Chapter 2: Case study

Version 7: Better use of cache memory [advanced]

In version 4 and version 5 we worked hard to reuse data that we get from the main memory to CPU registers as much as possible. However, there are also many levels of cache memory between the main memory and the CPU. We will now try to make sure that we also reuse data that we get from the main memory to the cache memory as much as possible.

Cache memory hierarchy

Cache memory works automatically. There are three levels of (data) cache in the processor that we use:

- 32 KB of L1 cache per core
- 256 KB of L2 cache per core
- 6 MB of L3 cache, shared among all cores.

Whenever we access some element that is stored in the main memory, the CPU will always try to fetch it from L1. If the data is in L1, great, otherwise L1 cache will try to fetch it from L2 cache. If needed, L2 cache will try to get data from L3 cache, and finally, if needed, L3 cache will try to get data from the main memory.

If the data element was not already present in L1 cache, it will get stored in L1 cache, in the hope that the program might need it soon again, and some not-so-recently-accessed data elements will need to make way for it. The same happens on each level of cache.

Ideally, we would like to organize our program so that as often as possible, we will find data that is stored as close to the CPU as possible. In general, we would like to refer to some memory that we have recently accessed and that is still stored in some of the caches. Main memory bandwidth is relatively low and latencies are very high.

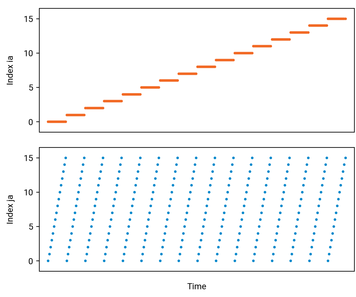
Improving reuse

Let us now look at the outermost loops in version 5:

```
for (int ia = 0; ia < na; ++ia) {
    for (int ja = 0; ja < na; ++ja) {
```

Here `ia` determines a row of array `vd` that we will read in the innermost loop. Similarly, `ja` determines a row of array `vt` that we will read in the innermost loop.

Ideally, in subsequent iterations we will repeatedly refer to same rows again and again, so that we would find them in the cache memory. However, as we have nested loops, `ia` stays constant for a long time (which is good), but `ja` goes through all values from `0` to `na-1` rapidly (which is bad):



In essence, there is a lot of reuse with array `vd`, but very little reuse with array `vt`. By the time we refer to the same row of `vt` again, we have already touched all other rows of the array, and it is so much data that it will not fit in any cache memory.

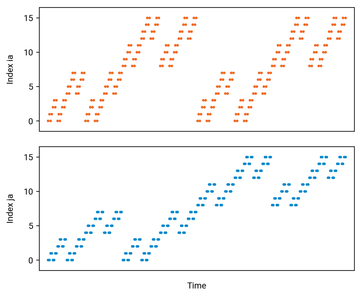
However, there is a simple fix: we can follow the Z-order curve to choose the order in which we process the pairs `(ia,ja)`. Here is a simple implementation of the idea: just interleave the bits of `ia` and `ja` to construct value `ija`, and then use `ija` as the sorting key to order:

```
std::vector<std::tuple<int,int,int>> rows(na*na);

for (int ia = 0; ia < na; ++ia) {
    for (int ja = 0; ja < na; ++ja) {
        int ija = _pdep_u32(ia, 0x55555555) | _pdep_u32(ja, 0xAAAAAAAA);
        rows[ia*na + ja] = std::make_tuple(ija, ia, ja);
    }
}
std::sort(rows.begin(), rows.end());
```

Here e.g. `_pdep_u32(x, 0x55555555)` turns the binary number `abcd` into `0a0b0c0d`, and `_pdep_u32(x, 0xAAAAAAAA)` turns the binary number `abcd` into `a0b0c0d0`. So if `ia` is `abcd` in binary, and `ja` is `stuv` in binary, `ija` will be `satbucvd` in binary.

Now if we look at the sorted array `rows` look at how `ia` and `ja` change, things look much better:



As we can see, we will revisit the same value very soon again, and then also a bit later again, etc. If there is room for multiple rows of data in any level of the cache, we will have lots of cache hits.

(Sorting may look like a silly thing to do here, and indeed there are more efficient ways to implement this. The reader is encouraged to experiment with better algorithms here, and see if it matters in terms of the overall running time.)

Shorter rows

If we simply followed this order, we would read, for example, the same row of `vt` twice in sequence and the same pair of rows of `vd` twice in sequence. Hence, if there is a cache near the CPU that is large enough to hold, e.g., 2 rows of `vt` and 2 rows of `vd`, we should get some nice cache hits.

Let us do the math. For example, when n = 4000, each row of `vd` contains 4000 vectors, each with 8 floats. This is, in total, 128 KB of data per row. For example, 2 rows of `vt` and 2 rows of `vd` already takes 512 KB of data. But this is much more than the size of the L2 cache!

There is a simple fix: we will calculate the result in narrow "slices". We will first extract e.g. 500 first elements of each row, and do calculations for that part. Then we repeat this for the next 500 elements and update the results, etc.

This way we will have shorter rows with only 16 KB of data per row, and e.g. L2 cache is now able to fit 8 + 8 such rows, which is already much better. We could make the rows even shorter than that, but at some point we will not get any benefit anymore; the overhead of managing all partial results will start to dominate.

Results

Now we are happy. The running time for n = 4000 is 0.7 seconds, we have improved the running time over the baseline by a factor of 151, and we are using now 93% of the theoretical maximum performance that we can get from this CPU, without resorting to an entirely different kind of algorithm.