

# Lecture 16 — Mostly Data Parallelism

Patrick Lam

`patrick.lam@uwaterloo.ca`

Department of Electrical and Computer Engineering  
University of Waterloo

September 12, 2024

**Data parallelism** is performing *the same* operations on different input.

**Example:** doubling all elements of an array.

**Task parallelism** is performing *different* operations on different input.

**Example:** playing a video file: one thread decompresses frames, another renders.

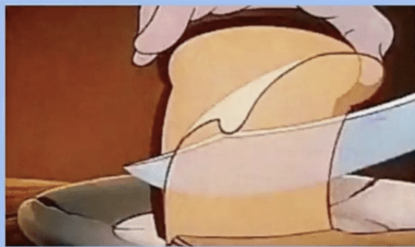
# You're Not Using Those Bytes, Are You?

Consider the size of an `i32`... 4 bytes? At least 2...

Array of capacity  $N$ ? That uses  $N \times 4$  bytes.

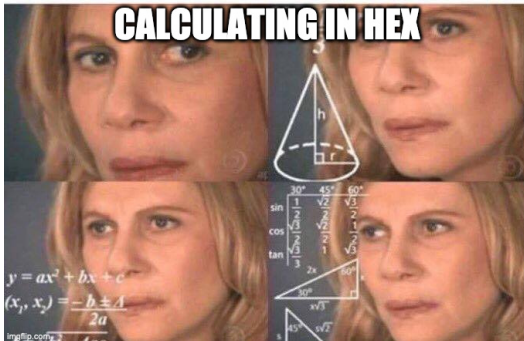
Can we limit the size of the integer? Is 65,535 enough?

When your mom says you have to share something with your sibling



This used to be easier...

The other hidden cost is that of course things that were simple, like `array[i] += 1`, are now more complicated.



What do we do now?

Instead of just  $+=1$  we need to calculate the new number to add.

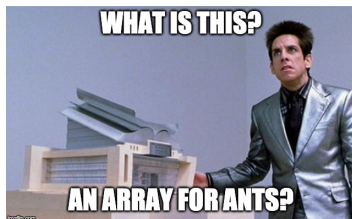
The interesting part is about how to represent the upper portion of the number.

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.

# Don't You Forget About Me

Maybe you think this example is silly because of Rust's `i8/C`'s short.

You can use this to reduce the size of the array.



But then modifying each short in a different instruction defeats the purpose.

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 version of Single Instruction Multiple Data (SIMD)...

# Data Parallelism: Single Instruction, Multiple Data

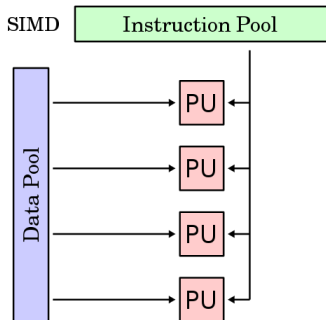
SIMD, an overview:

- You can load a bunch of data and perform arithmetic.
- Instructions process multiple data items simultaneously. (Exact number is hardware-dependent).

For x86-class CPUs, MMX and SSE extensions provide SIMD instructions.



SIMD provides an advantage by using a single control unit to command multiple processing units.



Example: consider I ask people to erase boards in class...

Only works if we are all erasing!

Consider the following code:

---

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

---

In this scenario, we have a regular operation over block data.

We could use threads, but we'll use SIMD.

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 and SSE2 instructions

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.

The compiler uses SIMD instructions if the target architecture supports it.

We can explicitly invoke the instructions, or use libraries

It is complementary to using threads, and good for cases where loops operate over vectors of data.

SIMD instructions also work well on small data sets, where thread startup cost is too high.

(+) A more efficient (= less CPU resources) way to parallelize code than threads.

(-) Data must be 16-byte aligned when loading to/storing from 128-bit registers.  
Not required in SSE4.2... if your CPU supports it (all modern x86s do,  
but maybe not ARM Cortex).

 <p><b>LAWFUL GOOD</b></p> <p>Many see him as a naive boy scout whipped by his own selflessness. They will not, cannot, see him for what he is, a hero.</p>	 <p><b>NEUTRAL GOOD</b></p> <p>A good tonic, ultrium. Nothing helps one put problems in perspective like allegiance to a higher cause.</p>	 <p><b>CHAOTIC GOOD</b></p> <p>No rest. No mercy. No matter what.</p>
 <p><b>LAWFUL NEUTRAL</b></p> <p>I am the state.</p>	 <p><b>TRUE NEUTRAL</b></p> <p>I don't expect you to understand. It's how we should exist. How we were meant to exist.</p>	 <p><b>CHAOTIC NEUTRAL</b></p> <p>I'm not gonna split hairs and I'm not gonna fight my teammates. I mean, unless it involves Tenny's clothes coming off and mud or chocolate pudding or something like that.</p>
 <p><b>LAWFUL EVIL</b></p> <p>Slaves would be tyrants, were the chance theirs.</p>	 <p><b>NEUTRAL EVIL</b></p> <p>Do you really think fate turned us into gods so we could refuse these gifts?</p>	 <p><b>CHAOTIC EVIL</b></p> <p>All it takes is one bad day to reduce the sanest man alive to lunacy. That's how far the world is from where I am. Just one bad day.</p>

Oh. Not this kind of alignment.

Rust will generally align primitives to their sizes.

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.



```
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_WIDTHH
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_WIDTHH) {
        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))
    }
}
```

“Can you run faster just by trying harder?”



Performance improvements to date have used parallelism to improve throughput.

Decreasing latency is trickier—often requires domain-specific tweaks.

Today: one example of decreasing latency:  
Stream VByte.

Even Stream VByte uses parallelism:  
vector instructions.

But there are sequential improvements,  
e.g. Stream VByte takes care to be predictable for the branch predictor.

# Inverted Indexes (like it's CS 137 again!)

Abstractly: store a sequence of small integers.

Why Inverted indexes?

- allow fast lookups by term;
- support boolean queries combining terms.

# Dogs, cats, cows, goats. In ur documents.

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

Here's the index and the inverted index:

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

Inverted indexes contain many small integers.

Deltas typically small if doc ids are sorted.

VByte uses a variable number of bytes to store integers.

Why? Most integers are small,  
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.

Control bit, or high-order bit, is:

0 once done representing the int,  
1 if more bits remain.

Isn't dealing with variable-byte integers harder?

- Yup!

But perf improves:

- We are using fewer bits!

We fit more information into RAM and cache, and can get higher throughput.  
(think inlining)

Storing and reading 0s isn't good use of resources.

However, a naive algorithm to decode VByte gives branch mispredicts.

Stream VByte: a variant of VByte using SIMD.

Science is incremental.

Stream VByte builds on earlier work—  
masked VByte, VARINT-GB, VARINT-G8IU.

Innovation in Stream VByte:

*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

Per decode iteration:

- reads 1 byte from the control stream,  
and 16 bytes of data.

Lookup table on control stream byte: decide how many bytes it needs out of the 16 bytes it has read.

SIMD instructions:

- shuffle the bits each into their own integers.

Unlike VByte,

Stream VByte uses all 8 bits of data bytes as data.

Say 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 uses 9 bytes;

⇒ it advances the data pointer by 9.

The SIMD “shuffle” instruction puts decoded integers from data stream at known positions in the 128-bit SIMD register.

Pad the first 3-byte integer with 1 byte, then the next 1-byte integer with 3 bytes, etc.

# Stream VByte: Shuffling the Bits

Say the data input is:

0xf823 e127 2524 9748 1b.. ....

The 128-bit output is:

0x00f8 23e1/0000 0027/2524 9748/0000 001b

/s denote separation between outputs.

Shuffle mask is precomputed and read from an array.

The core of the implementation uses  
three SIMD instructions:

---

```
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++;
```

---



Stream VByte performs better than previous techniques on a realistic input.

Why?

- control bytes are sequential:  
CPU 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:  
takes 1 cycle;
- control-flow is regular  
(tight loop which retrieves/decodes control & data;  
no conditional jumps).