# Flynn’s Taxonomy

1.1 Explain SISD and give an example if available.

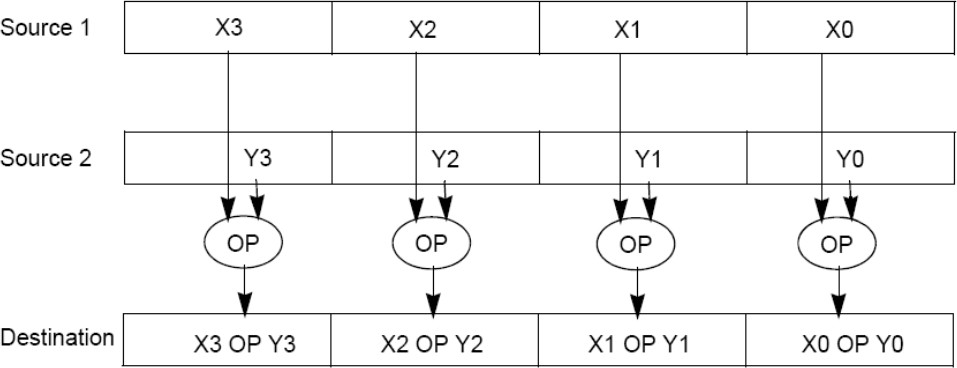
1.2 Explain SIMD and give an example if available.

1.3 Explain MISD and give an example if available.

1.4 Explain MIMD and give an example if available.

# Data-Level Parallelism

The idea central to data level parallelism is vectorized calculation: applying operations to multiple items (which are part of a single vector) at the same time.



Some machines with x86 architectures have special, wider registers, that can hold 128, 256, or even 512 bits. Intel intrinsics (Intel proprietary technology) allow us to use these wider 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.

* m128i \_mm\_set1\_epi32(**int** i):

Set the four signed 32-bit integers within the vector to i.

* m128i \_mm\_loadu\_si128( m128i \*p):

Load the 4 successive ints pointed to by p into a 128-bit vector.

* m128i \_mm\_mullo\_epi32( m128i a, m128i b): Return vector (*a*0 *· b*0*, a*1 *· b*1*, a*2 *· b*2*, a*3 *· b*3).
* m128i \_mm\_add\_epi32( m128i a, m128i b): Return vector (*a*0 + *b*0*, a*1 + *b*1*, a*2 + *b*2*, a*3 + *b*3)
* **void** \_mm\_storeu\_si128( m128i \*p, m128i a): Store 128-bit vector a at pointer p.
* 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.

**Notice:** On this worksheet, we are using the *unaligned* versions of the commands that interface with memory (i.e. storeu/loadu vs. store/load). This is because the store/load commands require that the address we are loading at is aligned at some byte boundary (and not necessarily just word-aligned), whereas the unaligned versions have no such requirements. For instance, \_mm\_store\_si128 needs the address to be aligned on a 16-byte boundary (i.e. is a multiple of 16). There is extra work that needs to be done to achieve these alignment requirements, so for this class, we just use the unaligned variants.

2.1 You have an array of 32-bit integers and a 128-bit vector as follows:

1 **int** arr[8] = {1, 2, 3, 4, 5, 6, 7, 8};

2 m128i vector = \_mm\_loadu\_si128(( m128i \*) arr);

For each of the following tasks, fill in the correct arguments for each SIMD instruction, and where necessary, fill in the appropriate SIMD function. Assume they happen independently, i.e. the results of Part (a) do not at all affect Part (b).

1. Multiply vector by itself, and set vector to the result.

1 vector = ( , );

1. Add 1 to each of the first 4 elements of the arr, resulting in arr = {2, 3, 4, 5, 5, 6, 7, 8}

1 m128i vector\_ones = \_mm\_set1\_epi32( );

2 m128i result = \_mm\_add\_epi32( , );

3 \_mm\_storeu\_si128( , );

1. Add the second half of the array to the first half of the array, resulting in arr = {1 + 5, 2 + 6, 3 + 7, 4 + 8, 5, 6, 7, 8} = {6, 8, 10, 12, 5, 6, 7, 8}

1 m128i result = \_mm\_add\_epi32(\_mm\_loadu\_si128( ), );

2 \_mm\_storeu\_si128( , );

1. Set every element of the array that is not equal to 5 to 0, resulting in arr

= {0, 0, 0, 0, 5, 0, 0, 0}. Remember that the first half of the array has already been loaded into vector.

1 m128i fives = ( );

2 m128i mask = ( , );

3 m128i result = ( , );

4 \_mm\_storeu\_si128( , );

5 vector = \_mm\_loadu\_si128( );

6 mask = ( , );

7 result = ( , );

8 \_mm\_storeu\_si128( , );

2.2 SIMD-ize the following function, which returns the product of all of the elements in an array. Things to think about: When iterating through a loop and grabbing elements 4 at a time, how should we update our index for the next iteration? What if our array has a length that isn’t a multiple of 4? Can we always SIMD-ize an entire array? What can we do to handle this tail case?

**static int** product\_naive(**int** n, **int** \*a) {

**int** product = 1;

**for** (**int** i = 0; i < n; i++) { product \*= a[i];

}

**return** product;

}

**static int** product\_vectorized(**int** n, **int** \*a) {

**int** result[4];

m128i prod\_v = ;

**for** (**int** i = 0; i < ; i += ) { // Vectorized loop

prod\_v = ;

}

mm\_storeu\_si128( , );

**for** (**int** i = ; i < ; i++) { // Handle tail case result[0] \*= ;

}

**return** ;

}

# Thread-Level Parallelism

As powerful as data level parallelization is, it can be quite inflexible, as not all applications have data that can be vectorized. Multithreading, or running a single piece of software on multiple hardware threads, is much more powerful and versatile.

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

* 1. 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!

* 1. 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 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++) { ... }

}

There are two functions you can call that may be useful to you:

* 1. **int** omp\_get\_thread\_num() will return the number of the thread executing the code
  2. **int** omp\_get\_num\_threads() will return the number of total hardware threads executing the code

3.1 For each question below, state and justify whether the program is **sometimes incorrect**, **always incorrect**, **slower than serial**, **faster than serial**, or **none of the above**. Assume the default number of threads is greater than 1. Assume no thread will complete before another thread starts executing. Assume arr is an **int**[] of length n.

1. // Set element i of arr to i

**#pragma** omp parallel

{

**for** (**int** i = 0; i < n; i++) arr[i] = i;

}

1. // Set arr to be an array of Fibonacci numbers. arr[0] = 0;

arr[1] = 1;

**#pragma** omp parallel **for for** (**int** i = 2; i < n; i++)

arr[i] = arr[i-1] + arr[i - 2];

1. // Set all elements in arr to 0;

**int** i;

**#pragma** omp parallel **for for** (i = 0; i < n; i++)

arr[i] = 0;

3.2 What potential issue can arise from this code?

1 // Decrements element i of arr. n is a multiple of omp\_get\_num\_threads()

2 **#pragma** omp parallel

3 {

4 **int** threadCount = omp\_get\_num\_threads();

5 **int** myThread = omp\_get\_thread\_num();

6 **for** (**int** i = 0; i < n; i++) {

7 **if** (i % threadCount == myThread) arr[i] -= 1;

8 }

9 }

3.3 // Assume n holds the length of arr

2 **double** fast\_product(**double** \*arr, **int** n) {

3 **double** product = 1;

4 #pragma omp parallel **for**

5 **for** (**int** i = 0; i < n; i++) {

6 product \*= arr[i];

7 }

8 **return** product;

9 }

1. What is wrong with this code?
2. Fix the code using #pragma omp critical
3. Fix the code using #pragma omp reduction(operation: var).

# Instruction Level Parallelism - Trace Scheduling

Trace scheduling is a compiler technique that increases ILP by removing control dependencies, allowing operations following branches to be moved up and speculatively executed in parallel with operations before the branch. It was originally developed for statically scheduled VLIW machines, but it is a general technique that can be used in different types of machines and in this question, we apply it to a single-issue RISC-V processor.

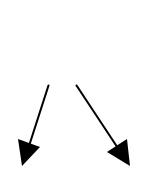
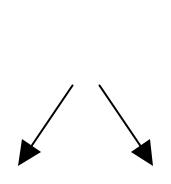
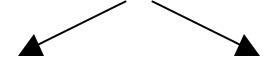
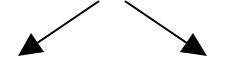
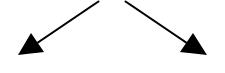
Consider the following piece of C code (% is modulus) with basic blocks labeled:

1. if (data % 8 == 0)
2. X = V0 / V1; else
3. X = V2 / V3;
4. if (data % 4 == 0)
5. Y = V0 \* V1; else
6. Y = V2 \* V3;

G

Assume that **data** is a uniformly distributed integer random variable that is set sometime before executing this code.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| The program’s control flow graph is | | | | | | |  | The decision tree is | | | | | | |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  | A |  |  |  |  |  |  |  | A |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  | |  |  |  |  |  |  |  |  |  |  |  |  |
|  | B |  | |  | C |  |  | B | |  |  |  | C | |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  | D | |  |  |  | D | |  |
|  |  |  | D |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  | |  |  | |  |  |
|  |  |  |  |  |  |  | E |  | F | |  | E | |  | F |
|  | E |  |  |  | F |  |
|  |  | |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  | Path | G |  | G | |  | G | |  | G |
|  |  |  |  |  |  |
|  |  |  |  |  |  | probabilities |  |  |  |  |  |  |  |  |  |
|  |  |  | G |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  | for 5.A: | 1/8 |  |  | 0 |  | 1/8 | |  | 6/8 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |



A control flow graph and the decision tree both show the possible flow of execution through basic blocks. However, the control flow graph captures the static structure of the program, while the decision tree captures the dynamic execution (history) of the program.

4.1 On the decision tree, label each path with the probability of traversing that path. For example, the leftmost block will be labeled with the total probability of executing the path ABDEG. (Hint: you might want to write out the cases). Circle the path that is most likely to be executed.

Circle: ACDFG

4.2 This is the RISC-V code:

|  |  |  |
| --- | --- | --- |
| A: | lw x1, data |  |
|  | andi x2, x1, 7 | # x2 <- x1 % 8 |
|  | beqz x2, C |  |
| B: | div x3, x4, x5 | # X <- V0 / V1 |
|  | j D |  |
| C: | div x3, x6, x7 | # X <- V2 / V3 |
| D: | andi x2, x1, 3 | # x2 <- x1 % 4 |
|  | bnez x2, F |  |
| E: | mul x8, x4, x5 | # Y <- V0 \* V1 |
|  | j G |  |
| F: | mul x8, x6, x7 | # Y <- V2 \* V3 |
| G: |  |  |

This code is to be executed on a single-issue processor **with perfect branch prediction**. Assume that the memory, divider, and the multiplier are all separate, long latency, **unpipelined** units that can be run in parallel.

Assume that the load takes x cycles, the divider takes y cycles, and the multiplier takes z cycles. Approximately how many cycles does this code take ***in the best case, in the worst case***, and ***on average***? (ignore the latency of ALU)

Best, worst, average: x + max(y,z) cycles

4.3 With trace scheduling, we can obtain the following code:

|  |  |  |
| --- | --- | --- |
| ACF: | ld x1, data |  |
|  | div x3, x6, x7 | # X <- V2 / V3 |
|  | mul x8, x6, x7 | # Y <- V2 \* V3 |
| D: | andi x2, x1, 3 | # x2 <- x1 % 4 |
|  | bnez x2, G |  |
| A: | andi x2, x1, 7 | # x2 <- x1 % 8 |
|  | bnez x2, E |  |
| B: | div x3, x4, x5 | # X <- V0 / V1 |
| E: | mul x8, x4, x5 | # Y <- V0 \* V1 |
| G: |  |  |
|  |  |  |

We optimize only for the most common path, but the other paths are still correct. Approximately how many cycles does the new code take ***in the best case, in the worst case*** and ***on average***? Is it faster ***in the best case, in the worst case*** and ***on average*** than the code in Problem 4.2?

Best case: max(x, y, z)

Worst case: max(x, y, z) + max(y, z)

Average:

6/8 \* max(x,y,z) + 1/8 \* (max(x, y, z) + z) + 1/8 \* (max(x, y, z) + max(y, z))

= max(x, y, z) + 1/8 \* max(y, z) + 1/8 \* z

<= max(x, y, z) + 1/4 \* max(y, z)

If max(x,y,z) = x, this code is faster in the best case and on average, and not slower in

the worst case.

If max(x,y,z) = max(y,z), this code is still faster in the best case, but slower in the

worst case(x + max(y,z) < 2\*max(y, z)). If max(y, z) <= 4 \* x, this code is faster on

average but slower otherwise.

# Amdahl’s Law

In the programs we write, there are sections of code that are naturally able to be sped up. However, there are likely sections that just can’t be optimized any further to maintain correctness. In the end, the overall program speedup is the number that matters, and we can determine this using Amdahl’s Law:

True Speedup = 1/(S+(1-S)/F)

where *S* is the non-sped-up part and *P* is the speedup factor (determined by the number of cores, threads, etc.).

5.1 You are going to run a convolutional network to classify a set of 100,000 images using a computer with 32 threads. You notice that 99% of the execution of your project code can be parallelized on these threads. What is the speedup?

5.2 You run a profiling program on a different program to find out what percent of this program each function takes. You get the following results:

|  |  |
| --- | --- |
| Function | % Time |
| f  g  h | 30%  10%  60% |

1. We don’t know if these functions can actually be parallelized. However, assuming all of them can be, which one would benefit the most from parallelism?
2. Let’s assume that we verified that your chosen function can actually be parallelized. What speedup would you get if you parallelized just this function with 8 threads?

# Multithreading

This problem evaluates the effectiveness of multithreading using a simple database benchmark. The benchmark searches for an entry in a linked list built from the following structure, which contains a key, a pointer to the next node in the linked list, and a pointer to the data entry.

struct node { int key;

struct node \*next; struct data \*ptr;

}

The following RISC-V code shows the core of the benchmark, which traverses the linked list and finds an entry with a particular key.

|  |  |  |
| --- | --- | --- |
| loop: | **LW x3, 0(x1)** | # load a key |
|  | **LW x4, 4(x1)** | # load the next pointer |
|  | **SEQ x3, x3, x2** | # set x3 if x3 == x2 |
|  | **BNEZ x3, end** | # found the entry |
|  | **ADD x1, x0, x4** |  |
|  | **BNEZ x4, loop** | # check the next node |
| end: |  |  |

We run this benchmark on a single-issue in-order processor. The processor can fetch and issue one instruction per cycle. If an instruction cannot be issued due to a data dependency, the processor stalls. Integer instructions take one cycle to execute and the result can be used in the next cycle. For example, if SEQ is executed in cycle 1, BNEZ can be executed in cycle 2. We also assume that the processor has a perfect branch predictor with no penalty for both taken and not-taken branches.

6.1 Assume that our system does not have a cache. Each memory operation directly accesses main memory and takes 100 CPU cycles. The load/store unit is fully pipelined, and non-blocking. After the processor issues a memory operation, it can continue executing instructions until it reaches an instruction that is dependent on an outstanding memory operation. How many cycles does it take to execute one iteration of the loop in steady state?

|  |  |  |
| --- | --- | --- |
| Instruction | Start Cycle | End Cycle |
| LW x3, 0(x1) | 0 | 100 |
| LW x4, 4(x1) | 1 | 101 |
| SEQ x3, x3, x2 | 101 | 101 |
| BNEZ x3, End | 102 | 102 |
| ADD x1, x0, x4 | 103 | 103 |
| BNEZ x1, Loop | 104 | 104 |

6.2 Now we add zero-overhead multithreading to our pipeline. A processor executes multiple threads, each of which performs an independent search. Hardware mechanisms schedule a thread to execute each cycle.

In our first implementation, the processor switches to a different thread every cycle using fixed round robin scheduling (similar to CDC 6600 PPUs). Each of the N threads executes one instruction every N cycles. What is the **minimum** number of threads that we need to fully utilize the processor, i.e., execute one instruction per cycle?

If we have N threads and the first load executes at cycle 1, SEQ , which depends on the load,

executes at cycle 2N + 1. To fully utilize the processor, we need to hide 100-cycle memory

latency, 2N + 1  101. The minimum number of thread needed is 50.

6.3 How does multithreading affect throughput (number of keys the processor can find within a given time) and latency (time processor takes to find an entry with a specific key)? Assume the processor switches to a different thread every cycle and is fully utilized. Check the correct boxes.

|  |  |  |
| --- | --- | --- |
|  | Throughput | Latency |
| Better | √ |  |
| Same |  |  |
| Worse |  | √ |

6.4 We change the processor to only switch to a different thread when an instruction cannot execute due to data dependency. What is the minimum number of threads to fully utilize the processor now? Note that the processor issues instructions in-order in each thread.

In steady state, each thread can execute 6 instructions (SEQ, BNEZ, ADD, BNEZ, LW,

LW). Therefore, to hide 98 (100-2(# of instructions between LW and SEQ)) cycles

between the second LW and SEQ, a processor needs 98/6+1=18 threads.