OpenCL CNN, now with 100% more GPU programming and 579% more performance.

David Akeley

1 Introduction

My GPU CNN implementation is considerably faster than the sequential CPU version:

```
> make test 2>&1 | grep Perf
Perf: 2848.23 GFlops
> make test-seq 2>&1 | grep Perf
Perf: 10.4025 GFlops
```

2 Parallelization Strategies

I'll summarize the main parallelization strategies from the CPU CNN project, and the changes done for this lab. This lab's code was mostly the same as the previous lab's – even my initial CPU code had `A+` performance (900 GFLOPS) after minor tweaking, without even using any local memory.

Tiling: The code I wrote supports tiling on all three output dimensions: `i`, `h`, and `w`; each work item computes one such `i_tile X h_tile X w_tile` output tile in the resulting 3D matrix. The main differences for this lab were:

1. A reduction of `i`-tiling parameter – I chose $4 \times 2 \times 8$ for the CPU version, while I settled on $1 \times 2 \times 8$ for the GPU version.
2. I introduced thread-level parallelism along the `i`-dimension. The CPU version computed an entire "column" of output pixels in one work item, exploiting thread-level parallelism only in the `h`- and `w`-dimensions. It was evident that this wouldn't be enough TLP for the GPU, so I now divide up the output array into work items along all three dimensions.

Vectorization: I don't believe that modern Nvidia GPUs support vectorization at all – they depend entirely on massive thread-level parallelism instead. Nevertheless, I kept using the `float16` type anyway. My code ties the vector width to the `w`-tiling parameter, so changes to the vector width also affect the efficiency of the tiling. For whatever reason, the top-scoring parameters all resulted in the `float16` type being used, just like the CPU version. This may be solely because this produces the optimal `w`-tile size, but the vectorization seemed harmless enough, so I didn't try to separate out the vectorization and `w`-tiling parameters into two parameters.

I-tiling and loop fusion: Funnily enough, the optimal tile size for the `i`-dimension was 1, which obviously defeated the loop fusion I did along the `i`-dimension in the CPU version (basically, I improved instruction-level parallelism in the CPU version by doing computation for `i_tile`-many output feature layers at once). The reasons I cited in the previous report are both no longer relevant for GPU programming:

1. I argued that doing `i`-loop fusion would help hide the latency chain of individual FPU instructions. The warp scheduler and aggressive symmetric multithreading allows the GPU to better hide instruction latency without this manual loop fusion optimization.

2. I also observed that tiling `i` by some parameter `N` would reduce memory reads to the input array by a factor of `N`, due to data re-use. The GPU's shared memory (`__local`) makes this optimization redundant – I was still able to exploit data re-use with TLP, rather than with ILP.

Instead of doing loop fusion on the `i`-dimension, I instead assign many `i`-layers to a single work group (the optimal parameters I found produce very "tall but narrow" work groups). Parallelizing a work group mostly along the `i`-dimension (instead of `h` or `w`), produces the same input array read re-use optimization that loop fusion did for the CPU version.

Actually, now that I think about it, despite Eddie's advice, I seem to have been able to get outstanding performance even though I hardly did any loop fusion. The generating script for my kernel code supports loop fusion, but the optimal fusion parameters turned out to be `1` surprisingly often.

3 Optimizations

3.1 Local Memory

I seem to have been able to get away with relatively little change from the CPU version. The only major optimization I had to apply was caching `input` array contents in a `__local` (shared) array. This local array stores all the inputs needed for one `j`-iteration of an entire work group.

This manual caching was a little bit challenging: the convolutional kernel introduced an "aura" that made the local buffer unusually sized – 12 by 36 – and it wasn't clear how to map a single thread's coordinate (local id) to an assignment of a portion of the local array to fill. I eventually settled on this code:

```
#define I_local_rows 12
#define I_local_cols 36

/* ... */

    __local float I_local_layer[I_local_rows][I_local_cols];

/* ... */

    barrier(CLK_LOCAL_MEM_FENCE);
    const int local_id = 2 * (2 * get_local_id(0) + get_local_id(1))
        + get_local_id(2);
    const int w = local_id + w_global_tile * 2;
    if (local_id < I_local_cols) {
        int h = 2 * h_global_tile;
        for (int i = 0; i < I_local_rows; ++i) {
            I_local_layer[i][local_id] = I_layer[h++][w];
        }
    }
    barrier(CLK_LOCAL_MEM_FENCE);
```

Basically, I decided not to try to force any connection between a thread / work item's assigned `h` and `w` coordinates, and the `h,w` coordinates of the local memory pixels it fills. Instead I just convert the thread's local coordinates to 1D (the `local_id` variable), and load one row of the local memory in parallel in every step of the loop. In order to minimize bank conflicts and keep global memory access sequential, each thread is assigned one column of the local array to load, instead of one row.

Excess threads that don't have a column to fill sit idle (this is guarded by the rare GPU `if` statement). I recognized the potential inefficiency of this, but my attempts to fix it badly reduced performance, so I just tolerated it. One possibly-unfortunate side effect of this is a marginal performance bias in favor of tile sizes with higher `w`-tiling and lower `h`-tiling parameters, since this results in "short-but-wide" local arrays that parallelize better with my local memory loading scheme.

Performance Comparison:

```
> ./gen.py 1 2 8 16 2 2 0 && make test 2>&1 | grep Perf # No local memory
global workgroup count: 3136
Perf: 428.71 GFlops
> ./gen.py 1 2 8 16 2 2 && make test 2>&1 | grep Perf # With local memory
global workgroup count: 3136
Perf: 2840.26 GFlops
```

3.2 Optimal Parameter Search

For the first three labs, I used a combination of intuition and theoretical reasoning in order to decide on a small set of parameter candidates (for loop fusion, tile sizes, etc.), and tested their performance manually instead of using a script. This was out of laziness (actually mostly pride), but this time, my brilliant roommate Marlon Trifunovic finally convinced me to relent and use a script to automatically search for the optimal parameters.

The relevant parameters are:

`i_tile`, `h_tile`, `w_tile`: The size of a tile of the 3D output matrix that an individual work item is assigned to fill. **Note:** Although these are "just changing parameters", they heavily modify the code due to my automatic loop fusion, and setting up my `gen.py` script to automatically rewrite my kernel code based on these parameters was not trivial.

`i_local_count`, `h_local_count`, `w_local_count`: The count of work items per work group in each dimension. These parameters were also not easy to change, because they required recalculation of the size of the local memory (which stores the input layer that all work items in a work group share per `j`-iteration).

I didn't think that the computer could find a better parameter set than I, with my magnificent human mind, could. As usual, it turned out that Marlon was right and I was wrong. This compares the performance of the best parameters I found manually (top) and the best one found by the script[1] (bottom):

```
# ./gen.py  i_tile  h_tile  w_tile  i_local_count  h_local_count  w_local_count
> ./gen.py 4 1 8 8 4 1 && make test 2>&1 | grep Perf
global workgroup count: 3136
Perf: 2297.64 GFlops
> ./gen.py 1 2 8 16 2 2 && make test 2>&1 | grep Perf
global workgroup count: 3136
Perf: 2825.9 GFlops
```

My explanation: My manual tests all used an `i_tile` parameter greater than 1 – this allowed for my manual loop fusion along the `i`-dimension. As reasoned earlier, this turned out to not be necessary on GPUs due to their high TLP doing a better job at hiding latency than my manual loop fusion, and I needed the cold indifference of a computer script to get me to realize this fact and pull my proud head out of my butt.

Another notable observation is how aggressively high the `i_local_count` parameter is compared to `h_`- and `w_local_count`. The 2 other top scoring parameter sets had an even higher `i_local_count`: 32 or 64. This compensates for the memory re-use lost by eliminating `i`-loop fusion. I don't think I ever had the imagination to try manually testing `i_local_count`s that high, which also explains why I was so irrationally attached to the `i`-loop fusion optimization.

To conclude: I've been served humble pie, and I'm glad my roommate set me straight.

[1] Actually, I decided to submit the third best one found. The top 3 all have very similar performance on Nvidia, but have drastically different performances on my own PCs, all of which use AMD graphics. The final parameter set I chose yield good performance on all three platforms, although there were some parameters that gave even greater performance – 3.7 teraflops – on my AMD GPU, yet had mediocre (but not poor; still above 2 TFLOPS) on Nvidia. I'm actually quite disheartened by how sensitive GPU programs are to platform-specific tuning.

4 Work Groups & Work Items

These are the parameters ($i \times h \times w$) I settled on for my submission:

Work item output tile size: $1 \times 2 \times 8$

Work items per work group: $16 \times 2 \times 2 = 64$

Work group output tile size: $16 \times 4 \times 16 = 1024$

Total work groups: $3136 = \frac{256}{16} \times \frac{112}{4} \times \frac{112}{16}$

There are 2048 CUDA cores per Tesla M60 GPU, which does not match the 3136 work groups. Unlike with OpenMP or MPI, the number of concurrently-running threads doesn't seem to be exposed by OpenCL. I suspect that if there are more work groups than hardware cores available, OpenCL will simply defer some work groups rather than try to time-share cores between work groups. So there's no danger of thread oversubscription, unlike with MPI and OpenMP. Therefore, I think other concerns – chiefly the local buffer size, and memory re-use by `i`-tiling – dominate performance, and the optimal tile and work group size is influenced most by those concerns, rather than trying to precisely match the number of hardware cores.

5 Conclusion

Although I'm quite happy with the performance of my GPU CNN kernel, which in all likelihood is the best in the class, I'm left with a strangely empty feeling after all this. The code I've submitted is clearly not scalable; not in computer performance terms, but in human terms. One layer of a CNN is not even that hard of a problem, yet my (albeit small) kernel code is already a baffling hard-to-modify mess. The local memory optimization in particular is very fragile and was hard to get right. I've always prided myself on forcing myself to write commented, clear code (even if it's just some dumb Eggert assignment that no one will even read before feeding it to the autograde script), but the last two CNN assignments finally got me to break this tradition. It definitely was fun to tweak the performance of my code manually – it's just that now that fun's over, I'm a bit sad to see the ugly end result. Furthermore, at the back of my mind, I feel that the optimizations I had to do – loop fusion and especially loading local memory – are things that computers could easily do a better job at than I could if only they knew how, and I feel that if I did GPU programming for a living these manual optimizations would quickly become tedious instead of fascinating.

Although I praised OpenCL in my last lab report, and I still have warm feeling for it, it doesn't seem like it alone is high-level enough for humans to write optimal code for tasks more complicated than this one-layer CNN assignment. I suppose I really should try to learn Halide properly one of these days.

6 Appendix: Parameter Search Script

```
#!/usr/bin/python3
import os
max_gflops = 0.0
max_gflops_config = None
gflops_dict = {}

def record_benchmark(i_tile, h_tile, w_tile,
                     i_local_count, h_local_count, w_local_count):

    global max_gflops, max_gflops_config
    config = (i_tile, h_tile, w_tile,
              i_local_count, h_local_count, w_local_count)
    pipe = os.popen("./gen.py %i %i %i %i %i %i && make test 2>&1" % config)
    gflops = None
    for line in pipe.read().split('\n'):
        if "FAIL" in line:
            print("Warning, configuration %r failed", (config,))
        if line.startswith("Perf:"):
            gflops = float(line.split()[1])
            gflops_dict[config] = gflops
            better = (gflops >= max_gflops)
            if better:
                max_gflops_config = config
                max_gflops = gflops
            print("%24r %.1f GFlops %s" % (config, gflops, "GOOD"*better))
    pipe.close()
```

```
i_tile_choices = (1,2,4,8)
h_tile_choices = (1,2,4,8)
w_tile_choices = (2,4,8)
i_group_count_choices = (4,8,16,32,64)
h_group_count_choices = (4,7,14,28,56)
w_group_count_choices = (4,7,14,28,56)

sample_count = 3
for i in range(sample_count):
    for i_tile in i_tile_choices:
        i_item_count = 256 // i_tile
        for h_tile in h_tile_choices:
            h_item_count = 112 // h_tile
            for w_tile in w_tile_choices:
                w_item_count = 112 // w_tile
                for i_group_count in i_group_count_choices:
                    if i_item_count % i_group_count != 0: continue
                    i_local_count = i_item_count // i_group_count
                    for h_group_count in h_group_count_choices:
                        if h_item_count % h_group_count != 0: continue
                        h_local_count = h_item_count // h_group_count
                        for w_group_count in w_group_count_choices:
                            if i_group_count * h_group_count * w_group_count < 1920:
                                continue
                            if w_item_count % w_group_count != 0: continue
                            w_local_count = w_item_count // w_group_count
                            record_benchmark(
                                i_tile, h_tile, w_tile,
                                i_local_count, h_local_count, w_local_count)

print("Best: %r %.1f GFlops" % (max_gflops_config, max_gflops))

out = open("bench.out", 'w')
for config, gflops in gflops_dict.items():
    out.write("%r %.1f\n" % (config, gflops))
out.close()
```

7 Appendix: Parameter Search Results

Top 64 scoring entries (best of three performance recorded)

```
(i_tile, h_tile, w_tile, i_local_count, h_local_count, w_local_count) GFLOPS
(1, 2, 8, 32, 1, 2) 2961.8
(1, 2, 8, 64, 1, 1) 2846.5
(1, 2, 8, 16, 2, 2) 2830.7      [chosen by virtue of good performance on AMD]
(1, 2, 8, 32, 2, 1) 2820.4
(2, 1, 8, 8, 4, 2)  2785.1
(1, 2, 8, 16, 4, 1) 2781.4
(2, 2, 4, 8, 2, 4)  2765.7
(2, 2, 8, 8, 4, 1)  2752.6
(2, 4, 2, 8, 1, 8)  2718.5
(2, 2, 4, 8, 4, 2)  2716.7
(2, 1, 8, 8, 8, 1)  2714.5
(2, 1, 8, 4, 8, 1)  2704.2
(4, 2, 2, 2, 4, 8)  2690.8
(1, 4, 4, 16, 1, 4) 2680.9
(1, 2, 8, 16, 2, 1) 2670.0
(1, 4, 4, 64, 1, 1) 2666.1
(2, 4, 4, 8, 2, 2)  2663.5
(1, 4, 4, 16, 1, 2) 2653.6
(2, 2, 4, 8, 2, 2)  2645.8
(1, 4, 4, 32, 1, 2) 2638.1
(4, 2, 2, 2, 4, 4)  2636.9
(2, 1, 8, 4, 8, 2)  2632.6
(2, 1, 8, 8, 4, 1)  2622.0
(2, 2, 4, 4, 4, 2)  2613.7
(2, 1, 8, 4, 16, 1) 2611.9
(2, 2, 8, 16, 2, 1) 2605.2
(1, 1, 8, 8, 4, 2)  2603.3      [Best on AMD Vega 64; max 3681.8 GFLOPS]
(1, 4, 4, 32, 2, 1) 2597.5
(2, 2, 4, 4, 4, 4)  2591.6
(1, 1, 8, 8, 8, 1)  2590.7
(1, 2, 4, 8, 2, 4)  2581.0
(2, 4, 2, 8, 1, 4)  2573.6
(2, 2, 4, 8, 8, 1)  2565.6
(2, 4, 4, 16, 1, 2) 2561.7
(1, 1, 8, 8, 8, 2)  2538.5
(1, 4, 8, 32, 1, 1) 2531.9
(8, 1, 2, 1, 8, 8)  2507.3
(1, 2, 4, 8, 2, 2)  2500.9
(1, 1, 8, 8, 4, 1)  2486.8
(4, 1, 8, 2, 16, 1) 2480.2
(1, 1, 8, 8, 16, 1) 2475.8
(4, 1, 4, 2, 8, 4)  2467.4
(8, 1, 4, 2, 8, 2)  2459.0
```