ECE459: Programming for Performance Winter 2024

Lecture 17 — Mostly Data Parallelism

Patrick Lam *2025-06-01*

Data and Task Parallelism

There are two broad categories of paralellism: data parallelism and task parallelism. An analogy to data parallelism is hiring a call center to (incompetently) handle large volumes of support calls, *all in the same way*. Assembly lines are an analogy to task parallelism: each worker does a *different* thing.

More precisely, in data parallelism, multiple threads perform the *same* operation on separate data items. For instance, you have a big array and want to double all of the elements. Assign part of the array to each thread. Each thread does the same thing: double array elements.

In task parallelism, multiple threads perform *different* operations on separate data items. So you might have a thread that renders frames and a thread that compresses frames and combines them into a single movie file.

You're not using those bytes, are you?

So as a first idea we might think of saving some space by considering the range of, for example, an integer. An `i32` is 4 bytes. (In C, `int` is usually 4, though only guaranteed to be at least 2). If we have an integer array of capacity N that uses $N \times 4$ bytes and if we want to do something like increment each element, we iterate over the array and increment it, which is a read of 4 and write of 4. Now, if we could live with limiting our maximum value from 2,147,483,647 (signed, or 4,294,967,295 unsigned) to 32,767 (signed, or 65,535 unsigned), we could reduce in half the amount of space needed for this array and make operations like incrementing take half as much time!

Aside from the obvious tradeoff of limiting the maximum value, the other hidden cost is that of course things that were simple like `array[i] += 1` is more complicated. What do we do now?

Instead of `+=1` we need to calculate the new number to add. The interesting part is about how to represent the upper portion of the number. For just adding 1 it might be simple, and we can manually break out our calculators or draw a bit vector or think in hexadecimal about how to convert a number if it's more difficult. But you wouldn't—you would probably just use bit shift to calculate it. But one must be careful with that as well: the bit shift does sign extension which sometimes you don't want (or does unexpected things), and if we have to bit shift on every iteration of the loop, it's not clear that this is better than two assignment statements...

Maybe you think this example is silly because of Rust's `i8`/C's `short` types. Which you could certainly use to reduce the size of the array. But then modifying each `short` in a different instruction defeats the purpose.

Aha! We can also take it a step farther: if it's a 64-bit processor there's no reason why you couldn't modify 8 bytes in a single instruction. The principle is the same, even if the math is a little more complex.

What we've got here is a poor person's version of Single Instruction Multiple Data (SIMD) (in NZ-speak, using No. 8 wire to implement SIMD), because we have to do our own math in advance and/or do a lot of bit shifting every time we want to use a value... This is a pain. Fortunately, we don't have to...

Data Parallelism with SIMD

The "typical" boring standard uniprocessor is Single Instruction Single Data (SISD) but since the mid-1980s we've had more options than that. We'll talk about single-instruction multiple-data (SIMD) later on in this course, but

here's a quick look. Each SIMD instruction operates on an entire vector of data. These instructions originated with supercomputers in the 70s. More recently, GPUs; the x86 SSE instructions; the SPARC VIS instructions; and the Power/PowerPC AltiVec instructions all implement SIMD.

SIMD provides an advantage by using a single control unit to command multiple processing units and therefore the amount of overhead in the instruction stream. This is something that we do quite frequently in the everyday: if I asked someone to erase the board[1], it's more efficient if I say "erase these segments of the board" (and clearly indicate which segments) than if I say "erase this one" and when that's done, then say "erase that one"... and so on. So we can probably get some performance benefit out of this!

There is the downside, though, that because there's only the one control unit, all the processing units are told to do the same thing. That might not be what you want, so SIMD is not something we can use in every situation. There are also diminishing returns: the more processing units you have, the less likely it is that you can use all of that power effectively (because it will be less likely to have enough identical operations) [Ton09].

Compilation. Let's look at an example of SIMD instructions when they are compiled.

By default your compiler will assume a particular target architecture; which one exactly is dependent on what the Rust team decided some time in the past. Choosing a too-new architecture will cause your code to fail on older machines. The choice of architecture can be overridden in your compile-time options with the `target` parameter. Let's look at some SSE code to add two slices and put the result in a third slice:

```
pub fn foo(a: &[f64], b: &[f64], c: &mut [f64]) {
    for ((a, b), c) in a.iter().zip(b).zip(c) {
        *c = *a + *b;
    }
}
```

We can compile with `rustc` defaults and get something like this as core loop contents:

```
movsd   xmm0, qword ptr [rcx]
addsd   xmm0, qword ptr [rdx]
movsd   qword ptr [rax], xmm0
```

This uses the SSE[2] register `xmm0` and SSE2 instructions `movsd` and `addsd`; the `sd` suffix denotes scalar double instructions, applying only to the first 64 bits of the 128-bit `xmm0` register—this is a literal translation of the code. If you additionally specify `-O`, the compiler generates a number of variants, including this middle one:

```
movupd  xmm0, xmmword ptr [rdi + 8*rcx]
movupd  xmm1, xmmword ptr [rdi + 8*rcx + 16]
movupd  xmm2, xmmword ptr [rdx + 8*rcx]
addpd   xmm2, xmm0
movupd  xmm0, xmmword ptr [rdx + 8*rcx + 16]
addpd   xmm0, xmm1
movupd  xmmword ptr [r8 + 8*rcx], xmm2
movupd  xmmword ptr [r8 + 8*rcx + 16], xmm0
```

The *packed* operations (`p`) operate on multiple data elements at a time (what kind of parallelism is this?) The implication is that the loop only needs to loop half as many times. The compiler includes more variants, not shown, to handle cases where there are odd numbers of elements in the slices.

So this is a piece of good news, for once: there's automatic use of the SSE instructions if your compiler knows the target machine architecture supports them. However, we can also explicitly invoke these instructions, or use

[1] Classrooms. How 2019.

[2] You can also compile without SIMD using `-target=i586-unknown-linux-gnu` and see the stack-based x87 instructions.

libraries[3], although we won't do that much. Instead, we'll learn more about how they work and then do some measurement as to whether they really do.

SIMD is different from the other types of parallelization we're looking at, since there aren't multiple threads working at once. It is complementary to using threads, and good for cases where loops operate over vectors of data. These loops could also be parallelized; multicore chips can do both, achieving high throughput. SIMD instructions also work well on small data sets, where thread startup cost is too high, while registers are just there.

In [Lem18], Daniel Lemire argues that vector instructions are, in general, a more efficient way to parallelize code than threads. That is, when applicable, they use less overall CPU resources (cores and power) and run faster.

Rust will generally align primitives to their sizes. On some architectures, this may have performance implications (ARM Cortex), but perhaps not on x86 since you were in elementary school. Under the default representation, Rust promises nothing else about alignment. You can use the `repr(packed(N)` or `repr(align(N)` directives to express constraints on alignment, and you can specify the C representation, which allows you more control over data layout.

Worked Example. So let's say that you actually wanted to try it out. Let's consider a `simdeez` example, which I've put in the repo's live coding subdir under `lectures/live-coding/L16`.

```
use simdeez::*;
use simdeez::scalar::*;
use simdeez::sse2::*;
use simdeez::sse41::*;
use simdeez::avx2::*;

simd_runtime_generate!(
// assumes that the input sizes are evenly divisible by VF32_WIDTH
pub fn add(a:&[f32], b: &[f32]) -> Vec<f32> {
  let len = a.len();
  let mut result: Vec<f32> = Vec::with_capacity(len);
  result.set_len(len);
  for i in (0..len).step_by(S::VF32_WIDTH) {
    let a0 = S::loadu_ps(&a[i]);
    let b0 = S::loadu_ps(&b[i]);
    S::storeu_ps(&mut result[0], S::add_ps(a0, b0));
  }
  result
});

fn main() {
  let a : [f32; 4] = [1.0, 2.0, 3.0, 4.0];
  let b : [f32; 4] = [5.0, 6.0, 7.0, 8.0];

  unsafe {
    println!("{:?}", add_sse2(&a, &b))
  }
}
```

What this does is generate an `add_*` function for each of `scalar`, `sse2`, `sse41`, and `avx`. Then `main` unsafely calls `add_sse2` with two length-4 arrays of `f32`s and gets a `Vec<f32>` back.

`simdeez` is a fairly lightweight wrapper around SIMD instructions and just calls the `loadu_ps` and `storeu_ps` calls to load and store packed single-precision numbers, and `add_ps` to add them. Operator overloading works too.

[3] A discussion of libraries available as of May 2020: `https://www.mdeditor.tw/pl/pdnr`; your choices are `packed_simd` (nightly Rust only), `faster` (unmaintained), or `simdeez` (must use unsafe Rust).

Case Study on SIMD: Stream VByte

"Can you run faster just by trying harder?"

The performance improvements we've seen to date have been leveraging parallelism to improve throughput. Decreasing latency is trickier—it often requires domain-specific tweaks.

Sometimes it's classic computer science: Quantum Flow found a place where they could cache the last element of a list to reduce time complexity for insertion from $O(n^2)$ to $O(n \log n)$.

`https://bugzilla.mozilla.org/show_bug.cgi?id=1350770`

We'll also look at a more involved example of decreasing latency today, Stream VByte [LKR18], and briefly at parts of its C++ implementation. Even this example leverages parallelism—it uses vector instructions. But there are some sequential improvements, e.g. Stream VByte takes care to be predictable for the branch predictor.

Context. We can abstract the problem to that of storing a sequence of small integers. Such sequences are important, for instance, in the context of inverted indexes, which allow fast lookups by term, and support boolean queries which combine terms.

Here is a list of documents and some terms that they contain:

docid	terms
1	dog, cat, cow
2	cat
3	dog, goat
4	cow, cat, goat

The inverted index looks like this:

term	docs
dog	1, 3
cat	1, 2, 4
cow	1, 4
goat	3, 4

Inverted indexes contain many small integers in their lists: it is sufficient to store the delta between a doc id and its successor, and the deltas are typically small if the list of doc ids is sorted. (Going from deltas to original integers takes time logarithmic in the number of integers).

VByte is one of a number of schemes that use a variable number of bytes to store integers. This makes sense when most integers are small, and especially on today's 64-bit processors.

VByte works like this:

- x between 0 and $2^7 - 1$, e.g. $17 = 0b10001$: $0xxxxxxx$, e.g. 00010001;
- x between 2^7 and $2^{14} - 1$, e.g. $1729 = 0b11011000001$: $1xxxxxxx/0xxxxxxx$, e.g. $11000001/00001101$;
- x between 2^{14} and $2^{21} - 1$: $1xxxxxxx/1xxxxxxx/0xxxxxxx$;
- etc.

That is, the control bit, or high-order bit, is 0 if you have finished representing the integer, and 1 if more bits remain. (UTF-8 encodes the length, from 1 to 4, in high-order bits of the first byte.)

It might seem that dealing with variable-byte integers might be harder than dealing fixed-byte integers, and it is. But there are performance benefits: because we are using fewer bits, we can fit more information into our limited RAM and cache, and even get higher throughput. Storing and reading 0s isn't an effective use of resources. However, a naive algorithm to decode VByte also gives lots of branch mispredictions.

Stream VByte is a variant of VByte which works using SIMD instructions. Science is incremental, and Stream VByte builds on earlier work—masked VByte as well as VARINT-GB and VARINT-G8IU. The innovation in Stream VByte is to store the control and data streams separately.

Stream VByte's control stream uses two bits per integer to represent the size of the integer:

00	1 byte	10	3 bytes
01	2 bytes	11	4 bytes

Each decode iteration reads a byte from the control stream and 16 bytes of data from memory. It uses a lookup table over the possible values of the control stream to decide how many bytes it needs out of the 16 bytes it has read, and then uses SIMD instructions to shuffle the bits each into their own integers. Note that, unlike VByte, Stream VByte uses all 8 bits of each data byte as data.

For instance, if the control stream contains $0b1000\ 1100$, then the data stream contains the following sequence of integer sizes: $3, 1, 4, 1$. Out of the 16 bytes read, this iteration will use 9 bytes; it advances the data pointer by 9. It then uses the SIMD "shuffle" instruction to put the decoded integers from the data stream at known positions in the 128-bit SIMD register; in this case, it pads the first 3-byte integer with 1 byte, then the next 1-byte integer with 3 bytes, etc. Let's say that the input is `0xf823 e127 2524 9748 1b.. .... .... .....`. The 128-bit output is `0x00f8 23e1/0000 0027/2524 9748/0000 001b`, with the */s* denoting separation between outputs. The shuffle mask is precomputed and, at execution time, read from an array.

The core of the (C++) implementation uses three SIMD instructions (also available in `simdeez`):

```
uint8_t C = lengthTable[control];
__m128i Data = _mm_loadu_si128 ((__m128i *) databytes);
__m128i Shuf = _mm_loadu_si128(shuffleTable[control]);
Data = _mm_shuffle_epi8(Data, Shuf);
databytes += C; control++;
```

Discussion. The paper [LKR18] includes a number of benchmark results showing how Stream VByte performs better than previous techniques on a realistic input. Let's discuss how it achieves this performance.

- control bytes are sequential: the processor can always prefetch the next control byte, because its location is predictable;
- data bytes are sequential and loaded at high throughput;
- shuffling exploits the instruction set so that it takes 1 cycle;
- control-flow is regular (executing only the tight loop which retrieves/decodes control and data; there are no conditional jumps).

We're exploiting SIMD, so this isn't quite strictly single-threaded performance. Considering branch prediction and caching issues, though, certainly improves single-threaded performance.

SIMD and Planetary Motion

At the moment, I'm not planning to cover this, but you can read more about SIMD in Rust here:

`https://medium.com/@Razican/learning-simd-with-rust-by-finding-planets-b85ccfb724c3`

References

[Lem18] Daniel Lemire. Multicore versus SIMD instructions: the "fasta" case study, 2018. Online; accessed 03-January-2018. URL: https://lemire.me/blog/2018/01/02/multicore-versus-simd-instructions-the-fasta-case-study/.

[LKR18] Daniel Lemire, Nathan Kurz, and Christoph Rupp. Stream VByte: Faster byte-oriented integer compression. *Information Processing Letters*, 130(Supplement C):1 – 6, 2018. URL: http://www.sciencedirect.com/science/article/pii/S0020019017301679.

[Ton09] Tuomas Tonteri. A practical guide to SSE SIMD with c++, 2009. Online; accessed 2019-12-08. URL: http://sci.tuomastonteri.fi/programming/sse.