OpenCL Convolutional Neural Network

David Akeley

1 Parallelization Strategy

This is pretty much the most baffling code I've written, although at least this time I got very good results. I will make a probably futile attempt at describing the parallelization strategies I used.

Rough Strategy / Loop Ordering:

The original sequential cnn code has 6 nested loops, with `i`, `j`, `h`, `w`, `p`, and `q` dimensions, listed in order from outer-most to inner-most. I do tiling along the `i`, `h`, and `w` dimensions, splitting them into `_hi` and `_lo` dimensions. The rough loop ordering I chose is

Global Work Groups:

```
h_hi
  w_hi
```

Kernel Loops:

```
    i_hi
      j
        p
          q
```

Loop Fusion:

```
            h_lo
              i_lo
```

Vectorized:

```
                w_lo
```

Tiling: The output 3D-array has `i`, `h`, and `w` dimensions. I tile on all three dimensions: each "column" of tiles (along the `i`-axis) is assigned to a single work group. (i.e. only the `h` and `w` tiling is exposed to the scheduler). The implementation function, `fill_output_tile`, computes & writes out an entire output tile, without depending on any intermediate results of other running kernels.

The `h,w` tiling provides the only coarse-grain parallelism I used for the assignment – all other parallelisation is fine-grained, or from vectorization.

Loop Fusion:

I loop along `i_hi`, `j`, `p`, and `q` the normal way. However, I unroll and fuse the `h_lo` and `i_lo` loops. This provides instruction-level parallelism: calculations for different `i` and `h` coordinates have no dependencies, so by servicing several `(i,h)` coordinates per iteration, the latency of individual instructions are better hidden. Fusion along the `i` dimension also saves on memory reads to the `input` array.

Vectorization:

I chose to vectorize along the `w`-direction. The tile size I chose exactly matches the CPU's vector width: the `w` tile size is 8, which due to the `2x2` max pooling step exactly matches the vector width of 16. The `w` dimension was a natural candidate for vectorization, since this was the inner-most dimension of the `input` and `output` arrays and would allow for easy vector memory operations.

I suspect many of my classmates tried to vectorize along `p` or `q` instead – this seems intuitive (applying a convolutional kernel looks like an obvious vector operation). However, I think those are actually bad candidates for vectorization – the 5x5 kernel size limits the vector width, and many "horizontal" (within-vector) operations would be needed.

I'm actually quite surprised by how high up the `p` and `q` dimensions ended up in my loop hierarchy. I don't think that the `p,q` loops are even unrolled. I treat each entry of the kernel as a scalar quantity, performing scalar-vector multiplies in each iteration of the `p,q` loop.

I also believe that alignment is overrated here. Intel knew that alignment would mostly be a pipe dream for such wide vectors, so they put a lot of work into the performance of `AVX 512` unaligned memory access. I don't use any aligned memory at all, mostly because I felt that any cleverness to force alignment would cost more than the time saved by alignment; yet my kernel's performance still exceeded my expectations.

Max Pooling:

The max pooling and ReLu steps are completely separate from the convolution step (within a single `fill_output_tile` call); I didn't try to save on temporary buffer space ("C buffer") by trying to fuse the convolution and max pool/ReLu steps. However, due to the small tile size, and the enormous amount of register space provided by the `AVX-512` extension – 32 vector registers of 16 floats each – I was actually able to store this entire temporary "buffer" in registers.

Hao Zhuang ("Tim") encouraged me not to completely remove the temporary buffer, since doing so inhibited some loop transformations. Indeed, the dimensions that correspond to the temporary buffer – `h_lo` and `i_lo` – are exactly those involved in the loop fusion optimization I performed.

The max pooling and ReLu steps are vectorized as well; unfortunately though, a fair amount of horizontal operations are needed to do max pooling along the `w`-dimension.

2 Optimization Testing

Unlike with lab 1, I actually didn't start out with a really great solution and then incrementally de-optimize it until I had something to talk about. My initial solution really was poor, and the optimizations I did weren't initially obvious.

The exact history of my kernel code is lost, but I tried to faithfully generate older versions with my `gen.py` script, which outputs `intel.cl` and `params.sh` for me – `gen.py` takes three arguments: an `i`-tiling parameter, a `h`-tiling parameter, and an optimization level. If this optimization level is below 3, `gen.py` tries to recreate historical versions of my code.

My starting point was just loop tiling (along the `h` and `w` dimensions). I did not initially tile along the `i` dimension. The performance of this kernel was abysmal, both with the 8 `h`-tiling I initially used, and the 2 `h`-tiling I now use in my final submission:

```
> ./gen.py 1 8 0 && make test 2>&1 | grep Perf
Perf: 12.8577 GFlops
> ./gen.py 1 2 0 && make test 2>&1 | grep Perf
Perf: 13.0168 GFlops
```

I tried various optimizations, such as removing the temporary buffer (i.e. fusing the convolution and max pooling loops), as well as trying the "one local work group computes one output pixel" strat (i.e. don't manually tile). None of these attempts came close to escaping C-range performance.

2.1 Vectorization – optimization level 1

The first glimmer of hope came from me trying out manual vectorization. The first vectorization attempt I tried was to vectorize the `w`-dimension by 16 – even though all my classmates on Wednesday were talking about vectorizing the convolution kernel, I'm just clairvoyant and could tell that wasn't a good idea.

This first attempt gave slightly under `A+` range performance with my (at the time) `h`-tile size of 8. (There must be some subtle difference between my original vectorized code and the version generated by `gen.py`, since the `gen.py` version does barely get `A+` performance).

```
> ./gen.py 1 8 1 && make test 2>&1 | grep Perf
Perf: 144.008 GFlops
> ./gen.py 1 2 1 && make test 2>&1 | grep Perf
Perf: 161.594 GFlops
```

As I've come to expect, the compiler seems completely incapable of vectorizing my code automatically. I seriously just produced the replica of my original non-vectorized code by taking the vectorized version and dumbly replacing every vector op with a tight loop – yet the dunce of a compiler still can't do anything with it. To answer the assignment's question, the underlying performance benefits of vectorization are:

1. Fewer memory ops, both by converting groups of scalar read/writes to single vector read-/writes, and by enabling greater register capacity.
2. Removing the instruction decode bottleneck. Regardless of how many FPUs the CPU has, they can't be saturated with scalar code because the CPU can not decode instructions fast

enough to saturate all FPUs with scalar instructions alone.

2.2 i-tiling and loop fusion

The next optimization I applied was tiling on the `i` direction, and fusing the `i_low` loops together: instead of calculating each `i` layer of an output tile sequentially, I interleave the work of calculating 4 `i` layers at once within the tightest inner loop of the convolution:

```
// j-loop above
__constant float (*I_layer)[kInImSize] = I[j]; // j-layer of input image
__constant float (*W_layer0)[kKernel] = W[i0][j];
__constant float (*W_layer1)[kKernel] = W[i1][j];
__constant float (*W_layer2)[kKernel] = W[i2][j];
__constant float (*W_layer3)[kKernel] = W[i3][j];
for (int p = 0; p < kKernel; ++p) {
    for (int q = 0; q < kKernel; ++q) {
        const float16 weight0 = (float16) (W_layer0[p][q]);
        const float16 weight1 = (float16) (W_layer1[p][q]);
        const float16 weight2 = (float16) (W_layer2[p][q]);
        const float16 weight3 = (float16) (W_layer3[p][q]);
        const int wq = w_tile * 2 + q;
        for (int h_lo = 0; h_lo < 2 * kHTile; ++h_lo) {
            const int hp = h_tile * 2 + h_lo + p;
            const float16 img_row = vload16(0, &I_layer[hp][wq]);
            sums0[h_lo] += weight0 * img_row;
            sums1[h_lo] += weight1 * img_row;
            sums2[h_lo] += weight2 * img_row;
            sums3[h_lo] += weight3 * img_row;
        }
    }
}
```

This code is pretty unreadable, but the takeaway is that we're working on four output `i`-layers at once (buffers `sums0` through `sums3`), which naturally requires me to have four kernels loaded at once (`W_layer0` - `W_layer3`).

The benefits of this optimization are:

1. FPU latency hiding. The `+=` operations create a long dependency chain. Due to these dependencies, without this `i`-tiling & loop fusion optimization, many FPUs sat idle while waiting for the previous iteration's result to be available. By scheduling FPU ops for 4 `i`-layers in parallel, the FPUs are better utilized.
2. Memory operation reduction. With `i`-tiling, the number of reads to the `weights` array is the same, but there are $\frac{1}{4}$ the number of vector reads to the `inputs` array. Note how the `img_row` variable is shared by all 4 layers being computed. This is particularly important since the `img_row` reads are unaligned (this is a consequence of my decision to vectorize on the `w`-dimension).

A comparison between no `i`-tiling, and `i`-tiling by 4:

```
> ./gen.py 1 8 1 && make test 2>&1 | grep Perf
Perf: 144.008 GFlops
> ./gen.py 4 8 1 && make test 2>&1 | grep Perf
Perf: 209.98 GFlops
```

At this point, I was safely in `A+` range and getting quite excited, so I decided to turn my focus to the mundane task of searching for optimal tile sizes (`w` is fixed by the vector width, but `i` and `h` are free). I was not emotionally prepared for this kind of improvement though:

```
> ./gen.py 4 8 1 && make test 2>&1 | grep Perf
Perf: 209.98 GFlops
> ./gen.py 4 2 1 && make test 2>&1 | grep Perf
Perf: 410.645 GFlops
```

2.3 Temporary Array Removal – optimization level 3

(I'll explain optimization level 2 soon)

At this point, I was getting outrageously high on 400+ GFlops of performance, and my priapism wouldn't go away unless I kept coding. I realized that since the optimal `i`- and `h`-tiling parameters were so low, the `sums` temporary buffer (between the convolution and max pool/ReLu steps) could fit entirely within registers. At this point, there were 4 16-vector entries in each of the 4 `sums` arrays. This could easily fit within the 32 vector registers offered by `AVX 512`.

A comparison between level-1 optimization (memory sum buffers) and level-3 optimization (register sum buffers):

```
> ./gen.py 4 2 1 && make test 2>&1 | grep Perf
Perf: 410.645 GFlops
> ./gen.py 4 2 3 && make test 2>&1 | grep Perf
Perf: 421.48 GFlops
```

It turned out that this optimization didn't have all that dramatic an effect, but it's still measurable. A side effect of this optimization is that I had to unroll all the `h_lo` loops, since loop indicies could not be used to index registers. To control for that, I also tested a level-2 optimization level, which unrolls the `h_lo` loops but still stored the intermediate sums in a memory array instead of registers:

```
> ./gen.py 4 2 1 && make test 2>&1 | grep Perf
Perf: 410.645 GFlops
> ./gen.py 4 2 2 && make test 2>&1 | grep Perf
Perf: 412.68 GFlops
> ./gen.py 4 2 3 && make test 2>&1 | grep Perf
Perf: 421.48 GFlops
```

It appears that the bulk of the (small) effect of this array removal optimization comes from the actual array-to-register conversion, and not the loop unrolling. As typical, the improvement from this optimization is by reducing memory operations.

3 Execution Time Comparison

In this table, I compare the GFlops achieved by the provided sequential version, as well as my own version at various optimization levels. For my own version, I show results for five different tiling parameters, and the results for the version with my final tiling parameters when run sequentially (this is done by assigning all work items to a single local work group).

	sequential test	parallel tests				
threads	1	8	8	8	8	8
`i`-tiling	–	4	4	2	2	1
`h`-tiling	2	2	4	2	4	2
`CnnSequintial`	11.5355	–	–	–	–	–
Non-vectorized	2.284	–	–	–	–	13.0304
Vectorized	100.972	403.719	287.052	300.514	223.262	162.425
Submit version	106.915	415.099	301.550	334.186	224.946	171.913

(For the sequential test, `i`-tiling of 4 was used for the vectorized versions, and 1 was used for unvectorized. The tiling parameters do not apply for `CnnSequential`).

The code seems very scalable: the performance of the parallel vectorized versions are about 4X that of the version forced to run sequentially. This is about what I'd expect for very FPU-intensive code running on 4-physical-core machines. I tried playing around with `OPENCL_WORKGROUP_LOCAL`, but no really interesting patterns popped out. As far as I could tell, the best performance could be gotten (for CPUs) just by setting the parameter to `'1 1'` and letting the scheduler take care of everything:

```
> cat params.sh
export OPENCL_WORKGROUP_GLOBAL='56 14'
#export OPENCL_WORKGROUP_LOCAL='56 14'
> OPENCL_WORKGROUP_LOCAL='56 14' make test 2>&1 | grep Perf
Perf: 104.779 GFlops
> OPENCL_WORKGROUP_LOCAL='28 14' make test 2>&1 | grep Perf
Perf: 124.964 GFlops
> OPENCL_WORKGROUP_LOCAL='14 14' make test 2>&1 | grep Perf
Perf: 216.673 GFlops
> OPENCL_WORKGROUP_LOCAL='7 7' make test 2>&1 | grep Perf
Perf: 421.453 GFlops
> OPENCL_WORKGROUP_LOCAL='1 1' make test 2>&1 | grep Perf
Perf: 424.642 GFlops
```

Overall, I'm very happy with the performance of my final submission. For once, it seems that being a drunk overthinker paid off and the computer actually liked my complicated code. I found OpenCL a bit confusing at first, but now that I have good performance, I'm won over by how easy and elegant the vector syntax is for OpenCL, compared to that offered by intel intrinsics and the like. I'm also super impressed by Intel's `AVX 512` extension – I was skeptical when I first heard of it (especially because I found no hardware actually supporting it), but now that I've had a chance to play around with it, I'm blown away by the performance its wide vectors and enormous register capacities enable. Such a complicated, rarely-supported extension is particularly inviting to use through the friendly abstractions provided by OpenCL, which both improved vector syntax, and allowed me to do correctness testing locally on my own, non-`AVX 512` computer.

4 Appendix: Performance Testing Log

```
> make test-seq 2>&1 | grep Perf
Perf: 11.5355 GFlops
> ./gen.py 1 2 0 && make test 2>&1 | grep Perf
Perf: 13.0304 GFlops
> ./gen.py 4 2 1 0 && make test 2>&1 | grep Perf
Perf: 100.972 GFlops
> ./gen.py 4 2 1 1 && make test 2>&1 | grep Perf
Perf: 403.719 GFlops
> ./gen.py 4 4 1 1 && make test 2>&1 | grep Perf
Perf: 287.052 GFlops
> ./gen.py 2 2 1 1 && make test 2>&1 | grep Perf
Perf: 300.514 GFlops
> ./gen.py 2 4 1 1 && make test 2>&1 | grep Perf
Perf: 223.262 GFlops
> ./gen.py 1 2 1 1 && make test 2>&1 | grep Perf
Perf: 162.425 GFlops
> ./gen.py 4 2 3 0 && make test 2>&1 | grep Perf
Perf: 106.915 GFlops
> ./gen.py 4 2 && make test 2>&1 | grep Perf
Perf: 415.099 GFlops
> ./gen.py 4 4 && make test 2>&1 | grep Perf
Perf: 301.55 GFlops
> ./gen.py 2 2 && make test 2>&1 | grep Perf
Perf: 334.186 GFlops
> ./gen.py 2 4 && make test 2>&1 | grep Perf
Perf: 224.946 GFlops
> ./gen.py 1 4 && make test 2>&1 | grep Perf
Perf: 171.913 GFlops
> ./gen.py 1 2 0 0 && make test 2>&1 | grep Perf
Perf: 2.28356 GFlops
```