Parallel Computing - Lab 2: Vectorization

- Benjamín Ocampo: nicolasbenjaminocampo@gmail.com
- Mateo de Mayo: mateodemayo@gmail.com

Before getting into the mud

In order to improve the performance obtained in the first laboratory and take advantage of SIMD, the entire project was migrated to the redblack traversal. Similar to a checkerboard, cells are going to be colored and separated in RED and BLACK squares. Then, cells will be updated separately by means of its color, leading to independent loads and writes. In order to improve the spatial and temporal locality, cells of one color must be stored together.

One improvement that we were not aware of in the previous lab, was that the function *advect* is called twice in order to perform the function *vel_step*. Both of them called with u0 and v0 as parameters.

```
void vel_step(unsigned int n, float *u, float *v, float *u0, float *v0,
              float visc, float dt) {
  ...

  SWAP(u0, u);
  SWAP(v0, v);
  advect(n, VERTICAL, u, u0, u0, v0, dt);
  advect(n, HORIZONTAL, v, v0, u0, v0, dt);
  ...

}
```

If we get into the definition of advect:

```
static void advect(unsigned int n, boundary b, float *d, const float *d0,
                   const float *u0, const float *v0, float dt) {
  int i0, i1, j0, j1;
  float x, y, s0, t0, s1, t1;

  float dt0 = dt * n;
  for (unsigned int i = 1; i <= n; i++) {
    for (unsigned int j = 1; j <= n; j++) {
      x = i - dt0 * u0[IX(i, j)];
      y = j - dt0 * v0[IX(i, j)];

      ...
```

```
d[IX(i, j)] = s0 * (t0 * d0[IX(i0, j0)] + t1 * d0[IX(i0, j1)]) +
              s1 * (t0 * d0[IX(i1, j0)] + t1 * d0[IX(i1, j1)]);
```

Here, u0 and v0 will be used to compute **x** and **y** (which will be needed to reach the indexes i0, i1, j0, and j1). All of these in order to update the array **d** (which will be **u** and **v** as real values in the vel_step called). If we look at the indexes, they are the same for the update of **u** and **v**. Therefore, **x** an **y** will be computed twice!, leading to unnecessary (and expensive) reads and mults. So, we decided to implement a version of advect for *vel_step* (which we called *vel_advect*) that updates the arrays **u** and **v** in just one call of advect.

The entire migration to red-black traversal can be found in the slides.

After doing these changes, a comparison with the red-black code given at the begginning of the lab was needed. In this comparison, the new red-black version was called *baseline* on account of our new starting point.

If we look at the images posted in the slides, there is an improvement not only in the number of ns needed to update a cell but also in the number of cache references per cell iteration needed. The *rb* needs six times more llcache references than *baseline*. The ratio of cache hits are similar, but always avoid referencing to the llcache is better. This improvement is related to the independent updates which were talked about above. Since cells are stored differently in red-black, *rb* updates interpersed cells (which leads to unfriendly cache accesses). It is also interesting that *baseline* results resemblance to what was obtained at the end of lab 1.

Optimizations

Up to this point, we needed to take different paths, one of us worked with *intrinsics* and the other one with *ispc*. Nevertheless, both of us tried take the same approaches in order to compare their outcomes.

In each of the following titles, we briefly describe how the modification was done along with some implementations details that may be relevant. The four next items talk about the main loops found in navierstokes.

linsolve

This is a linear equation solver and it is the main loop of the program, so it was our first target to vectorize as well. We applied a simple vectorization scheme in which 8 elements are computed in the same loop only one would have been done before.

Some details of the implementation are as follow:

- `fload2x4` is an alias for the intrinsic that loads two unaligned 128b vectors to a 256 avx register, we used this for loading instead of the regular `_m256_loadu_ps` as recommended in the intel optimization manual (14.6.2) for unaligned memory accesses before skylake. It improves the performance a bit.
- Shared load: we will let this one for the `shload` section.

During the development of the project, some ideas came up as to how to improve the iterative method, for the record we will list some of those:

- As the new redblack algorithm is a little less precise, because in each iteration, all the neighbours of the red cells are an iteration behind, compared to the previous method in which only half neighbours were behind. What would happen if you have neighbors that are 2, 3 or more iterations behind? This idea does, in fact, improve spatial and temporal locality, and thus the performance, but we ended up not using it as it needed more iterations of grace than what we thought was acceptable for the visual quality of the simulation.
- The idea of using float16 to compute over 16 elements instead of 8 was tempting. However, we discovered that intel does not provide more than two operations on float16: encoding and decoding, you can't do arithmetic or anything on 16 bit floats. And so the idea fall right there before knowing if their precision was enough.
- We did a test branch in which we aligned the memory to 32bytes but it had many drawbacks. However, we bring this idea back again in the `stream` section.

In the case of ispc, it was mostly the same, but we applied the vectorization scheme by means of *programCount* which can be a multiple of the vector unit size.

Details of the ispc implementation are as follow:

- Loop iterators are uniform values. If this is not the case, ispc uses *vmaskmovps* in order to evaluate the loop condition (Which is usually true every iteration).
- How can we avoid the use of masks? Some ideas came into our minds. One possibility would be the use of cfor, which tells the compiler that the loop condition would be usually true. Another possibility is stopping at the position *width - programCount + start* and use an if clause to check boundaries (Which will be compiled as just one *vmaskmovps* at the end of updating a row of the grid).
- Anyway, these *vmaskmovps* were not so problematic. Just a few nanoseconds were gained at the expense of readability.

addsource

There is not much to say here, a plain vectorization with loop peeling to sum over all the grid elements. In the case of ispc, We just used a foreach clause over all the grid elements.

advect

After measuring with `perf`, `coz-profiler` and `toplev`, both `advect` and `project` appeared as the next bottlenecks in the program after improving `lin_solve`.

The vectorization of advect followed directly the previous improvements made in the non vectorized version. It was rather hard to debug vectorization as many states were in place, about 50 `__m256` variables were needed.

Two insights that came from the tortuous debuggin were:

- `_mm256_mul_epi32` does not multiply 8 elements as one would be tempted to believe after using `_mm256_mul_ps`
- The default conversion intrinsic `_mm256_cvtps_epi32` does not behave like a c cast, it rounds.

On the other hand, The ispc version of advect was straightforward, there was no necessity of a thoroughly debugging as in the case of intrinsics.

Unfortunately, there is an issue in the vectorization of both versions. It is the use of gathers because of indexes that were calculated by means of velocity values. Since these accesses would be unpredictable we did not found a way to vectorize advect without this drawback.

project

Project vectorization was rather straightforward, and not much to say here more than we thought it was appropiate to split the loop in two as so the two accesses didn't thrash the cache.

blocks

Against all our hopes, loop blocking on `lin_solve` degrade by much the performance, and it was our main idea for gaining more performance as it have had a deep impact on the previous project. Our main suspicions is cache thrashing happening on `lin_solve`. More on this on the `stream` section.

shload

In `lin_solve`, when reading left and right neighbours vectors, seven elements overlap, the idea with a shared load is simple: just use one load, and then extract its last element for reuse within the next cell computation.

The gains again, unfortunately, were small as the load that was being saved was probably in cache already.

icc

This was just a quick test on icc with flags `-O3 -xHost -fp-model fast=2 -no-prec-div` and indeed it returned some gains.

stream

`toplev` informed that indeed, our application is mainly memory bound (~50%), and after tinkering a little with reads and writes that could be causing problems we discovered that many write only addresses were being brought to cache lines.

There are two main ways of making addresses "non temporal" for the intel microarchitecture, one is setting the memory region as non temporal, which would be a bad idea as we later in the program need to read from that memory. The second way is with `movnt` instructions or the `stream` intrinsics which perform writes directly to `DRAM` without passing through cache. Unfortunately for us, this `stream` instructions require the writes to be aligned, and a great portion of our code relies on unaligned writes.

As we had not much time for doing a full rewrite of the program we came up with an idea in which we merge the right and left borders of the grid so as to make all non-boundary cells be aligned. In this way, we were able to make a lot of write-only loops be streamed directly into main memory freeing the cache for other addresses. This increased performance a fair bit, about 20~30%.

Resources

- Intel optimization manual: https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pd
- Agner fog optimization manuals: https://www.agner.org/optimize/
- Peter Cordes on stack overflow, he is in charge of the assembly tag and he's been doing an amazing job desmitifying it, each of its answers has a lot of useful information.
- Nvidia gpu gems: https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-30-real-time-simulation-and-rendering-3d-fluids