CS61C Fall 2026

Precheck: DLP, TLP, and Concurrency

Discussion 11

1 Pre-Check: Data-Level Parallelism (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)

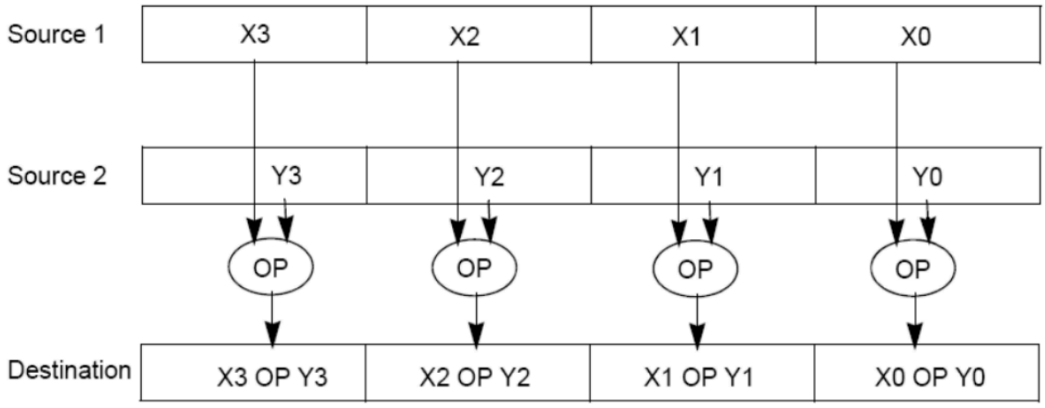
  In this class, we will focus mostly on SISD & SIMD.

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 `int`s, 8 `short`s or 16 `char`s; `__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 vector 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 = $(a_0 b_0, a_1 b_1, a_2 b_2, a_3 b_3)$.

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:

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

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

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

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

6 Pre-Check: Thread-Level Parallelism (T/F?)

6.1 Multithreaded code will always outperform single-threaded code.

False. Not all code lends itself to parallelization. Code with many dependencies may require long critical sections which may be slower than single-threaded code. Additionally, spawning new threads takes time that will slow down the program.

6.2 Race conditions occur when multiple threads attempt to modify the same resource at the same time.

True. Race conditions can have unintended side-effects when writing parallel code. An example is given below:

```
void race_condition(int n, int *a) {
  int result = 0;
  #pragma omp parallel for
  for (int i = 0; i < n; i += 1) {
    result += a[i];
  }
}
```

Accessing `a[i]`, adding `result + a[i]`, and storing back to `result` are separate operations during which threads may operate concurrently. These can be solved with critical sections!

6.3 Every thread has its own set of registers, program counter, heap and global variables

False. Each thread has its own set of registers and program counter, but the heap and global variables are shared amongst all threads.

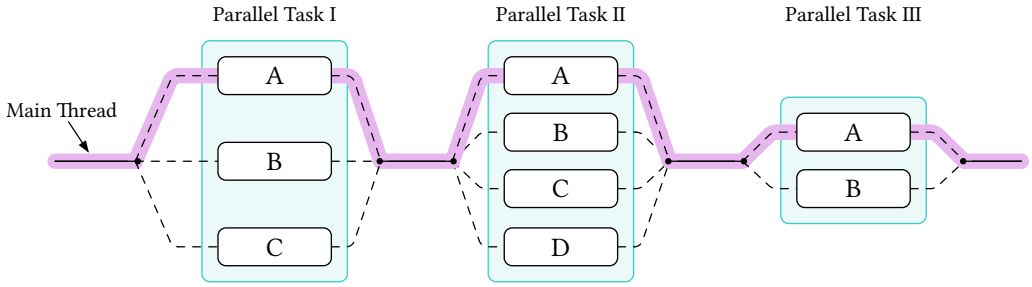
6.4 Multithreaded programs can still execute on single-core processors

True. Single-core processors will interleave its execution between different threads.

7 Thread-Level Parallelism

Multithreading improves performance by utilizing a form of parallelism called **thread-level parallelism**. The key idea behind thread-level parallelism is splitting from a single line of execution to multiple lines executing concurrently.

Multithreaded programs will start with a main thread that will spawn multiple threads when parallelism is required (See Lecture 31).



In this class, we use the OpenMP library to create and manage threads. Consider the sample hello world program, which prints "hello world from thread #" from each thread:

```
int main() {
    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num();
        printf("hello world from thread %d\n", thread_id);
    }
}
```

This program will create a team of parallel threads. Each thread prints out a hello world message, along with its own thread number.

Let's break down the `pragma omp parallel` line:

- `pragma` tells the compiler that the rest of the line is a directive.
- `omp` declares that the directive is for OpenMP.
- `parallel` says that the following block statement – the part inside the curly braces ({/}) – should be executed in parallel by different threads.

Note that each thread has its own registers (including stack pointer) and program counter (PC). However, memory in the heap or global variables are shared amongst all threads.

8 OpenMP

OpenMP provides an easy interface for using multithreading within C programs. Some examples of OpenMP directives:

- The `parallel` directive indicates that each thread should run a copy of the code within the block. If a for loop is put within the block, **every** thread will run every iteration of the for loop.

  ```
  #pragma omp parallel
  {
      ...
  }
  ```

  NOTE: The opening curly brace needs to be on a newline or else there will be a compile-time error!

- The `parallel for` directive will split up iterations of a for loop over various threads. Every thread will run **different** iterations of the for loop. The exact order of execution across all threads, as well as the number of iterations each thread performs, are both non-deterministic, as the OpenMP library load balances threads for performance. The following two code snippets are equivalent.

  ```
  #pragma omp parallel for
  for (int i = 0; i < n; i++) {
      ...
  }
  ```

```
#pragma omp parallel
{
    #pragma omp for
    for (int i =0; i < n; i++) { ... }
}
```

- The `critical` directive only allows a single thread to access the following line / block of code at once. It is useful to prevent race conditions when modifying resources shared between threads.

```
// Assume arr has length n
int fast_sum(int *arr, int n) {
    int result = 0;
    int result2 = 0;
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        #pragma omp critical
        result += arr[i];  // limited to one thread access at once
    }
    return result;
}
```

- The `parallel for reduction(operation : var)` directive creates and optimizes the critical section for a loop, given a variable that should be in the critical section and the operation being performed on that variable. An example is given below.

```
// Assume arr has length n
int fast_sum(int *arr, int n) {
    int result = 0;
    #pragma omp parallel for reduction(+: result)
    for (int i = 0; i < n; i++) {
        result += arr[i];
    }
    return result;
}
```

Additionally, there are two OpenMP functions that may be useful to you:

- `int omp_get_thread_num()` will return the number of the thread executing the code
- `int omp_get_num_threads()` will return the number of total hardware threads executing the code

9 Race Conditions

Two threads attempting to access the same memory can result in a race condition. Consider the following:

```
void example(int n, int *a) {
  int result = 0;
  #pragma omp parallel for
  for (int i = 0; i < n; i += 1) {
    result += a[i];
  }
}
```

For a single thread to compute the line `result += a[i]`:
(a) Read the value `a[i]`
(b) Compute the value `result + a[i]`
(c) Store the new value back to `result`

Different threads often have their instructions interleaved. Without defining a critical section, these threads may be executing steps (a), (b), and (c) at any time and the value of `result` may be indeterminate.

To fix this, we can introduce a critical section to limit one thread to executing `result += a[i]` at any given time:

```
void example(int n, int *a) {
  int result = 0;
  #pragma omp parallel for
  for (int i = 0; i < n; i += 1) {
    #pragma omp critical
    result += a[i];
  }
}
```

It can also be fixed with the `reduction` directive (see above)!