1 Vector Intrinsics and Instruction-Level Parallelism

1.1 Analyzing slide 14

Slide 14 explains how we can utilize vector instrinsics to transpose 8x8 matrices. Here is the code:

```
// transpose 8 x 8 matrix with intrinsics
inline void transpose_8x8_int_intrinsics(__m256i *vecs) {
    __m256 *v = reinterpret_cast<__m256 *>(vecs);
    __m256 a = _mm256_unpacklo_ps(v[0], v[1]);
    __m256 b = _mm256_unpackhi_ps(v[0], v[1]);
    __m256 c = _mm256_unpacklo_ps(v[2], v[3]);
    __m256 d = _mm256_unpackhi_ps(v[2], v[3]);
    __m256 e = _mm256_unpacklo_ps(v[4], v[5]);
    __m256 f = _mm256_unpackhi_ps(v[4], v[5]);
    __m256 g = _mm256_unpacklo_ps(v[6], v[7]);
    __m256 h = _mm256_unpackhi_ps(v[6], v[7]);
    auto tmp = _mm256_shuffle_ps(a, c, 0x4E);
    a = _mm256_blend_ps(a, tmp, 0xCC);
    c = _mm256_blend_ps(c, tmp, 0x33);
    tmp = _mm256_shuffle_ps(b, d, 0x4E);
    b = _mm256_blend_ps(b, tmp, 0xCC);
    d = _mm256_blend_ps(d, tmp, 0x33);
    tmp = _mm256_shuffle_ps(e, g, 0x4E);
    e = _mm256_blend_ps(e, tmp, 0xCC);
    g = _mm256_blend_ps(g, tmp, 0x33);
    tmp = _mm256_shuffle_ps(f, h, 0x4E);
    f = _mm256_blend_ps(f, tmp, 0xCC);
    h = _mm256_blend_ps(h, tmp, 0x33);
    v[0] = _mm256_permute2f128_ps(a, e, 0x20);
    v[1] = _mm256_permute2f128_ps(c, g, 0x20);
    v[2] = _mm256_permute2f128_ps(b, f, 0x20);
    v[3] = _mm256_permute2f128_ps(d, h, 0x20);
    v[4] = _mm256_permute2f128_ps(a, e, 0x31);
    v[5] = _mm256_permute2f128_ps(c, g, 0x31);
    v[6] = _mm256_permute2f128_ps(b, f, 0x31);
    v[7] = _mm256_permute2f128_ps(d, h, 0x31);
}
```

Let us analyze how we can calculate a specific column, say $v[0]$. It depends on the variables a and e, so let us track them:

We use

```
    __m256 a = _mm256_unpacklo_ps(v[0], v[1]);
```

in the first step (similar for e). Here, we unpack the first two rows, which means we interleave them (compare to last week). *lo* specifies that we focus on the *lower order bits* $(a[\bullet][0], a[\bullet][1], a[\bullet][4], a[\bullet][5])$. Hence, after this operation, a and e contain the following matrix elements:

```
    a = {a00, a10, a01, a11, a04, a14, a05, a15};
    e = {a40, a50, a41, a51, a44, a54, a45, a55};
```

For the next operation, we calculate *tmp*, for which we need variable c in case of a, so let us quickly do that:

```
    c = {a20, a30, a21, a31, a24, a34, a25, a35};
```

Now we can calculate *tmp* with:

```
    auto tmp = _mm256_shuffle_ps(a, c, 0x4E);
```

We have

```
     0x4E = 0b01001110 (= 1 0 3 2)      ,
```

and hence *tmp* becomes:

```
    tmp = {a01, a11, a20, a30, a05, a15, a24, a34};
```

Let us visualize *tmp*:

	1				5
	2				6
3				7	
4				8	

Next, we calculate a using:

```
     a = _mm256_blend_ps(a, tmp, 0xCC);
```

Note that

```
     0xCC = 0b11001100      .
```

Hence, we get after this blending operation, which selects elements based on the mask:

```
    a = {a00, a10, a20, a30, a04, a14, a24, a34};
    e = {a40, a50, a60, a70, a44, a54, a64, a74};
```

From here, we can see that we are almost done, a and e look like they almost contain one column, in fact the contain two columns "cut in the middle". Hence, after a final permutation we should be done calculating a column. To this end, we use `_mm256_permute2f128_ps` and specify that we want to use the lower bits of both a and e by setting the mask to `CC`:

```
    v[0] = _mm256_permute2f128_ps(a, e, 0x20);
    v[0] = {a00, a10, a20, a30, a40, a50, a60, a70};
```

This explanation was not very detailed, but it gives a guideline to get a *feel* for the algorithm, which might improve one's understanding of this algorithm.

1.1.1 Latency and Throughput

Latency measures the time it takes for an intrinsic function to produce its result once it starts executing, typically expressed in clock cycles.

Throughput indicates the rate at which the function can be executed, showing how many operations can be completed per unit of time (e.g., cycles per instruction) when pipelined.

Obviously, the lower the latency, and the the higher the throughput, the better. In certain situations, however, one metric is much more decisive than the other:

If one writes dependent code, i.e. the next line of code may only be executed after the current one because of a dependency, the latency of the operation will be the main factor for performance, since we want our result as quickly as possible to procede with the execution.

On the other hand, if we write independent code which also uses many repeated operations, throughput will be key, as it determines how many clock cycles we approximately need to finish all calculations. In this case, the latency has a neglectable influence on the performance, especially if the number of operations grows bigger.

1.1.2 Analyzing Vectorized Hash Tables Across CPU Architectures

In this paper the authors explained how to employ SIMD instructions to efficiently implement hash tables. Hash tables are quite diverse, as different implementation approaches can be used. One common implementation employs linear probing to search through the key array to find the specified key, so that the desired value can be returned for reading. Here is how we might implement this using vector instructions:

Vectorized Linear Probing The keys should be integer values, so that they can fit comfortably into the vector registers. Linear probing starts by checking the index specified by the hash value of the current key first, and then keeps going one step at a time until we found our key. The ”key” idea is to simply load multiple consecutive keys into one vector register, and to compare them all against the search key. If one comparison is successful, we can calculate the index and return tha value at that index. Otherwise, we keep probing with the next set of keys.

The authors also mentioned that there are subtle challenges, as vector instructions require the data in memory to be correctly aligned (by their register size). When we start probing at the hash index, however, we are almost certainly not at an aligned address. Hence, we should ensure to ”align ourselves”, for example by loading the aligned memory region containing the first search index into a vector register.

Vectorized Fingerprinting As mentioned, vectorized linear probing require integer keys. So what could we do if wanted to use strings instead? The ”key” idea is to calculate ”keys” for the keys themself (quite many keys around here). One can think of these keys as hash values (though not necessarily the same as the real hashes) of the keys. To avoid confusion, they are called *fingerprints* instead, and are quite short, in the paper the authors used fingerprints of size 8 or 16 bits. Using this technique, we can use vectorized linear probing on these fingerprints as described earlier, with the only difference being that we might run into fingerprint-conflicts. Thus, after a successful match of fingerprints, we have to then compare the actual keys to determine an actual hit.