CS 61C Fall 2019

Flynn's Taxonomy, DLP

Discussion 11: November 11, 2019

1 Flynn's Taxonomy

1.1 Explain SISD and give an example if available.

Single Instruction Single Data; each instruction is executed in order, acting on a single stream of data. For example, traditional computer programs.

1.2 Explain SIMD and give an example if available.

Single Instruction Multiple Data; each instruction is executed in order, acting on multiple streams of data. For example, the SSE Intrinsics.

1.3 Explain MISD and give an example if available.

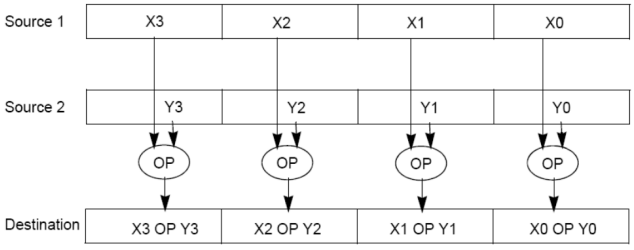
Multiple Instruction Single Data; multiple instructions are executed simultaneously, acting on a single stream of data. There are no good modern examples. can help with redundancy

1.4 Explain MIMD and give an example if available.

Multiple Instruction Multiple Data; multiple instructions are executed simultaneously, acting on multiple streams of data. For example, map reduce or multithreaded programs.

2 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 $(a_0 \cdot b_0, a_1 \cdot b_1, a_2 \cdot b_2, a_3 \cdot b_3)$.
- `__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).

(a) Multiply `vector` by itself, and set `vector` to the result.

look at fdble to find match

```
1 vector = _mm_mullo_epi32(vector, vector);
```

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

make vector of 1's.

add our victor + the 1's vector

store it back to where we got it.

```
1 __m128i vector_ones = _mm_set1_epi32(1);
2 __m128i result = _mm_add_epi32(vector, vector_ones);
3 _mm_storeu_si128((__m128i *) arr, result);
```

(c) 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}`

add the two vectors together

load in the second half of the array

```
1  __m128i result = _mm_add_epi32(_mm_loadu_si128((__m128i *) (arr + 4)), vector);
2  _mm_storeu_si128((m128i*) arr, result);
```

override the original front part of the array.

(d) 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.

first half

second half

```
1  __m128i fives = _mm_set1_epi32(5);
2  __m128i mask = _mm_cmpeq_epi32(vector, fives);
3  __m128i result = _mm_and_si128(mask, vector);
4  __mm_storeu_si128((__m128i *) arr, result);
5  vector = _mm_loadu_si128((__m128i *) (arr + 4));
6  mask = _mm_cmpeq_epi32(vector, fives);
7  result = _mm_and_si128(mask, vector);
8  _mm_storeu_si128((__m128i *) (arr + 4), result);
```

make vector full of 5's.

compute the 5's vector to the input vector. Remember, this will then output all 1's or all 0's i.e. 0xFFFFFFFF or 0.

Because of this, we just re-apply the mask to get back just 5's or zeros.

Finally we store our result back to the array

This is us Replicating what we did for the first parts.

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) {
1    int product = 1;
2    for (int i = 0; i < n; i++) {
3        product *= a[i];
     }
4    return product;
}
```

```
static int product_vectorized(int n, int *a) {
     int result[4];
1    __m128i prod_v = __mm_set1_epi32(1);
2    for (int i = 0; i < n/4 * 4; i += 4) { // Vectorized loop
3        prod_v = __mm_mullo_epi32(prod_v, __mm_loadu_si128((__m128i *) (a + i)));
     }
23   _mm_storeu_si128((__m128i *) result, prod_v);
2    for (int i = n/4 * 4; i < n; i++) { // Handle tail case
3        result[0] *= a[i];
     }
3+4  return result[0] * result[1] * result[2] * result[3];
}
```

Initializer stays the same.

we prep our result

init product to 1.

work on data sets of multiple 4.

do our products together

load in array.

store our product vector to an array

tail case for n % 4 ≠ 0

multiply them back together.

Remember n/4 is floor division so n/4 * 4 will chop off the remainder bits (i.e. 9 → 8)