# Deoptimizing a program for the pipeline in Intel Sandybridge-family CPUs

Asked 8 years, 7 months ago   Modified 2 years, 4 months ago   Viewed 50k times

348

I've been racking my brain for a week trying to complete this assignment and I'm hoping someone here can lead me toward the right path. Let me start with the instructor's instructions:

> Your assignment is the opposite of our first lab assignment, which was to optimize a prime number program. Your purpose in this assignment is to pessimize the program, i.e. make it run slower. Both of these are CPU-intensive programs. They take a few seconds to run on our lab PCs. You may not change the algorithm.
>
> To deoptimize the program, use your knowledge of how the Intel i7 pipeline operates. Imagine ways to re-order instruction paths to introduce WAR, RAW, and other hazards. Think of ways to minimize the effectiveness of the cache. Be diabolically incompetent.

The assignment gave a choice of Whetstone or Monte-Carlo programs. The cache-effectiveness comments are mostly only applicable to Whetstone, but I chose the Monte-Carlo simulation program:

```
// Un-modified baseline for pessimization, as given in the assignment
#include <algorithm>    // Needed for the "max" function
#include <cmath>
#include <iostream>

// A simple implementation of the Box-Muller algorithm, used to generate
// gaussian random numbers - necessary for the Monte Carlo method below
// Note that C++11 actually provides std::normal_distribution<> in
// the <random> library, which can be used instead of this function
double gaussian_box_muller() {
  double x = 0.0;
  double y = 0.0;
  double euclid_sq = 0.0;

  // Continue generating two uniform random variables
  // until the square of their "euclidean distance"
  // is less than unity
  do {
    x = 2.0 * rand() / static_cast<double>(RAND_MAX)-1;
    y = 2.0 * rand() / static_cast<double>(RAND_MAX)-1;
    euclid_sq = x*x + y*y;
  } while (euclid_sq >= 1.0);

  return x*sqrt(-2*log(euclid_sq)/euclid_sq);
```

```cpp
}

// Pricing a European vanilla call option with a Monte Carlo method
double monte_carlo_call_price(const int& num_sims, const double& S, const
double& K, const double& r, const double& v, const double& T) {
  double S_adjust = S * exp(T*(r-0.5*v*v));
  double S_cur = 0.0;
  double payoff_sum = 0.0;

  for (int i=0; i<num_sims; i++) {
    double gauss_bm = gaussian_box_muller();
    S_cur = S_adjust * exp(sqrt(v*v*T)*gauss_bm);
    payoff_sum += std::max(S_cur - K, 0.0);
  }

  return (payoff_sum / static_cast<double>(num_sims)) * exp(-r*T);
}

// Pricing a European vanilla put option with a Monte Carlo method
double monte_carlo_put_price(const int& num_sims, const double& S, const
double& K, const double& r, const double& v, const double& T) {
  double S_adjust = S * exp(T*(r-0.5*v*v));
  double S_cur = 0.0;
  double payoff_sum = 0.0;

  for (int i=0; i<num_sims; i++) {
    double gauss_bm = gaussian_box_muller();
    S_cur = S_adjust * exp(sqrt(v*v*T)*gauss_bm);
    payoff_sum += std::max(K - S_cur, 0.0);
  }

  return (payoff_sum / static_cast<double>(num_sims)) * exp(-r*T);
}

int main(int argc, char **argv) {
  // First we create the parameter list
  int num_sims = 10000000;   // Number of simulated asset paths
  double S = 100.0;  // Option price
  double K = 100.0;  // Strike price
  double r = 0.05;   // Risk-free rate (5%)
  double v = 0.2;    // Volatility of the underlying (20%)
  double T = 1.0;    // One year until expiry

  // Then we calculate the call/put values via Monte Carlo
  double call = monte_carlo_call_price(num_sims, S, K, r, v, T);
  double put = monte_carlo_put_price(num_sims, S, K, r, v, T);

  // Finally we output the parameters and prices
  std::cout << "Number of Paths: " << num_sims << std::endl;
  std::cout << "Underlying:      " << S << std::endl;
  std::cout << "Strike:          " << K << std::endl;
  std::cout << "Risk-Free Rate:  " << r << std::endl;
  std::cout << "Volatility:      " << v << std::endl;
  std::cout << "Maturity:        " << T << std::endl;

  std::cout << "Call Price:      " << call << std::endl;
  std::cout << "Put Price:       " << put << std::endl;

  return 0;
}
```

The changes I have made seemed to increase the code running time by a second but I'm not entirely sure what I can change to stall the pipeline without adding code. A point to the right direction would be awesome, I appreciate any responses.

---

## Update: the professor who gave this assignment posted some details

The highlights are:

- It's a second semester architecture class at a community college (using the Hennessy and Patterson textbook).
- the lab computers have Haswell CPUs
- The students have been exposed to the `CPUID` instruction and how to determine cache size, as well as intrinsics and the `CLFLUSH` instruction.
- any compiler options are allowed, and so is inline asm.
- Writing your own square root algorithm was announced as being outside the pale

Cowmoogun's comments on the meta thread indicate that it wasn't clear compiler optimizations could be part of this, and assumed `-O0`, and that a 17% increase in run-time was reasonable.

So it sounds like the goal of the assignment was to get students to re-order the existing work to reduce instruction-level parallelism or things like that, but it's not a bad thing that people have delved deeper and learned more.

---

Keep in mind that this is a computer-architecture question, not a question about how to make C++ slow in general.

c++ optimization x86 intel cpu-architecture

Share
Improve this question
Follow

edited Jun 20, 2020 at 9:12
Community Bot
1 ● 1

asked May 21, 2016 at 9:29
Cowmoogun
2,557 ● 4 ● 13 ● 17

113 I hear the i7 does very poorly with `while(true){}` – Cliff AB May 22, 2016 at 2:36

3 Number 2 on HN atm: news.ycombinator.com/item?id=11749756 – mlvljr May 22, 2016 at 20:36

5 With openmp if you do it badly you ought to be able to make N threads take longer than 1. – Flexo - Save the data dump ♦ May 22, 2016 at 20:43

9 This question is now being discussed in meta – Madara's Ghost May 23, 2016 at 11:38

3 @bluefeet: I added that because it had already attracted one close-vote in under an hour of being re-opened. It only takes 5 people to come along and VTC without realizing reading comments to see it's under discussion on meta. There's another close vote now. I think at least one sentence will help avoid close/reopen cycles. – Peter Cordes May 24, 2016 at 16:01

## 4 Answers

Sorted by: Highest score (default)

430

+50

Important background reading: **Agner Fog's microarch pdf**, and probably also Ulrich Drepper's What Every Programmer Should Know About Memory. See also the other links in the `x86` tag wiki, especially Intel's optimization manuals, and David Kanter's analysis of the Haswell microarchitecture, with diagrams.

Very cool assignment; much better than the ones I've seen where students were asked to optimize some code for `gcc -O0`, learning a bunch of tricks that don't matter in real code. In this case, you're being asked to learn about the CPU pipeline and use that to guide your de-optimization efforts, not just blind guessing. **The most fun part of this one is justifying each pessimization with "diabolical incompetence", not intentional malice.**

---

**Problems with the assignment wording and code**:

The uarch-specific options for this code are limited. It doesn't use any arrays, and much of the cost is calls to `exp`/`log` library functions. There isn't an obvious way to have more or less instruction-level parallelism, and the loop-carried dependency chain is very short.

It would be hard to get a slowdown just from re-arranging the expressions to change the dependencies, to reduce ILP from hazards.

Intel Sandybridge-family CPUs are aggressive out-of-order designs that spend lots of transistors and power to find parallelism and avoid hazards (dependencies) that would trouble a classic RISC in-order pipeline. Usually the only traditional hazards that slow it down are RAW "true" dependencies that cause throughput to be limited by latency.

**WAR and WAW hazards for registers are pretty much not an issue, thanks to register renaming**. (except for `popcnt`/`lzcnt`/`tzcnt`, which have a false dependency their destination on Intel CPUs, even though it should be write-only).

For memory ordering, modern CPUs use a store buffer to delay commit into cache until retirement, also avoiding WAR and WAW hazards. See also this answer about what a store buffer is, and being essential essential for OoO exec to decouple execution from things other cores can see.

Why does mulss take only 3 cycles on Haswell, different from Agner's instruction tables? (Unrolling FP loops with multiple accumulators) has more about register renaming and hiding FMA latency in an FP dot product loop.

---

**The "i7" brand-name was introduced with Nehalem (successor to Core2)**, and some Intel manuals even say Core i7 when they seem to mean Nehalem, but they kept the "i7" branding for Sandybridge and later microarchitectures. SnB is when the P6-family evolved into a new species, the SnB-family. In many ways, Nehalem has more in common with Pentium III than with Sandybridge (e.g. register read stalls aka ROB-read stalls don't happen on SnB, because it changed to using a physical register file. Also a uop cache and a different internal uop format). **The term "i7 architecture" is not useful**, because it makes little sense to group the SnB-family with Nehalem but not Core2. (Nehalem did introduce the shared inclusive L3 cache architecture for connecting multiple cores together, though. And also integrated GPUs. So chip-level, the naming makes more sense.)

---

## Summary of the good ideas that diabolical incompetence can justify

Even the diabolically incompetent are unlikely to add obviously useless work or an infinite loop, and making a mess with C++/Boost classes is beyond the scope of the assignment.

- Multi-thread with a single *shared* `std::atomic<uint64_t>` loop counter, so the right total number of iterations happen. Atomic uint64_t is especially bad with `-m32 -march=i586`. For bonus points, arrange for it to be misaligned, and crossing a page boundary with an uneven split (not 4:4).
- **False sharing** for some other non-atomic variable -> memory-order mis-speculation pipeline clears, as well as extra cache misses.
- Instead of using `-` on FP variables, XOR the high byte with 0x80 to flip the sign bit, causing **store-forwarding stalls**.
- Time each iteration independently, with something even heavier than `RDTSC`. e.g. `CPUID` / `RDTSC` or a time function that makes a system call. Serializing instructions are inherently pipeline-unfriendly.
- Change multiplies by constants to divides by their reciprocal ("for ease of reading"). **div is slow and not fully pipelined.**

- Vectorize the multiply/sqrt with AVX (SIMD), but fail to use `vzeroupper` before calls to scalar math-library `exp()` and `log()` functions, causing **AVX<->SSE transition stalls**.
- Store the RNG output in a linked list, or in arrays which you traverse out of order. Same for the result of each iteration, and sum at the end.

Also covered in this answer but excluded from the summary: suggestions that would be just as slow on a non-pipelined CPU, or that don't seem to be justifiable even with diabolical incompetence. e.g. many gimp-the-compiler ideas that produce obviously different / worse asm.

---

## Multi-thread badly

Maybe use OpenMP to multi-thread loops with very few iterations, with way more overhead than speed gain. Your monte-carlo code has enough parallelism to actually get a speedup, though, esp. if we succeed at making each iteration slow. (Each thread computes a partial `payoff_sum`, added at the end). `#omp parallel` on that loop would probably be an optimization, not a pessimization.

**Multi-thread but force both threads to share the same loop counter (with `atomic` increments so the total number of iterations is correct).** This seems diabolically logical. This means using a `static` variable as a loop counter. This justifies use of `atomic` for loop counters, and creates actual cache-line ping-ponging (as long as the threads don't run on the same physical core with hyperthreading; that might not be *as* slow). Anyway, this is *much* slower than the un-contended case for `lock xadd` or `lock dec`. And `lock cmpxchg8b` to atomically increment a contended `uint64_t` on a 32bit system will have to retry in a loop instead of having the hardware arbitrate an atomic `inc`.

Also create **false sharing**, where multiple threads keep their private data (e.g. RNG state) in different bytes of the same cache line. (Intel tutorial about it, including perf counters to look at). **There's a microarchitecture-specific aspect to this**: Intel CPUs speculate on memory mis-ordering *not* happening, and there's a memory-order machine-clear perf event to detect this, at least on P4. The penalty might not be as large on Haswell. As that link points out, a `lock`ed instruction assumes this will happen, avoiding mis-speculation. A normal load speculates that other cores won't invalidate a cache line between when the load executes and when it retires in program-order (unless you use `pause`). True sharing without `lock`ed instructions is usually a bug. It would be interesting to compare a non-atomic shared loop counter with the atomic case. To really pessimize, keep the shared atomic loop counter, and cause false sharing in the same or a different cache line for some other variable.

---

# Random uarch-specific ideas:

If you can introduce **any unpredictable branches**, that will pessimize the code substantially. Modern x86 CPUs have quite long pipelines, so a mispredict costs ~15 cycles (when running from the uop cache).

---

## Dependency chains:

I think this was one of the intended parts of the assignment.

Defeat the CPU's ability to exploit instruction-level parallelism by choosing an order of operations that has one long dependency chain instead of multiple short dependency chains. Compilers aren't allowed to change the order of operations for FP calculations unless you use `-ffast-math`, because that can change the results (as discussed below).

To really make this effective, increase the length of a loop-carried dependency chain. Nothing leaps out as obvious, though: The loops as written have very short loop-carried dependency chains: just an FP add. (3 cycles). Multiple iterations can have their calculations in-flight at once, because they can start well before the `payoff_sum +=` at the end of the previous iteration. (`log()` and `exp` take many instructions, but not a lot more than [Haswell's out-of-order window for finding parallelism: ROB size=192 fused-domain uops, and scheduler size=60 unfused-domain uops](). As soon as execution of the current iteration progresses far enough to make room for instructions from the next iteration to issue, any parts of it that have their inputs ready (i.e. independent/separate dep chain) can start executing when older instructions leave the execution units free (e.g. because they're bottlenecked on latency, not throughput.).

The RNG state will almost certainly be a longer loop-carried dependency chain than the `addps`.

---

## Use slower/more FP operations (esp. more division):

Divide by 2.0 instead of multiplying by 0.5, and so on. FP multiply is heavily pipelined in Intel designs, and has one per 0.5c throughput on Haswell and later. **FP `divsd` / `divpd` is only partially pipelined**. (Although Skylake has an impressive one per 4c throughput for `divpd xmm`, with 13-14c latency, vs not pipelined at all on Nehalem (7-22c)).

The `do { ...; euclid_sq = x*x + y*y;  } while (euclid_sq >= 1.0);` is clearly testing for a distance, so clearly it would be proper to `sqrt()` it. :P (`sqrt` is even slower than `div`).

As @Paul Clayton suggests, rewriting expressions with associative/distributive equivalents can introduce more work (as long as you don't use `-ffast-math` to allow the compiler to re-optimize). `(exp(T*(r-0.5*v*v))` could become `exp(T*r - T*v*v/2.0)`. Note that while math on real numbers is associative, floating point math is *not*, even without considering overflow/NaN (which is why `-ffast-math` isn't on by default). See Paul's comment for a very hairy nested `pow()` suggestion.

If you can scale the calculations down to very small numbers, then FP math ops take **~120 extra cycles to trap to microcode when an operation on two normal numbers produces a denormal**. See Agner Fog's microarch pdf for the exact numbers and details. This is unlikely since you have a lot of multiplies, so the scale factor would be squared and underflow all the way to 0.0. I don't see any way to justify the necessary scaling with incompetence (even diabolical), only intentional malice.

---

###If you can use intrinsics (`<immintrin.h>`)

Use `movnti` to evict your data from cache. Diabolical: it's new and weakly-ordered, so that should let the CPU run it faster, right? Or see that linked question for a case where someone was in danger of doing exactly this (for scattered writes where only some of the locations were hot). `clflush` is probably impossible without malice.

Use integer shuffles between FP math operations to cause bypass delays.

**Mixing SSE and AVX instructions without proper use of `vzeroupper` causes large stalls in pre-Skylake** (and a different penalty in Skylake). Even without that, vectorizing badly can be worse than scalar (more cycles spent shuffling data into/out of vectors than saved by doing the add/sub/mul/div/sqrt operations for 4 Monte-Carlo iterations at once, with 256b vectors). add/sub/mul execution units are fully pipelined and full-width, but div and sqrt on 256b vectors aren't as fast as on 128b vectors (or scalars), so the speedup isn't dramatic for `double`.

`exp()` and `log()` don't have hardware support, so that part would require extracting vector elements back to scalar and calling the library function separately, then shuffling the results back into a vector. libm is typically compiled to only use SSE2, so will use the legacy-SSE encodings of scalar math instructions. If your code uses 256b vectors and calls `exp` without doing a `vzeroupper` first, then you stall. After returning, an AVX-128 instruction like `vmovsd` to set up the next vector element as an arg for `exp` will also stall. And then `exp()` will stall again when it runs an SSE instruction. **This is exactly what happened in this question, causing a 10x slowdown.** (Thanks @ZBoson).

See also Nathan Kurz's experiments with Intel's math lib vs. glibc for this code. Future glibc will come with vectorized implementations of `exp()` and so on.

---

If targeting pre-IvB, or esp. Nehalem, try to get gcc to cause partial-register stalls with 16bit or 8bit operations followed by 32bit or 64bit operations. In most cases, gcc will use `movzx` after an 8 or 16bit operation, but here's a case where gcc modifies `ah` and then reads `ax`

---

## With (inline) asm:

With (inline) asm, you could break the uop cache: A 32B chunk of code that doesn't fit in three 6uop cache lines forces a switch from the uop cache to the decoders. An incompetent `ALIGN` (like NASM's default) using many single-byte `nop`s instead of a couple long `nop`s on a branch target inside the inner loop might do the trick. Or put the alignment padding after the label, instead of before. :P This only matters if the frontend is a bottleneck, which it won't be if we succeeded at pessimizing the rest of the code.

Use self-modifying code to trigger pipeline clears (aka machine-nukes).

LCP stalls from 16bit instructions with immediates too large to fit in 8 bits are unlikely to be useful. The uop cache on SnB and later means you only pay the decode penalty once. On Nehalem (the first i7), it might work for a loop that doesn't fit in the 28 uop loop buffer. gcc will sometimes generate such instructions, even with `-mtune=intel` and when it could have used a 32bit instruction.

---

A common idiom for timing is `CPUID` (to serialize) then `RDTSC`. Time every iteration separately with a `CPUID`/`RDTSC` to make sure the `RDTSC` isn't reordered with earlier instructions, which will slow things down a *lot*. (In real life, the smart way to time is to time all the iterations together, instead of timing each separately and adding them up).

---

## Cause lots of cache misses and other memory slowdowns

Use a `union { double d; char a[8]; }` for some of your variables. **Cause a store-forwarding stall** by doing a narrow store (or Read-Modify-Write) to just one of the bytes. (That wiki article also covers a lot of other microarchitectural stuff for load/store queues). e.g. **flip the sign of a `double` using XOR 0x80 on just the high byte**, instead of a `-` operator. The diabolically incompetent developer may have heard that FP is slower than integer, and thus try to do as much as possible using integer ops. (A compiler could theoretically still compile this to an `xorps` with a constant like `-`, but for x87 the compiler would have to realize that it's negating the value and `fchs` or replace the next add with a subtract.)

---

Use `volatile` if you're compiling with `-O3` and not using `std::atomic`, to force the compiler to actually store/reload all over the place. Global variables (instead of locals) will also force some stores/reloads, but the C++ memory model's weak ordering doesn't require the compiler to spill/reload to memory all the time.

**Replace local vars with members of a big struct, so you can control the memory layout.**

Use arrays in the struct for padding (and storing random numbers, to justify their existence).

Choose your memory layout so everything goes into a different line in the same "set" in the L1 cache. It's only 8-way associative, i.e. each set has 8 "ways". Cache lines are 64B.

Even better, **put things exactly 4096B apart, since loads have a false dependency on stores to different pages but with the same offset within a page**. Aggressive out-of-order CPUs use Memory Disambiguation to figure out when loads and stores can be reordered without changing the results, and Intel's implementation has false-positives that prevent loads from starting early. Probably they only check bits below the page offset so it can start before the TLB has translated the high bits from a virtual page to a physical page. As well as Agner's guide, see this answer, and a section near the end of @Krazy Glew's answer on the same question. (Andy Glew was an architect of Intel's PPro - P6 microarchitecture.) (Also related: https://stackoverflow.com/a/53330296 and https://github.com/travisdowns/uarch-bench/wiki/Memory-Disambiguation-on-Skylake)

Use `__attribute__((packed))` to let you mis-align variables so they span cache-line or even page boundaries. (So a load of one `double` needs data from two cache-lines). Misaligned loads have no penalty in any Intel i7 uarch, except when crossing cache lines and page lines. Cache-line splits still take extra cycles. Skylake dramatically reduces the penalty for page split loads, from 100 to 5 cycles. (Section 2.1.3). (And can do two page walks in parallel).

**A page-split on an `atomic<uint64_t>` should be just about the worst case**, esp. if it's 5 bytes in one page and 3 bytes in the other page, or anything other than 4:4. Even splits down the middle are more efficient for cache-line splits with 16B vectors on some uarches, IIRC. Put everything in a `alignas(4096) struct __attribute((packed))` (to save space, of course), including an array for storage for the RNG results. Achieve the misalignment by using `uint8_t` or `uint16_t` for something before the counter.

If you can get the compiler to use indexed addressing modes, that will defeat uop micro-fusion. Maybe by using `#define`s to replace simple scalar variables with `my_data[constant]`.

If you can introduce an extra level of indirection, so load/store addresses aren't known early, that can pessimize further.

---

## Traverse arrays in non-contiguous order

I think we can come up with incompetent justification for introducing an array in the first place: It lets us separate the random number generation from the random number use. Results of each iteration could also be stored in an array, to be summed later (with more diabolical incompetence).

For "maximum randomness", we could have a thread looping over the random array writing new random numbers into it. The thread consuming the random numbers could generate a random index to load a random number from. (There's some make-work here, but microarchitecturally it helps for load-addresses to be known early so any possible load latency can be resolved before the loaded data is needed.) Having a reader and writer on different cores will cause memory-ordering mis-speculation pipeline clears (as discussed earlier for the false-sharing case).

For maximum pessimization, loop over your array with a stride of 4096 bytes (i.e. 512 doubles). e.g.

```
for (int i=0 ; i<512; i++)
    for (int j=i ; j<UPPER_BOUND ; j+=512)
        monte_carlo_step(rng_array[j]);
```

So the access pattern is 0, 4096, 8192, ...,
8, 4104, 8200, ...
16, 4112, 8208, ...

This is what you'd get for accessing a 2D array like `double rng_array[MAX_ROWS][512]` in the wrong order (looping over rows, instead of columns within a row in the inner loop, as suggested by @JesperJuhl). If diabolical incompetence can justify a 2D array with dimensions like that, garden variety real-world incompetence easily justifies looping with the wrong access pattern. This happens in real code in real life.

Adjust the loop bounds if necessary to use many different pages instead of reusing the same few pages, if the array isn't that big. Hardware prefetching doesn't work (as well/at all) across pages. The prefetcher can track one forward and one backward stream within each page (which is what happens here), but will only act on it if the memory bandwidth isn't already saturated with non-prefetch.

This will also generate lots of TLB misses, unless the pages get merged into a hugepage (Linux does this opportunistically for anonymous (not file-backed) allocations like `malloc`/`new` that use `mmap(MAP_ANONYMOUS)`).

Instead of an array to store the list of results, you could use a **linked list**. Every iteration would require a pointer-chasing load (a RAW true dependency hazard for the load-address of the next load). With a bad allocator, you might manage to scatter the list nodes around in memory, defeating cache. With a bad toy allocator, it could put every node at the beginning of its own page. (e.g. allocate with `mmap(MAP_ANONYMOUS)` directly, without breaking up pages or tracking object sizes to properly support `free`).

---

These aren't really microarchitecture-specific, and have little to do with the pipeline (most of these would also be a slowdown on a non-pipelined CPU).

## Somewhat off-topic: make the compiler generate worse code / do more work:

Use C++11 `std::atomic<int>` and `std::atomic<double>` for the most pessimal code. The MFENCEs and `lock`ed instructions are quite slow even without contention from another thread.

`-m32` will make slower code, because x87 code will be worse than SSE2 code. The stack-based 32bit calling convention takes more instructions, and passes even FP args on the stack to functions like `exp()`. `atomic<uint64_t>::operator++` on `-m32` requires a `lock cmpxchg8B` loop (i586). (So use that for loop counters! [Evil laugh]).

`-march=i386` will also pessimize (thanks @Jesper). FP compares with `fcom` are slower than 686 `fcomi`. Pre-586 doesn't provide an atomic 64bit store, (let alone a cmpxchg), so all 64bit `atomic` ops compile to libgcc function calls (which is probably compiled for i686, rather than actually using a lock). Try it on the Godbolt Compiler Explorer link in the last paragraph.

Use `long double` / `sqrtl` / `expl` for extra precision and extra slowness in ABIs where sizeof(`long double`) is 10 or 16 (with padding for alignment). (IIRC, 64bit Windows uses 8byte `long double` equivalent to `double`. (Anyway, load/store of 10byte (80bit) FP operands is 4 / 7 uops, vs. `float` or `double` only taking 1 uop each for `fld m64/m32`/`fst`). Forcing x87 with `long double` defeats auto-vectorization even for gcc `-m64 -march=haswell -O3`.

If not using `atomic<uint64_t>` loop counters, use `long double` for everything, including loop counters.

`atomic<double>` compiles, but read-modify-write operations like `+=` aren't supported for it (even on 64bit). `atomic<long double>` has to call a library function just for atomic loads/stores. It's probably really inefficient, because the x86 ISA doesn't naturally support atomic 10byte loads/stores, and the only way I can think of without locking (`cmpxchg16b`) requires 64bit mode.

---

At `-O0`, breaking up a big expression by assigning parts to temporary vars will cause more store/reloads. Without `volatile` or something, this won't matter with optimization settings that a real build of real code would use.

C aliasing rules allow a `char` to alias anything, so storing through a `char*` forces the compiler to store/reload everything before/after the byte-store, even at `-O3`. (This is a problem for auto-vectorizing code that operates on an array of `uint8_t`, for example.)

Try `uint16_t` loop counters, to force truncation to 16bit, probably by using 16bit operand-size (potential stalls) and/or extra `movzx` instructions (safe). Signed overflow is undefined behaviour, so unless you use `-fwrapv` or at least `-fno-strict-overflow`, signed loop counters don't have to be re-sign-extended every iteration, even if used as offsets to 64bit pointers.

---

Force conversion from integer to `float` and back again. And/or `double`<=>`float` conversions. The instructions have latency > 1, and scalar int->float (`cvtsi2ss`) is badly designed to not zero the rest of the xmm register. (gcc inserts an extra `pxor` to break dependencies, for this reason.)

---

Frequently **set your CPU affinity to a different CPU** (suggested by @Egwor). diabolical reasoning: You don't want one core to get overheated from running your thread for a long time, do you? Maybe swapping to another core will let that core turbo to a higher clock speed. (In reality: they're so thermally close to each other that this is highly unlikely except in a multi-socket system). Now just get the tuning wrong and do it way too often. Besides the time spent in the OS saving/restoring thread state, the new core has cold L2/L1 caches, uop cache, and branch predictors.

Introducing frequent unnecessary system calls can slow you down no matter what they are. Although some important but simple ones like `gettimeofday` may be implemented in user-space with, with no transition to kernel mode. (glibc on Linux does this with the kernel's help: the kernel exports code+data in the VDSO).

For more on system call overhead (including cache/TLB misses after returning to user-space, not just the context switch itself), the FlexSC paper has some great perf-counter analysis of the current situation, as well as a proposal for batching system calls from massively multi-threaded server processes.

Share
Improve this answer
Follow

edited Jul 27, 2022 at 17:43

answered May 21, 2016 at 11:17
Peter Cordes
361k ● 49 ● 699 ● 958

---

12 @JesperJuhl: yeah, I'll buy that justification. "diabolically incompetent" is such a wonderful phrase :) – Peter Cordes May 21, 2016 at 12:31

26 Some of those suggestions are so diabolically incompetent that I have to talk to the professor to see if the now 7 minute running time is too much for him to want to sit through to verify the output. Still working with this, this has probably been the most fun I've had with a project. – Cowmoogun May 22, 2016 at 22:21

5 What? No mutexes? Having two million threads running simultaneously with a mutex protecting each and every individual computation (just in case!) would bring the fastest supercomputer on the planet to its knees. That said, I do love this diabolically incompetent answer. – David Hammen May 22, 2016 at 23:55

3 @Nicholas: I basically went through the list of possible stalls (from Agner Fog's microarch pdf), and considered how to diabolically justify introducing them into this code. – Peter Cordes May 26, 2016 at 19:43

6 This post now comes up first when I google "*diabolically incompetent*" :) – rustyx Aug 6, 2016 at 20:26

37

A few things that you can do to make things perform as bad as possible:

- compile the code for the i386 architecture. This will prevent the use of SSE and newer instructions and force the use of the x87 FPU.
- use `std::atomic` variables everywhere. This will make them very expensive due to the compiler being forced to insert memory barriers all over the place. And this is something an incompetent person might plausibly do to "ensure thread safety".
- make sure to access memory in the worst possible way for the prefetcher to predict (column major vs row major).
- to make your variables extra expensive you could make sure they all have 'dynamic storage duration' (heap allocated) by allocating them with `new` rather than letting them have 'automatic storage duration' (stack allocated).
- make sure that all memory you allocate is very oddly aligned and by all means avoid allocating huge pages, since doing so would be much too TLB efficient.
- whatever you do, don't build your code with the compilers optimizer enabled. And make sure to enable the most expressive debug symbols you can (won't make the code *run* slower, but it'll waste some extra disk space).

Note: This answer basically just summarizes my comments that @Peter Cordes already incorporated into his very good answer. Suggest he get's your upvote if you only have one to spare :)

Share
Improve this answer
Follow

edited May 13, 2019 at 16:29

answered May 21, 2016 at 12:56
Jesper Juhl
31.3k ● 3 ● 52 ● 76

10 My main objection to some of these are the phrasing of the question: *To deoptimize the program, **use your knowledge of how the Intel i7 pipeline operates***. I don't feel like there's anything uarch-specific about x87, or `std::atomic`, or an extra level of indirection from dynamic allocation. They're going to be slow on an Atom or K8 as well. Still upvoting, but that's why I was resisting some of your suggestions. – Peter Cordes May 21, 2016 at 14:03

Those are fair points. Regardless, those things still work towards the goal of the asker somewhat. Appreciate the upvote :) – Jesper Juhl May 21, 2016 at 14:08

The SSE unit use ports 0, 1 and 5. The x87 unit use only ports 0 and 1. – Michas May 23, 2016 at 8:49

@Michas: You're wrong about that. Haswell doesn't run any SSE FP math instructions on port 5. Mostly SSE FP shuffles and booleans (xorps/andps/orps). x87 is slower, but your explanation of why is slightly wrong. (And this point is completely wrong.) – Peter Cordes May 28, 2016 at 3:52

1 @Michas: `movapd xmm, xmm` usually doesn't need an execution port (it's handled at register-rename stage on IVB and later). It's also almost never needed in AVX code, because everything but FMA is non-destructive. But fair enough, Haswell runs it on port5 if it's not eliminated. I hadn't looked at x87 register-copy (`fld st(i)`), but you're right for Haswell/Broadwell: it runs on p01. Skylake runs it on p05, SnB runs it on p0, IvB run it on p5. So IVB / SKL do some x87 stuff (including compare) on p5, but SNB/HSW/BDW don't use p5 at all for x87. – Peter Cordes May 29, 2016 at 1:57

11

You can use `long double` for computation. On x86 it should be the 80-bit format. Only the legacy, x87 FPU has support for this.

Few shortcomings of x87 FPU:

1. Lack of SIMD, may need more instructions.
2. Stack based, problematic for super scalar and pipelined architectures.
3. Separate and quite small set of registers, may need more conversion from other registers and more memory operations.
4. On the Core i7 there are 3 ports for SSE and only 2 for x87, the processor can execute less parallel instructions.

Share
Improve this answer
Follow

edited May 23, 2016 at 8:54

answered May 21, 2016 at 20:36
Michas
9,368 ● 6 ● 40 ● 63

3 For scalar math, x87 math instructions themselves are only slightly slower. Storing / loading 10byte operands is significantly slower, though, and x87's stack-based design tends to require extra instructions (like `fxch`). With `-ffast-math`, a good compiler might vectorize the monte-carlo loops, though, and x87 would prevent that. – Peter Cordes May 22, 2016 at 10:04

I have extended my answer a little. – Michas May 22, 2016 at 13:01

1 re:4: Which i7 uarch are you talking about, and which instructions? Haswell can run `mulss` on p01, but `fmul` only on `p0`. `addss` only runs on `p1`, same as `fadd`. There are only two execution ports that handle FP math ops. (The only exception to this is that Skylake dropped the dedicated add unit and runs `addss` in the FMA units on p01, but `fadd` on p5. So by mixing in some `fadd` instructions along with `fma...ps`, you can in theory do slightly more total FLOP/s.) – Peter Cordes May 23, 2016 at 15:10

2 Also note that the Windows x86-64 ABI has 64bit `long double`, i.e. it's still just `double`. The SysV ABI does use 80bit `long double`, though. Also, re:2: register renaming exposes the parallelism in the stack registers. The stack-based architecture requires some extra instructions, like `fxchg`, esp. when interleaving parallel calculations. So it's more like it's hard to express parallelism without memory round-trips, rather than it's hard for the uarch to exploit what is there. You don't need more conversion from other regs, though. Not sure what you mean by that. – Peter Cordes May 23, 2016 at 15:17

6

Late answer but I don't feel we have abused linked lists and the TLB enough.

Use mmap to allocate your nodes, such that your mostly use the MSB of the address. This should result in long TLB lookup chains, a page is 12 bits, leaving 52 bits for the translation, or around 5 levels it must travers each time. With a bit of luck they must go to memory each time for 5 levels lookup plus 1 memory access to get to your node, the top level will most likely be in cache somewhere, so we can hope for 5*memory access. Place the node so that is strides the worst border so that reading the next pointer would cause another 3-4 translation lookups. This might also totally wreck the cache due to the massive amount of translation lookups. Also the size of the virtual tables might cause most of the user data to be paged to disk for extra time.

When reading from the single linked list, make sure to read from the start of the list each time to cause maximum delay in reading a single number.

Share Improve this answer Follow

answered Feb 5, 2017 at 22:34

Surt
16.1k ● 3 ● 26 ● 40

x86-64 page tables are 4 levels deep for 48-bit virtual addresses. (A PTE has 52 bits of physical address). Future CPUs will support a 5-level page table feature, for another 9 bits of virtual address space (57). [Why in 64bit the virtual address are 4 bits short (48bit long) compared with the physical address (52 bit long)?](). OSes won't enable it by default because

it'd be slower and brings no benefit unless you need that much virt address space.
– Peter Cordes Dec 3, 2019 at 5:34

But yes, fun idea. You could maybe use `mmap` on a file or shared-memory region to get multiple virtual addresses for the same physical page (with the same contents), allowing more TLB misses over the same amount of physical RAM. If your linked-list's `next` was just a relative *offset*, you could have a series of mappings of the same page with a `+4096 * 1024` until you finally get to a different physical page. Or of course spanning over multiple pages to avoid L1d cache hits. There is caching of higher-level PDEs within page-walk hardware, so yes spread it out in virt addr space! – Peter Cordes Dec 3, 2019 at 5:40

Adding an offset to the old address also makes load-use latency worse by defeating [the special case for a `[reg+small_offset]` addressing mode](Is there a penalty when base+offset is in a different page than the base?); you'd either get a memory-source `add` of a 64-bit offset, or you'd get a load and an indexed addressing mode like `[reg+reg]`. Also see What happens after a L2 TLB miss? - page walk fetches through L1d cache on SnB-family. – Peter Cordes Dec 3, 2019 at 6:07