Note 21: Flynn's Taxonomy, Data-Level Parallelism

Kevin Moy

Abstract

Performance is measured in different ways: throughput and latency. Fundamentally, this comes from the IO law of processor performance, where we predict and try to minimize the time taken to finish execution of some instruction or program. Today, we discuss how we increase performance through parallelism.

1 Introduction

Moore's Law predicts double the number of transistors every two years. The exponential curve is starting to flatten though, because of inherent power limitations for cooling processors- we certainly can't let the chip catch on fire with millions of processors.

Some tried to lower voltage to lower temperature; however, this made transistors leak too much current. So we can't make transistors faster because of current leakage, and thus we can't make performance better by waiting for clock speeds to increase.

How can we improve performance then? We must use **domain-specific hardware**. It will only do a few tasks, but do them extremely well. One example of DSH is the GPU: graphics and image processing. It can dedicate ALL its resources to purely graphics.

What if we want to improve general computing performance? We can simply use parallelism: creating a supercomputer by connecting many smaller ones in parallel, e.g. adding more cores to hardware. However, if we do this,

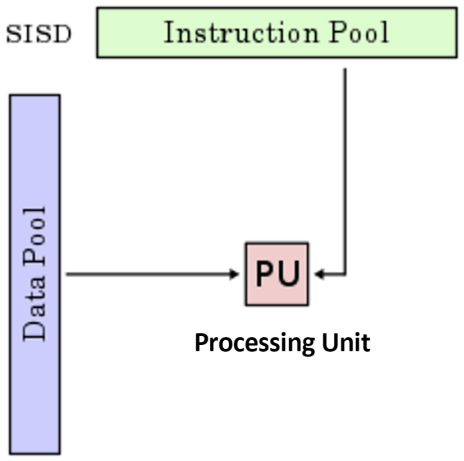
we require our software to account for parallel processing as well! This is quite challenging, and we'll cover this in today's note with SIMD- a parallel hardware ISA.

Now, multiple instructions can be executed at the same time. Let's get into it.

2 Flynn Taxonomy

When one result DOES NOT DEPEND on another- we can use parallelism and save so much time.

A modern example of parallelism in computing is the ability to have multiple instruction streams and data streams. For a conventional uniprocessor, we have one data stream from the **data pool**, and one instruction stream from the **instruction pool**:



No parallelism is exploited here. One instruction on one piece of data at a time. Familiar and easy to maintain/understand, but kind of boring and slow.

What if we could have multiple instructions though? In a multi-instruction/single data stream processor (MISD). The processor now performs a SERIES of computations on a single piece of data, according to the multiple instruction streams. No one really uses MISD anymore though.

What about multiple DATA streams (single instruction)? This is the SIMD we will spend this note covering. A single instruction can now be operated on multiple pieces of data at the same time. For example, we can add a 64-bit number *instantaneously* by sending 64 1-bit data streams to 64 ALUs, to calculate 64 sums, which are concanenated to form our result (in a single cycle).

There's also multiple data streams AND multiple instructions (MIMD). Each processing unit runs autonomously as its own computer. It can increase performance but starts to hit a questionable area for space. This has a lot of applications though, for warehouse-scale computers and multicore processors- more on this next note.

Below is a chart of all four classes of data-level parallelism. Notice that SISD corresponds to the single-cycle CPU we built in Logisim.

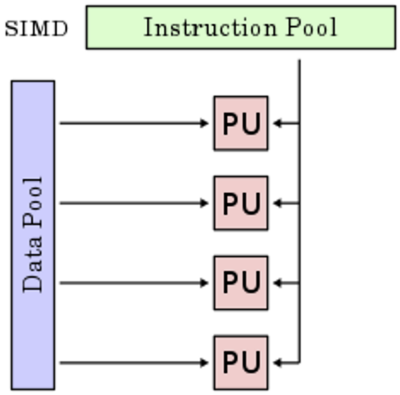
		Data Streams	
		Single	**Multiple**
Instruction Streams	**Single**	SISD: Single Stage Processor	SIMD: Vector Instructions
	Multiple	MISD: Nothing really here	MIMD: Multi-core Processors

We'll focus on SIMD this note- the vector instructions from Intel.

Sometimes, though, parallelism can fail. For example, if the goal is to get from point A to point B, doing 40 separate processes for this won't make it any faster than a single one. **We only want to use parallelism to increase performance or efficiency- otherwise sequential processing is simpler and better.**

3 SIMD- A Background

Let's take a look at the SIMD diagram:



For every element in our data pool, we process it with the same instruction on each data element. This is also known as **vector processing**. It really concerns calculating independent partial sums in parallel. Each of the operations within the vector are independent. Compilers CAN generate SIMD code, but they aren't always the best at it. We need to implement SIMD extensions ourselves. But at the same time, we only want a structure that allows us to only write assembly for single instructions- we can have parallelism implemented already. **Intel instrinsics** are instructions that look like C but are actually just directly converted by the compiler to assembly.

An example of an intrinsic is Intel's `_mm_add_epi32(__m128i a, __m128i b)`, which is an add instruction. `_mm_` means the function is a multimedia extension (all SIMD start with this). Next, `_add_` of course specifies the operation of the function. Finally, `epi32` means that the arguments are signed 32-bit **extended packed integers**. The arguments `a` and `b` have type `__m128i`- this represents a VECTOR that holds 128 bits (16 bytes) - space for 4 ints, 8 shorts, or 16 characters. Usually, these vectors will hold 4 ints. When `_mm_add_epi32` is called, 4 adds are computed on each pair of integers in parallel. The result will be saved in another 128-bit vector.

Go to Intel's intrinsics site to learn more about the different kind of intrinsics!

There are some more important data types:

- `__m256` is a 256-bit vector, holding 8 single-precision floats (each 32 bits).
- `__m256d` is a 256-bit vector, holding 4 DOUBLE-precision floats (each 64 bits).

- `__m256i` can hold bytes OR words but which depends on the function.

There are also some more important markings in instrinsic functions:

- `s` or `d` means single or double precision floats
- `innn` or `unnn` means a signed OR unsigned integer of size `nnn`, where `nnn` = 128, 64, 32, 16, 8.
- `ps` means packed single, `pd` means packed double, `sd` means scalar double (execution only on lowest value- rest are copied to result).
- `si256`: scalar 256-bit integer.

Now we move to important SSE intrinsics:

- `_mm_storeu_si128(__m128i *p, __m128i a)`: STORES 128-bit vector at pointer `p`.
- `_mm_loadu_si128(__m128i *p, __m128i a)`: LOADS 128 bits FROM pointer `p` INTO vector `a`.
- `_mm_set1_epi32(int i)`: SETS the four 32-bit ints in the vector to `i`.

4 A SIMD Example

Let's take a look at a function `add_no_SSE`, which takes two array pointers and a size. It iterates through each array, adds each pair of elements together, then assign the value back to the first array.

```
int add_no_SSE(int size, int *first_array, int *second_array) {
    for (int i = 0; i < size; ++i) {
        first_array[i] += second_array[i];
    }
}
```

This is fine- but now let's use Intel's intrinsics to implement data-level parallelism:

```
int add_SSE(int size, int *first_array, int *second_array) {
    for (int i=0; i + 4 <= size; i+=4) { // only works if (size % 4) == 0
```

```
    // load 128-bit chunks of each array
    __m128i first_values = _mm_loadu_si128((__m128i*) &first_array[i]);
    __m128i second_values = _mm_loadu_si128((__m128i*) &second_array[i]);
    // add each pair of 32-bit integers in the 128-bit chunks
    first_values = _mm_add_epi32(first_values, second_values);
    // store 128-bit chunk to first array
    _mm_storeu_si128((__m128i*) &first_array[i], first_values);
    }
}
```

Notice what changes. Notice with `_mm_loadu_si128` we load our arrays into a 128-bit vector (`first_values, second_values`) that will allow us to operate on 4 ints at a time. Specifically, the `_mm_add_epi32` intrinsic will treat the two argument vectors as packed with 4 32-bit integers (`epi32`) and add them. Notice that the number of iterations we have is actually divided by FOUR (`i+=4`), Notice our stopping condition will ALSO need to change.

We actually also need another for loop to handle what we call a **tail case**: leftover values to add that weren't included in our word-sized chunks. The start value in this case would be the *end* value of `i` in our outer for loop.

Note that we also can do parallelism with floats too. The only change is going from `epi32` to either `pd` (packed double) or `sd` (scalar double). Remember scalar double is a little weird in that only the LAST double will be considered in whatever operation is specified: the other ones are simply copied.

5 Loop Unrolling

For SIMD, the basic idea is we want adjacent values in memory to operate on in parallel. We usually specify this in loops:

```
for (i = 0; i < 1000; i++)
    x[i] = x[i] + s;
```

This loop will run 1000 iterations. However, we can **unroll** the loop and thus decrease the number of required iterations as well.

Let's see how looping in RISC-V works. Assume `s0` holds initial address of

array, `s1` holds scalar s, and `s2` holds address of end of array.

```
Loop:
    lw t0 0(s0)
    addu t0 t0 s1 #Add s to array elmt
    sw t0 0(s0) #store result
    addi s0 s0 4 #move to next elmt
    bne s0 s2 Loop #repeat Loop if not at end yet.
```

We can loop unroll and now load FOUR elements per iteration of loop:

```
Loop:
    lw t0 0(s0)
    add t0 t0 s1
    sw t0 0(s0)
    lw t1 4(s0)
    add t1 t1 s1
    sw t1 4(s0)
    lw t2 8(s0)
    add t2 t2 s1
    sw t2 8(s0)
    lw t3 12(s0)
    add t3 t3 s1
    sw t3 12(s0)
    addi s0 s0 16
    bne s0 s2 Loop
```

With loop unrolling, we decrease the overhead associated with loops because we only encounter a branch every FOUR data iterations. Again, the loop limit must be divisible by four for this to work- otherwise, we'd need to add a tail case. Finally, by using different registers `t0-t3`, we can reorder them to actually eliminate stalls! However, there is a huge disadvantage with code size.

Let's reorder our instructions to make it work with SIMD:

```
Loop:
```

```
    lw t0 0(s0)
    lw t1 4(s0)
    lw t2 8(s0)
    lw t3 12(s0) #Can replace with 1 wide SIMD load
    add t0 t0 s1
    add t1 t1 s1
    add t2 t2 s1
    add t3 t3 s1 #Can replace with 1 wide SIMD add.
    sw t0 0(s0)
    sw t1 4(s0)
    sw t2 8(s0)
    sw t3 12(s0) #Can replace with 1 wide SIMD store.
    addi s0 s0 16
    bne s0 s2 Loop
```

So for loop unrolling in C, we can unroll the given C loop

```
for (i = 0; i < 1000; i++)
    x[i] = x[i] + s;
```

to

```
for (i = 0; i < 1000; i+=4)
    x[i] = x[i] + s;
    x[i+1] = x[i+1] + s;
    x[i+2] = x[i+2] + s;
    x[i+3] = x[i+3] + s;
```

To generalize, for a loop with n iterations, we can perform a k-fold unrolling of the loop body by first running the loop $\lfloor n/k \rfloor$ times with k copies of the loop. Then for our tail case, run the loop with 1 copy of the body $n\%k$ times.

There are a lot of drawbacks to loop unrolling. First, it's super tedious. IRL our compiler would do this, but we should understand what it exactly is doing. Second, it increases static code size- important for accesses to instruction cache. We don't want k (the number of loop body copies) for each iteration to be *too* large.

Loop unrolling is an example of **code optimization**. It's harder to understand but has better performance.

For loop invariants, we want to compute those **outside** the loop. However, if it turns out we never want to compute these invariants ever, we should check if we NEED to calculate it at all before we do.

6 Summary

This note considered performance. Each year, we can stick more and more transistors onto a chip- and thus increase clock speed. However, there's eventually a power limit with cooling transistors. We could utilize parallelism and SIMD to exploit data-level parallelism. Loop unrolling pairs with SIMD to increase performance heavily.