test4096.png`.

Correctness:
The transposed image is correct but the histogram values slightly differ. The difference in values is explained at the correctness of histogram.

Final speedup: factor 80,16 for `test4096.png`.

4.1.4 Selgray

PERF: `perf stat -D 500 -d -d -r 15 -e cache-misses:u,cycles:u ./imglab-cuda selgray test4096.png`

For the CUDA implementation in selgray, there was only one buffer necessary. This is the buffer for the initial image. This buffer gets overwritten by the resulting image when finished. The block size and number of thread blocks is computed in the same way as transpose, histogram and transgram.
In the buffer we store the initial image with `cudaMemcpy`. Then we call the CUDA kernel `op_cuda_selgray` which then processes every pixel of the input image separately. And just like the histogram kernel, this kernel uses the `compute_intensity` and `compute_hue` device. Furthermore, the kernel and devices used are identical to the `ops-baseline.c` implementation.
When the kernel is finished, the output gets copied back to the buffer with `cudaMemcpy`. At last, the buffer gets freed with `cudaFree`.

	`ops-baseline.c`	`ops-cuda.cu`
Time (sec):	22,318626	0,070762
Speedup:	1,0	315,3999
Cache-misses:	61.701.839	313.431
LLC-loads:	86.566.574	2.577.596
Cycles:	86.369.149.987	150.867.770

Table 15: Results when using CUDA for selgray with image `test4096.png`.

Correctness:
Selective grayscale from `ops-baseline.c` and `ops-cuda.cu` produce identical images. This is tested with the compare utility.

Final speedup: factor 315,40 for `test4096.png`.

References

[1] Intel SIMD intrinsics:
`https://software.intel.com/sites/landingpage/IntrinsicsGuide/` (December 2018)

[2] Images used for testing:
`http://ca.liacs.nl/ca2018/ass3-data/` (December 2018)

[3] Compare utility used for checking if images are identical:
`https://blackboard.leidenuniv.nl/bbcswebdav/pid-4504179-dt-content-rid-6182921_1/courses/4032CMPA6-1819FWN/compare.c` (December 2018)