

How to properly use prefetch instructions?

Asked 5 years, 4 months ago Modified 5 years, 3 months ago Viewed 10k times



16



I am trying to vectorize a loop, computing dot product of a large float vectors. I am computing it in parallel, utilizing the fact that CPU has large amount of XMM registers, like this:

```
_m128* A, B;
_m128 dot0, dot1, dot2, dot3 = _mm_set_ps1(0);
for(size_t i=0; i<1048576;i+=4) {
    dot0 = _mm_add_ps( dot0, _mm_mul_ps( A[i+0], B[i+0]));
    dot1 = _mm_add_ps( dot1, _mm_mul_ps( A[i+1], B[i+1]));
    dot2 = _mm_add_ps( dot2, _mm_mul_ps( A[i+2], B[i+2]));
    dot3 = _mm_add_ps( dot3, _mm_mul_ps( A[i+3], B[i+3]));
}
... // add dots, then shuffle/hadd result.
```

I heard that using prefetch instructions could help speedup things, as it could fetch further data "in background", while doing muls and adds on a data that is in cache. However i failed to find examples and explanations on how to use `_mm_prefetch()`, when, with what addresses, and what hits. Could you assist on this?

caching x86 sse prefetch dot-product

Share Improve this question

Follow

edited Feb 26, 2018 at 18:39



Peter Cordes

324k 45 593 836

asked Feb 26, 2018 at 18:04



xakepp35

2,848 6 26 53


- 2 Usually hardware prefetch does a good job for sequential access, and you don't need software prefetch. See also [What Every Programmer Should Know About Memory?](#) – Peter Cordes Feb 26, 2018 at 18:08
- 1 You might consider using 8 accumulators, though, instead of only 4, to better hide the latency of `addps` (especially on Skylake where it's 4 cycle latency). x86-64 has 16 xmm registers (or ymm with AVX). And *especially* if you're compiling with FMA so those mul/add operations can collapse into a single FMA. More details [on another SSE/AVX dot-product question](#). This might not help if you're mostly memory bottlenecked, though. – Peter Cordes Feb 26, 2018 at 18:13
- 1 You probably want to use `prefetchnta` if A and B are large and won't be read again soon. You want to prefetch once per 64B cache line, and you'll need to tune how far ahead to prefetch. e.g. `_mm_prefetch((char*)(A+64), _MM_HINT_NTA);` and the same for B would prefetch $16 \times 64 = 1024$ bytes head of where you're loading, allowing for hiding some of the latency of a cache miss but still easily fitting in L1D. Read Ulrich Drepper's "What Every Programmer Should Know About Memory?" (the PDF in my earlier link) for more about caching and prefetch. – Peter Cordes Feb 26, 2018 at 19:02

Join Stack Overflow to find the best answer to your technical question, help others answer theirs.

Sign up



even I cannot consider myself competent enough to use it properly without it backfiring on Skylake X.
 – [Mysticial](#) Feb 26, 2018 at 19:36

- 2 If you `prefetchnta` and it gets evicted before it is used (either because it's too early or because the other hyperthread is misbehaving), it gets evicted from all levels of cache and then refetched from memory on the actual read. The result is 2x the bandwidth cost and a massive slowdown on anything that's bandwidth-bound. At least `prefetcht0` seems to get evicted into the higher level caches.
 – [Mysticial](#) Feb 26, 2018 at 19:36 

Sorted by:

Highest score (default) 

1 Answer



The short answer that probably works for perfectly linear streaming loops like yours is probably: **don't use them at all, let the hardware prefetchers do the work.**

33



Still, it's *possible* that you can speed things up with software prefetching, and here is the theory and some detail if you want to try...



Basically you call `_mm_prefetch()` on an address you'll need at some point in the future. It is similar in some respects to loading a value from memory and doing nothing with it: both bring the line into the L1 cache², but the prefetch intrinsic, which under the covers is emitting specific [prefetch instructions](#), has some advantages which make it suitable for prefetching.

It works at cache-line granularity¹: you only need to issue one prefetch for each cache line: more is just a waste. That means that in general, you should try to unroll your loop enough so that you can issue only one prefetch per cache line. In the case of 16-byte `__m128` values, that means unroll at least by 4 (which you've done, so you are good there).

Then simply prefetch each of your access streams by some `PF_DIST` distance ahead of the current calculation, something like:

```
for(size_t i=0; i<1048576;i+=4) {
    dot0 = _mm_add_ps( dot0, _mm_mul_ps( A[i+0], B[i+0]));
    dot1 = _mm_add_ps( dot1, _mm_mul_ps( A[i+1], B[i+1]));
    dot2 = _mm_add_ps( dot2, _mm_mul_ps( A[i+2], B[i+2]));
    dot3 = _mm_add_ps( dot3, _mm_mul_ps( A[i+3], B[i+3]));
    _mm_prefetch(A + i + PF_A_DIST, HINT_A);
    _mm_prefetch(B + i + PF_B_DIST, HINT_B);
}
```

Here `PF_[A|B]_DIST` is the distance to prefetch ahead of the current iteration and `HINT_` is the temporal hint to use. Rather than try to calculate the right distance value from first principles, I would simply determine good values of `PF_[A|B]_DIST` experimentally⁴. To reduce the search space, you can start by setting them both equal, since logically a similar distance is likely to be ideal. You might find that only prefetching one of the two streams is ideal.

It is very important that the ideal `PF_DIST` **depends on the hardware configuration**. Not just on the CPU model, but also on the memory configuration, including details such as the

Join Stack Overflow to find the best answer to your technical question, help others answer theirs.

Sign up



experiment on the actual hardware you are targeting, as much as possible. If you target a variety of hardware, you can test on all the hardware and hopefully find a value that's good on all of them, or even consider compile-time or runtime dispatching depending on CPU type (not always enough, as above) or based on a runtime test. Now just relying on hardware prefetching is starting to sound a lot better, isn't it?

You can use the same approach to find the best `HINT` since the search space is small (only 4 values to try) - but here you should be aware that the difference between the different hints (particularly `_MM_HINT_NTA`) might only show as a performance difference in code that runs *after* this loop, since they affect how much data unrelated to this kernel remain in the cache.

You might also find that this prefetching doesn't help at all, since your access patterns are perfectly linear and likely to be handled well by the L2 stream prefetchers. Still there are some additional, more hardcoded things you could try or consider:

- You might investigate whether prefetching only at the start of 4K page boundaries helps³. This will complicate your loop structure: you'll probably need a nested loop to separate the "near edge of page" and "deep inside the page" cases in order to only issue the prefetches near page boundaries. You'll also want to make your input arrays page-aligned too, or else it gets even more complicated.
- You can try [disabling some/all of the hardware prefetchers](#). This is usually terrible for overall performance, but on a highly tuned load with software prefetching, you might see better performance by eliminating interference from hardware prefetching. Selecting disabling prefetching also gives you an important key tool to help understand what's going on, even if you ultimately leave all the prefetchers enabled.
- Make sure you are using huge pages, since for large contiguous blocks like this they are ideal.
- There are problems with prefetching at the beginning and end of your main calculation loop: at the start, you'll miss prefetching all data at the start of each array (within the initial `PF_DIST` window), and at the end of the loop you'll prefetch additional and `PF_DIST` *beyond* the end of your array. At best these waste fetch and instruction bandwidth, but they may also cause (ultimately discarded) page faults which may affect performance. You can fix both by special intro and outro loops to handle these cases.

I also highly recommend the 5-part blog post [Optimizing AMD Opteron Memory Bandwidth](#), which describes optimizing a problem very similar to yours, and which covers prefetching in some detail (it gave a large boost). Now this is totally different hardware (AMD Opteron) which likely behaves differently to more recent hardware (and especially to Intel hardware if that's what you're using) - but the process of improvement is key and the author is an expert in the field.

¹ It may actually work at something like 2-cache-line granularity depending on how it interacts with the adjacent cache line prefetcher(s). In this case, you may be able to get away

² In the case of software prefetch, you can also select some other level of cache, using the temporal hint.

³ There is some indication that even with perfect streaming loads, and despite the presence of "next page prefetchers" in modern Intel hardware, page boundaries are still a barrier to hardware prefetching that can be partially alleviated by software prefetching. Maybe because software prefetch serves as a stronger hint that "Yes, I'm going to read into this page", or because software prefetch works at the virtual address level and necessarily involves the translation machinery, while L2 prefetching works at the physical level.

⁴ Note that the "units" of the `PF_DIST` value is `sizeof(__mm128)`, i.e., 16 bytes due to the way I calculated the address.

Share Improve this answer

edited Apr 15, 2018 at 17:10

answered Feb 26, 2018 at 19:14

Follow



BeeOnRope

59.9k 15 204 383

1 It depends on how large your arrays are: if the arrays are comparable or smaller than the size of the LLC, then sure you might get some benefit from doing things backwards since a reasonable portion of the data from the last processing might be in the cache (but this is also complicated by new LLC features that try to detect streaming loads and change the replacement policy of the cache when they are detected). The more general approach to make that work is simply to block up your processing: instead of doing each phase entirely, try to alternate between the ... – BeeOnRope Feb 27, 2018 at 3:38

1 @Noah - did you look at part 4 in that series? It has some suggestions as to why multiple streams might be useful. I believe Dr McCalpin reported elsewhere that multiple streams helps also on Intel because the L2 prefetchers aren't able to generate enough outstanding requests from a single stream (they are limited by 4k boundaries among other things), so multiple streams helps in that way too. – BeeOnRope Feb 27, 2021 at 20:22

1 @Noah - I don't think that's what that post says. It says row number are in bits 18-32. Most of the parallelism at DRAM comes from using both channels. A small amount of additional parallelism comes from having more than one bank/rank (bits 14-17 in the post you linked) active, although I think this depends on the timing. E.g., if you read all the columns in a row back-to-back (ignoring channels, that happens by default for linear access with this mapping since the column is in bits 0-5, 7-13) that's already pretty efficient since you have almost 100% open row hit rate. \ – BeeOnRope Jun 21, 2021 at 21:56

1 You only need to open a new row every 2048 bytes or whatever the row size is. Now there is *some* latency in opening the new row, so a slightly more efficient strategy might be to have two rows active (in different rank/banks of course!), offset by say 1024 bytes, so that while one bank/rank is changing its open row, you are still going full steam on the other. The memory controller itself reorders stuff heavily, so I don't know if the offset is actually needed. I haven't really played with this stuff yet so I'm mostly guessing. – BeeOnRope Jun 21, 2021 at 21:58

2 My impression is that DRAM mapping and fine-grained memory controller behavior is much more important for random or semi-random access than linear access, as the latter basically works well by default and can achieve close to the DRAM bandwidth w/o any tricks (e.g., NT stores), although it may have been different on older hardware. – BeeOnRope Jun 21, 2021 at 22:00