CS61C Spring 2025

Precheck: Hardware Performance, DLP

Discussion 10

1 Pre-Check: T/F?

1.1 SIMD would work well for a task that has to add a constant value to every element in an array with 10000 elements.

True. SIMD is designed to exploit data-level parallelism by having a single instruction perform multiple of the same operation in parallel on several data elements at once. Since each addition operation is independent and the number of elements is very large, this is an ideal use case for SIMD.

1.2 SIMD architectures improve performance by decreasing instruction latencies.

False. SIMD improves performance by increasing throughput, since it allows us to execute multiple operations at the same time in parallel. It does not decrease the latency of each instruction.

1.3 SIMD is ideal for flow-control heavy tasks (i.e. tasks with many branches/if statements).

False. Data-level parallelism shines when we need to repeatedly perform the same operation on a large amount of data. Flow control statements disrupt the continuous flow of computation, which makes programs with them hard to take advantage of SIMD.

1.4 SIMD vector instructions invoke large "vector" registers available on compatible CPU architectures to perform one operation on multiple values at once.

True. For example, we can pack four 32-bit integers in a single 128-bit register and perform the same arithmetic operation on all four integers in one go, using an instruction such as __m128i __mm_add_epi32(__m128i a, __m128i b).

2 Measuring Performance

In order to measure the performance of a processor, we use the **Iron Law of Performance**:

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

The following terms are often used when discussing processor performance:

Latency: The amount of time it takes to execute one instruction.

$$\frac{\text{Time}}{\text{Instruction}}$$

Throughput: The number of instructions we can execute in a unit of time.

$$\frac{\#\text{Instructions}}{\text{Unit Time}}$$

3 Flynn's Taxonomy

We can classify hardware architectures using a system called **Flynn's Taxonomy**.

Flynn's Taxonomy divides architectures into four categories:

- **SISD** (**Single Instruction, Single Data**): A single instruction stream operates on a single data stream (ex: RISC-V Datapath)
- **SIMD** (**Single Instruction, Multiple Data**): A single instruction stream operates on multiple data streams. (ex. Intel SIMD instruction extensions)
- **MISD** (**Multiple Instruction, Single Data**): Multiple instruction streams operate on a single data stream. Rarely used in practice, not covered in 61C.
- **MIMD** (**Multiple Instruction, Multiple Data**): Multiple autonomous processors simultaneously executing different instructions on different data. (ex. Multicore)

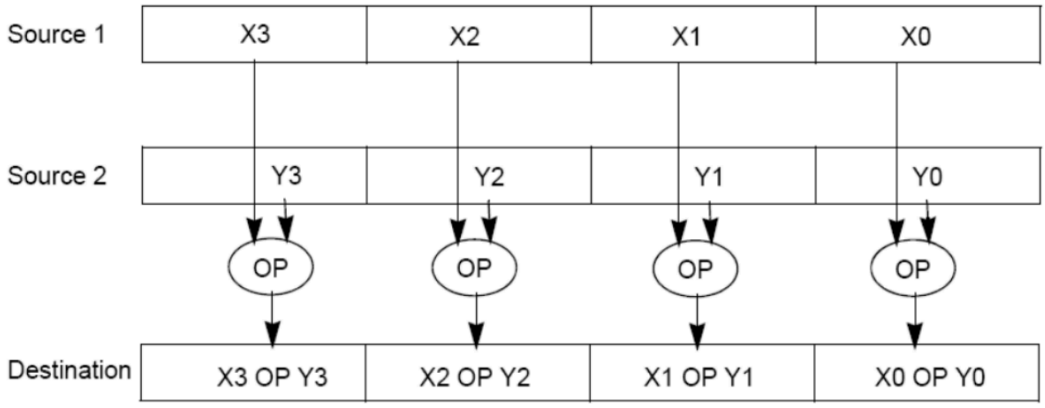
  In this class, we will focus mostly on SISD & SIMD.

4 Data-Level Parallelism

SIMD architectures improve performance by utilizing a form of parallelism called **data-level parallelism**.

The key idea behind data-level parallelism is vectorized calculations. In vectorized calculations, an operation is applied to multiple elements (which are part of a single vector) at the same time.

Vector registers on SIMD architectures are large enough to hold multiple values, and when a vector instruction is executed, the same operation is performed on each value in that vector. An example of this is shown below.



Some machines with x86 architectures can use Intel Intrinsics (Intel proprietary technology) which allow us to use these wider "vector" registers to harness the power of DLP in C code.

Below is a small selection of the available Intel intrinsic instructions. All of them perform operations using 128-bit registers. The type __m128i is used when these registers hold 4 ints, 8 shorts or 16 chars; __m128d is used for 2 double precision floats, and __m128 is used for 4 single precision floats. Where you see "epiXX", epi stands for **e**xtended **p**acked **i**nteger, and XX is the number of bits in the integer. "epi32" for example indicates that we are treating the 128-bit register as a pack of 4 32-bit integers.

Function	**Description**
`__m128i`	Datatype for a 128-bit vector.
`__m128i _mm_set1_epi32(int i)`	Creates a vector with four signed 32-bit integers where every element is equal to `i`.
`__m128i _mm_loadu_si128(__m128i *p)`	Load 4 consecutive integers at memory address `p` into a 128-bit vector.
`void _mm_storeu_si128(__m128i *p, __m128i a)`	Stores vector `a` into memory address `p`
`__m128i _mm_add_epi32(__m128i a, __m128i b)`	Returns a vector = $(a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3)$
`__m128i _mm_mullo_epi32(__m128i a, __m128i b)`	Returns a vector = $(a_0 \times b_0, a_1 \times b_1, a_2 \times b_2, a_3 \times b_3)$.

Function	Description
`__m128i _mm_and_si128(__m128i a, __m128i b)`	Perform a bitwise AND of 128 bits in a and b, and return the result.
`__m128i _mm_cmpeq_epi32(__m128i a, __m128i b)`	The ith element of the return vector will be set to 0xFFFFFFFF if the ith elements of a and b are equal, otherwise it'll be set to 0.

Here is an example of a function that adds 1 to each element in an array:

```
void add_one_naive(int32_t *a, size_t len) {
  for (int i = 0; i < len; i += 1) {
    a[i] = a[i] + 1;
  }
}
```

Here's the same function rewritten using SIMD vector instructions:

```
void add_one_simd(int32_t *a, size_t len) {
  __m128i vector; //declare a 128-bit SIMD register

  // declare a SIMD register with four 1s
  __m128i vector_ones = _mm_set1_epi32(1);

  for (int i = 0; i < len / 4 * 4; i += 4) {

    // load memory segment (4 ints) into vector. Note that
    // the memory address must be typecast as __m128i * (vector pointer)
    vector = _mm_loadu_si128((__m128i *)(a+i));

    // compute vectorized addition
    vector = _mm_add_epi32(vector, vector_ones);

    // store data in the vector back to memory
    _mm_storeu_si128((__m128i *)(a + i), vector)
  }

  // Handle Tail Case (if len isn't divisible by 4)
  for (int i = len / 4 * 4; i < len; i += 4) { //
    a[i] = a[i] + 1;
  }
}
```

Notice how the vectorized function operates in multiples of `4` and goes until the loop condition of `len / 4 * 4` (hint: what does this evaluate to in C?). Because we can only operate in units of our 4 integers because of our 128-bit vector length, we have to include a tail case for when our input array is not a multiple of 4.

5 Amdahl's Law

Amdahl's Law can be used to measure the maximum speedup that can be obtained through parallelization:

$$\text{Speedup} = \frac{1}{\left(1 - \text{frac}_{\text{optimized}}\right) + \frac{\text{frac}_{\text{optimized}}}{\text{factor}_{\text{improvement}}}}$$

For example, by using parallelism to increase the performance of 25% of a program by a factor of 4:

$$\begin{aligned}
\text{Speedup} &= \frac{1}{(1 - 0.75) + \frac{0.25}{4}} \\
&= \frac{1}{0.25 + 0.0625} \\
&= \frac{1}{0.3125} \\
&= 3.2
\end{aligned}$$

...meaning we get an overall performance boost of 3.2x by introducing parallelism to our program!