Final Report

Parallel Programming for Efficiency

Group 7
Jin Wen Ting, Roel Deckers

Contents

Abstract

In this report we present our findings on trying to optimize a MPEG encoder. We will discuss the theoretical limits of the system used and then go on to present our results as well as giving an in depth analysis of how these results were obtained and how performance could be improved further. We also give evidence which suggest that the motion-vector search stage of the program, which uses the most time, is limited by the instructions-per-second limit of the machine.

Benchmarking Setup

All results shown were obtained on a laptop[1] with an Intel i7-4750HQ.

The CPU part consists of 4 cores (8 threads) running at a maximum of 3.3 GHz each, each core can retire two single precision 8-way SIMD FMA instructions per cycle for a total of 422.4 GFLOPS.

The GPU part consists of 40 execution units running at a maximum of 1.2GHz each of which can retire two single precision 4-way SIMD FMA operations per cycle, for a total of 768 GFLOPS. Furthermore, each execution unit has up to 64KiB of local memory organized in 16 banks each 4 bytes wide. Access to the local memory is 1 cycle but will be serialized on bank conflicts, leading to a throughput of 64 bytes per cycle, or 76.8 GiB/s per execution unit or 3072 GiB/s total throughput. This memory is shared between workgroups running on the same execution unit and therefore puts a limit on the amount of workgroups than can be run on a execution unit at a time. Global memory uses the standard system ram, which has a theoretical limit of 25.6 GiB/s.

The machine runs Ubuntu 16, and all code was compiled using gcc 5.4 with the optimizations "-O2 -march=native". The OpenCL driver used was Beignet 1.1.1. Runtimes were measured using the

[1] Plugged in to the mains and with all power-saving turned off

high-resolution monotonic system clock, the clock itself has nanosecond level accuracy but the associated overhead is on the order of 10^4ns. Reported runtimes are the minimum result out of all frames.

All benchmarks were run agains the 'solar' image. A 2048×2048 movie of the sun consisting of 10 frames.

Results

Our final results are presented in figure 1. As can be seen, there is a huge performance increase most notably in the 'motionvector search (mvs)' routine but also in the combined 'convert' and 'lowpass' part which are the areas we focused the brunt of our efforts on.

Optimizations

pipelining

After an image is loaded, the first thing that happens is that we convert the colorspace from RGB to YCbCr, and then the image gets lowpassed. The original implementation performed this in two steps. First reading all the RGB values, converting them, and then writting back the YCbCr values to main memory before loading them once more to perform lowpass filtering.

We have optimized this routine not only by offloading it to the GPU but also by merging these two functions into one. The new kernel loads the RGB values from system memory and then converts all the colorspace and buffers the result in local memory before performing a lowpass filter on the results in local memory. This has dramatically improved performance, more than doubling the perfomance compared to the initial GPU offloaded version.

There is a slight overhead in terms of memory loads with this form of pipelining however; because local memory can not be shared between workgroups each workgroup has to load not only the pixels it will be converting but also the border surrounding it, as it needs those values to compute the result of the lowpass filter. As noted however, this effect does not have a significant impact on the final results.

Motionvector Search

As can be seen from figure 1 the motionvector search took by far the most time in the original code and thus was the main focus of our optimization efforts. The motionvector search algorithm works by splitting the image into 16×16 tiles and then, for each interior tile, finding the offset within a 48×48 area around the tile for which the previous frame has the lowest sum of absolute differences compared to the current tile. This means that there is a total of 1024 offsets for each of which the sum of absolute differences between two 16×16 tiles have to be computed.

We mapped this problem to the GPU by using workgroups consisting of 16×16 work-items, having each workgroup compute the result for a single tile and having each work-item in a workgroup compute the difference sum for 4 different offsets.

Initially we tried buffer all neccesary data into local memory, as they are accesed several times in a workgroup, and because each tile corresponds to 1KiB of local memory we tried to keep memory usage as low as possible. We did so initially by using a 2×2 pattern of tiles at a time instead of loading the 3×3 pattern that is needed as a whole in one pass. Initially this was implemented using dynamic pointer computation, which lead to the program crashing with a non-descript out-of-resources error as well as corrupting graphics memory of other running processes[2]. We then came up with a more suitable scheme

[2]While this might sound like an out-of-bounds error, after hours of debugging it became clear that this was not what was happening.

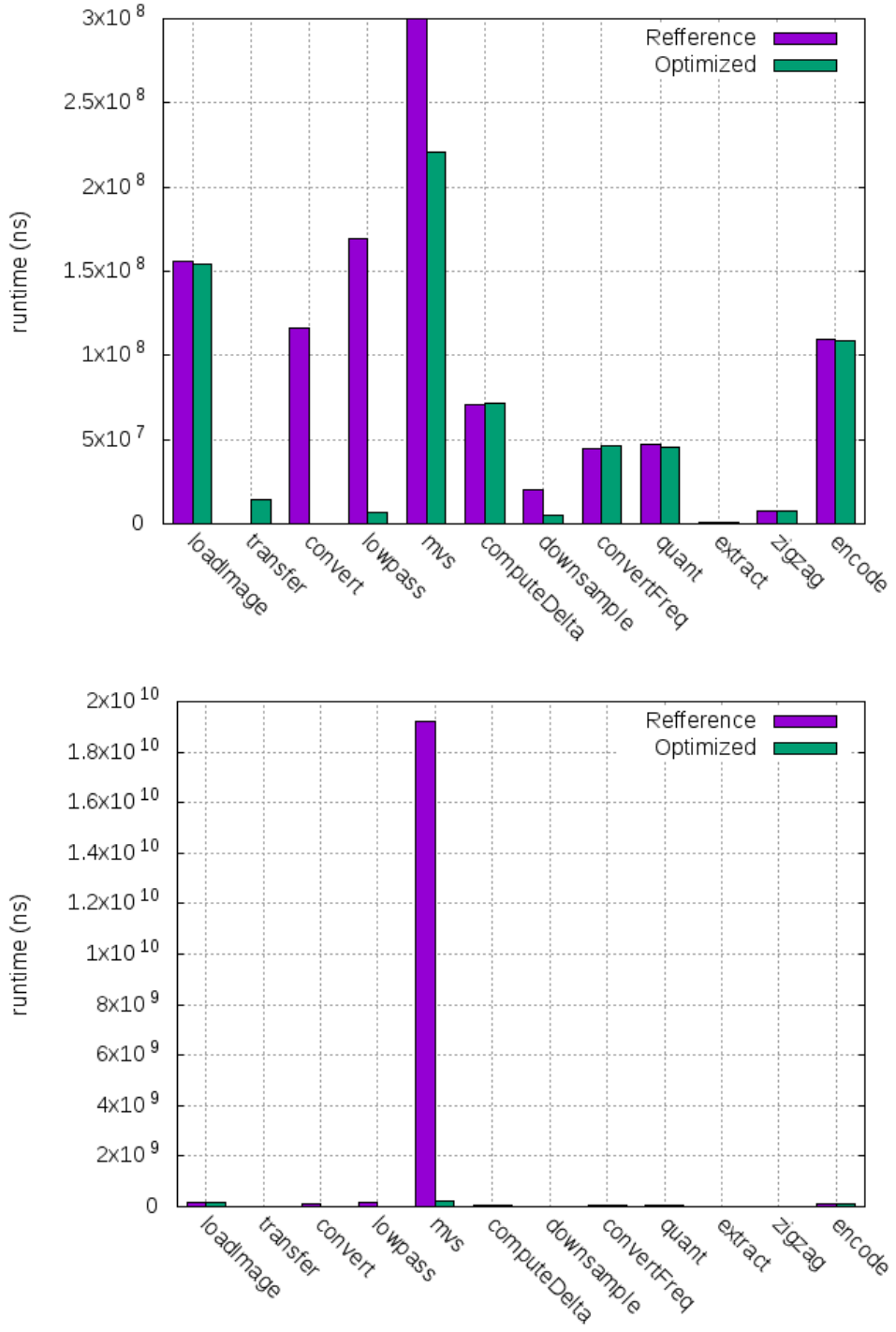

Figure 1: Final Results of our optimizations, note that the second graphic is zoomed in.

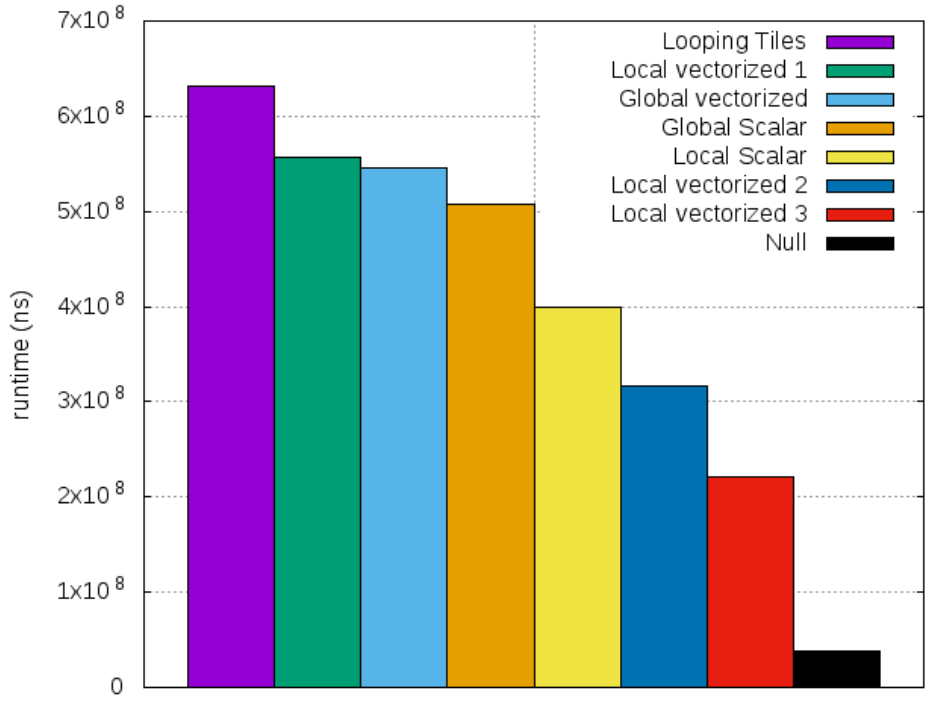

Figure 2: Results of our various experiments into optimizing the motionvector search routine. Null is the result of a kernel performing a simple addition instead of a sum of absolute differences.

using mod 32 access to the local buffer, as presented during the presentation, however it later turned out that this reduced performance rather than improving it.

Eventually we ended up trying several approaches, the results of each are presented in figure 2. 'Looping tiles' represents the method discussed above, all other methods are simple methods which copy the required 3×3 tiles into local memory all at once or load them directly from global memory. We then also tried different methods of vectorization, because the local memory version did outperform the global memory version in the scalar case we focused on vectorizing this one.

As can be seen our first version performed worse when vectorized than the scalar version. We believe this is due to the fact that in this case the different work-items each loaded a vector with and offset of one. This meant two things, firstly only one in four loads were aligned which can have a negative impact on some architectures, and secondly each load as a whole bank conflicted with atleast three other load instructions due to the overlap.

Our final vectorization attempt is writting in such a way that all loads are either aligned or broadcasts (i.e. access a single floating point from memory), any neccesary non-aligned vectors are constructed in register from the aligned vectors using a combination of shifts and masks.

Theoretical limits

Looking at the results we have a significant speedup, but how close are we actually to the theoretical performance limit? For this consider the maximum operations per second of the GPU, we will not look at the total GFLOPS because that measure only applies to raw numerical computation and the unfortunately the algorithm given does not use and FMA operations and requires several vector operations which do not using any floating point math such as taking the absolute value[3].

[3]Which is a bitmask operation, which could in theory be a 0-cycle operation on some architectures, this is however not the case for Intel's devices.

Note that we have 40 execution units running at 1.2GHz, each can retire two FMA instructions per cycle so let us be generous and say that the execution units can retire 2 instructions of any type on a float4 per cycle and ignore any overhead from loads or stores or regular integer (non-SIMD) instructions. Then we have a total theoretical throughput of 96 Gops/s.

Counting all the instructions in the inner sum of absolute differences and multiplying that by the amount of times it gets called gives us the total instruction count, dividing this by the theoretical operations per second gives us a minimum run time of 1.8×10^8ns, compare this to our measured runtime of 2.2×10^8ns and note that a kernel which only performs a simple addition in the inner loop runs at 0.38×10^8ns and it becomes clear we are very much limited by the total instructions per second throughput.

Future Work

Given more time, there are still a lot of optimizations that could be implemented in this code. For one, everything up to 'zigzag' would benefit from being run on the GPU (it is doubtful if the encoding section would greatly benefit from running on the GPU as it is highly branching). Furthermore, by moving more things over to the GPU we can reduce the overhead from copying to and and from system memory by performing as much steps as possible in one pass and using the local memory as an explicit cache. Another possible aveneu to further reduce the memory transfer overhead would be to look into storing half-precision data in any intermediate step that cannot be cached into local memory.

Having done so, we could further improve the run time by having the CPU part (loading the image and encoding) happen in the background while the GPU is processing a frame, in essence pipelining the entire program with respect to both the GPU and CPU.

In general we can also improve the encoding step by having it target a binary filetype similar to how "real" mpeg encoding works, instead of the string based backend that is currently employed. This would however be beyond the scope of this assignment. For the image loading function we could improve performance by using a more performant library and by decoding the raw image into a byte-sized format or possibly performing YCbCr conversion as it is being loaded (again, possibly combining this what half-precision data).

For the motionvector-search part of the code, we are limited by instructions per cycle so any optimizations should focus on reducing the total instructions used. One way to do that would be to have each work group handle a column (or several) of 16 pixels instead of a single 16×16 tile. A large part of the cycles taken is spend on permuting and blending different float4 vectors. By handling multiple tiles at once we could (assymptoptically) reduce the total shift and permutations needed by half (or up to a quarter in the case of multiple-tile wide columns), as the pixels used for lowest offset comparison with the top tiles are exactly those used by the highest offset comparison of the tile below. By increasing the width of the column we can reduce the total instruction count even further. The optimal size would then be determined by how large one can make these columns, while still saturating all the 40 execution units and making sure one does not run out of local memory or threads to saturate the execution units.

Another method, which has not been tried, might be to split up the inner loop such that first each work-item access all vectors it needs which are aligned properly, then it accesses those same vectors shifted by one, two and then three. Having each work-item be strided by four steps. This would remove the need to shift and blend but will cause three quarters of the accesses to be unaligned. Performance hits of unaligned accessess remain to be determined.

Discussion & Conclusion

By pipelining and efficient use of local memory we have been able to achieve huge relative speedups. Work on the iGPU part did not always prove to be as straightforward as expected. The memory access to either global or local memory seems to benefit heavilly from the most simplest of access patterns. Even having each work-item access memory mod 32 introduced a performance hit, this was not expected. We also did not get a 4 times performance improvement by vectorizing on 4-wide elements (the preffered and native size of the iGPU), which can probaly be attributed to the overhead of having to permute and blend vectors in registers for this algorithm.

Tooling for analysis of hardware performance proved to be quite powerful but very user-unfriendly. The Intel vTune amplifier allows one to sample GPU kernels and analyze various aspects but the information is not presented in a particularly enlightening way and the setup is quite difficult. In the end, simple trial and error with various methods proved to be the most effective method for debugging and optimizing our kernels, in contrast to CPU optimizing where it proved to be very useful previously.

Notes

Note that the makefile builds two executables, the refference one and the OpenCL one. The OpenCL one has a mandatory argument which is the full path to the kernel that needs to be used. Running 'make data' will run all possible version and produce timing files in the data directory, running 'make plots' will also produce the graphs shown.

The program has a limitation of assuming each file is a multiple of 16.