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.

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 extended packed integer, 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.

- \texttt{__m128i \_mm\_setl\_epi32(int i)}:
  Set the four signed 32-bit integers within the vector to i.
- \texttt{__m128i \_mm\_loadu\_si128( \_m128i \_p)}:
  Return the 128-bit vector stored at pointer \( p \).
- \texttt{__m128i \_mm\_mullo\_epi32(\_m128i a, \_m128i b)}:
  Return vector \((a_0 \cdot b_0, a_1 \cdot b_1, a_2 \cdot b_2, a_3 \cdot b_3)\).
- \texttt{__m128i \_mm\_add\_epi32(\_m128i a, \_m128i b)}:
  Return vector \((a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3)\).
- \texttt{\_mm\_storeu\_si128( \_m128i \_p, \_m128i a)}:
  Store 128-bit vector \( a \) at pointer \( p \).
- \texttt{\_mm\_and\_si128(\_m128i a, \_m128i b)}:
  Perform a bitwise AND of 128 bits in \( a \) and \( b \), and return the result.
- \texttt{\_mm\_cmpeq\_epi32(\_m128i a, \_m128i b)}:
  Compare packed 32-bit integers in \( a \) and \( b \) for equality, set return vector to \( 0xFFFFFFFF \) if equal and 0 if not.

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

```c
int arr[8] = {1, 2, 3, 4, 5, 6, 7, 8};
__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 \( \text{vector} \) by itself, and set \( \text{vector} \) to the result.

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

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

```c
__m128i vector\_ones = _mm_setl\_epi32(1);
__m128i result = _mm_add\_epi32(vector, vector\_ones);
_mm\_storeu\_si128(\_m128i \_p, result);
```

\textbf{Notice:} In this and the following solutions, we are using the \textit{unaligned} versions of the commands that interface with memory (i.e. \textit{storeu/loadu} vs. \textit{store/load}). This is because the \textit{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, \texttt{\_mm\_storeu\_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.

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

1. \( \_m128i \ \text{result} = \_mm\_add\_epi32(\_mm\_loadu\_si128((\_m128i *) (\text{arr} + 4)), \text{vector}); \)
2. \( \_mm\_storeu\_si128((\_m128i*) \text{arr}, \text{result}); \)

(d) Set every element of the array that is not equal to 5 to 0, resulting in \( \text{arr} = \{0, 0, 0, 0, 5, 0, 0, 0\} \). Remember that the first half of the array has already been loaded into vector.

1. \( \_m128i \ \text{fives} = \_mm\_set1\_epi32(5); \)
2. \( \_m128i \ \text{mask} = \_mm\_cmpeq\_epi32(\text{vector}, \text{fives}); \)
3. \( \_m128i \ \text{result} = \_mm\_and\_si128(\text{mask}, \text{vector}); \)
4. \( \_mm\_storeu\_si128((\_m128i*) \text{arr}, \text{result}); \)
5. \( \text{vector} = \_mm\_loadu\_si128((\_m128i*) (\text{arr} + 4)); \)
6. \( \text{mask} = \_mm\_cmpeq\_epi32(\text{vector}, \text{fives}); \)
7. \( \text{result} = \_mm\_and\_si128(\text{mask}, \text{vector}); \)
8. \( \_mm\_storeu\_si128((\_m128i*) (\text{arr} + 4), \text{result}); \)

Implement the following function, which returns the product of all of the elements in an array.

```c
static int product_naive(int n, int *a) {
    int product = 1;
    for (int i = 0; i < n; i++) {
        product *= a[i];
    }
    return product;
}
```

```c
static int product_vectorized(int n, int *a) {
    int result[4];
    _m128i prod_v = _mm_set1_epi32(1);
    for (int i = 0; i < n/4 * 4; i += 4) { // Vectorized loop
        prod_v = _mm_mullo_epi32(prod_v, _mm_loadu_si128((__m128i *) (a + i)));
    }
    _mm_storeu_si128((__m128i *) result, prod_v);
    for (int i = n/4 * 4; i < n; i++) { // Handle tail case
        result[0] *= a[i];
    }
    return result[0] * result[1] * result[2] * result[3];
}
```