Lecture 17 — Mostly Data Parallelism

Patrick Lam
patrick.lam@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

November 23, 2020

Data and Task Parallelism

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?

This used to be easier...

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

What do we do now?

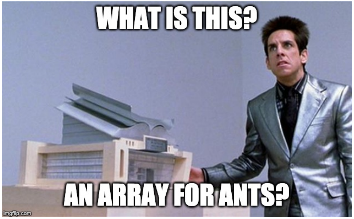
Instead of just +=1 we need to calculate the new number to add.

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.

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.

Bigger!

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.
- Intructions process multiple data items simultaneously. (Exact number is hardware-dependent).

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

One Command

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 all do the same thing!

SIMD Example

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.

SIMD Example—Assembly without SIMD

If we compile this without SIMD instructions on a 32-bit x86, (flags `-m32 -march=i386 -S`) we might get this:

```
loop:
  fldl  (%edx)
  faddl (%ecx)
  fstpl (%edx)
  addl  8, %edx
  addl  8, %ecx
  addl  1, %esi
  cmp   %eax, %esi
  jle   loop
```

Just loads, adds, writes and increments.

SIMD Example—Assembly with SIMD

Instead, compiling to SIMD instructions
(`-m32 -mfpmath=sse -march=prescott`) gives:

```
loop:
   movupd  (%edx),%xmm0
   movupd  (%ecx),%xmm1
   addpd   %xmm1,%xmm0
   movpd   %xmm0,(%edx)
   addl    16,%edx
   addl    16,%ecx
   addl    2,%esi
   cmp     %eax,%esi
   jle     loop
```

- Now processing two elements at a time on the same core.
- Also, no need for stack-based x87 code.

SIMD Overview

- Operations *packed*: operate on multiple data elements at the same time.
- On modern 64-bit CPUs, SSE has 16 128-bit registers.
- Very good if your data can be *vectorized* and performs math.
- Usual application: image/video processing.
- We'll see more SIMD as we get into GPU programming: GPUs excel at these types of applications.

"Can you run faster just by trying harder?"

Single-Thread Performance

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.

I have a cunning plan...

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

Inverting the Index

Here's the index and the inverted index:

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

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

Inverted indexes contain many small integers.

Deltas typically small if doc ids are sorted.

Storing inverted index lists: VByte

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 $2^{21} - 1$:
  $0xxxxxxx/1xxxxxxx/1xxxxxxx$;
- 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.

Control Stream

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

Decoding Stream VByte

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.

Stream VByte Example

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;
$\Rightarrow$ 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++;
```

But Does It Work?!

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).