Computer Architecture

CSEE 4824 HWX

Tailai Zhang & XXXXXXXXX

February 1, 2024

Please note that this document has been modified. Contents from the other contributors have been removed to reflect my personal contribution to the write-up.

Problem 2

(2.1) The naive implementation of multiplying two arbitrary-sized vectors is shown below:

```
int32_t vector_multiply_raw(int32_t *a, int32_t *b, size_t length)
{
    int32_t result = 0;
    for (int i = 0; i < length; i++)
    {
        result += a[i] * b[i];
    }
    return result;
}
```

(2.2-1) The vectorized implementation of multiplying two arbitrary-sized vectors is shown below. Note that this implementation requires input to be aligned on a 32-byte boundary for SIMD:

```
int32_t vector_multiply_32bit(int32_t *a, int32_t *b, size_t length)
{
    __m256i sum = _mm256_setzero_si256();
    int32_t cursor = 0;

    for ( ;cursor < (length & ~(size_t)7); cursor += 8)
    {
        __m256i a_vec256 = _mm256_load_si256((__m256i*)&a[cursor]);     // Load next 8 integers from a
        __m256i b_vec256 = _mm256_load_si256((__m256i*)&b[cursor]);     // Load next 8 integers from b
        __m256i result_vec256 = _mm256_mullo_epi32(a_vec256, b_vec256); // Multiply pairs of integers
        sum = _mm256_add_epi32(sum, result_vec256);                     // Add to the running total
    }

    // Dealing leftover of length % 8
    if(cursor < length)
    {
        // Add Padding
        __attribute__((aligned(32))) int32_t tail_a[8] = {0, 0, 0, 0, 0, 0, 0, 0};
        __attribute__((aligned(32))) int32_t tail_b[8] = {0, 0, 0, 0, 0, 0, 0, 0};
        memcpy(tail_a, &a[cursor], (length & (size_t)7) * sizeof(int32_t));
        memcpy(tail_b, &b[cursor], (length & (size_t)7) * sizeof(int32_t));

        // Same as above
        __m256i a_vec256 = _mm256_load_si256((__m256i*)tail_a);
        __m256i b_vec256 = _mm256_load_si256((__m256i*)tail_b);
        __m256i result_vec256 = _mm256_mullo_epi32(a_vec256, b_vec256);
        sum = _mm256_add_epi32(sum, result_vec256);
    }

    // Horizontal sum, see intel doc
    sum = _mm256_hadd_epi32(sum, sum);
    sum = _mm256_hadd_epi32(sum, sum);

    // Now the sum[0] and sum[4] contain the sums of the first and second 128-bit sectors in the m256i
    return _mm256_extract_epi32(sum, 0) + _mm256_extract_epi32(sum, 4);
}
```

(2.2-2) Below is an alternative, less optimal implementation for vector multiplication that doesn't require the input data to be memory-aligned. However, it performs less optimally in terms of speed when compared to the first implementation. Performance of the two multiplication methods will be discussed in later sections.

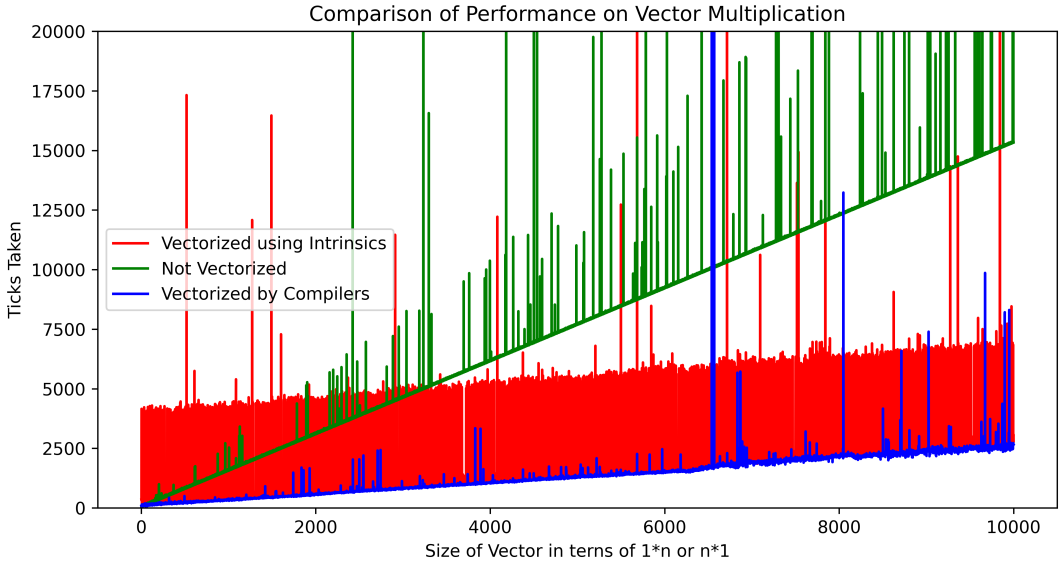
```c
int32_t vector_multiply_32bit(int32_t *a, int32_t *b, size_t length)
{
    __m256i sum = _mm256_setzero_si256();
    int32_t cursor = 0;

    for ( ;cursor < (length & ~(size_t)7); cursor += 8)
    {
        __m256i a_vec256 = _mm256_loadu_si256((__m256i*)&a[cursor]);    // Load next 8 integers from a
        __m256i b_vec256 = _mm256_loadu_si256((__m256i*)&b[cursor]);    // Load next 8 integers from b
        __m256i result_vec256 = _mm256_mullo_epi32(a_vec256, b_vec256); // Multiply pairs of integers
        sum = _mm256_add_epi32(sum, result_vec256);                     // Add to the running total
    }

    // Dealing leftover of length % 8
    if(cursor < length)
    {
        // Add Padding
        int32_t tail_a[8] = {0, 0, 0, 0, 0, 0, 0, 0};
        int32_t tail_b[8] = {0, 0, 0, 0, 0, 0, 0, 0};
        memcpy(tail_a, &a[cursor], (length & (size_t)7) * sizeof(int32_t));
        memcpy(tail_b, &b[cursor], (length & (size_t)7) * sizeof(int32_t));

        // Same as above
        __m256i a_vec256 = _mm256_loadu_si256((__m256i*)tail_a);
        __m256i b_vec256 = _mm256_loadu_si256((__m256i*)tail_b);
        __m256i result_vec256 = _mm256_mullo_epi32(a_vec256, b_vec256);
        sum = _mm256_add_epi32(sum, result_vec256);
    }

    /* Alternatively, Use Horizontal Sum, Speed improvement is not visible unless at very low vector length */
    // sum = _mm256_hadd_epi32(sum, sum);  sum = _mm256_hadd_epi32(sum, sum);
    // int32_t* result = (int32_t*)&sum;   int32_t final_sum = result[0] + result[4];
    // return final_sum;

    int32_t result[8];
    _mm256_storeu_si256((__m256i*)result, sum);
    int32_t final_sum = result[0] + result[1] + result[2] + result[3] + result[4] + result[5] + result[6] + result[7];

    return final_sum;
}
```

Correctness for Horizontal Sum Note the use of horizontal sum in the first implementation. `_mm256_hadd_epi32` takes argument a and b in $\{a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8\}$ and $\{b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8\}$ and returns $\{(a_1 + a_2), (a_3 + a_4), (b_1 + b_2), (b_3 + b_4), (a_5 + a_6), (a_7 + a_8), (b_5 + b_6), (b_7 + b_8)\}$. Thus easy to see after two rounds of horizontal sum, we have $(a_1 + a_2 + a_3 + a_4)$ at first `int32_t` and $(a_5 + a_6 + a_7 + a_8)$ at fifth `int32_t`.

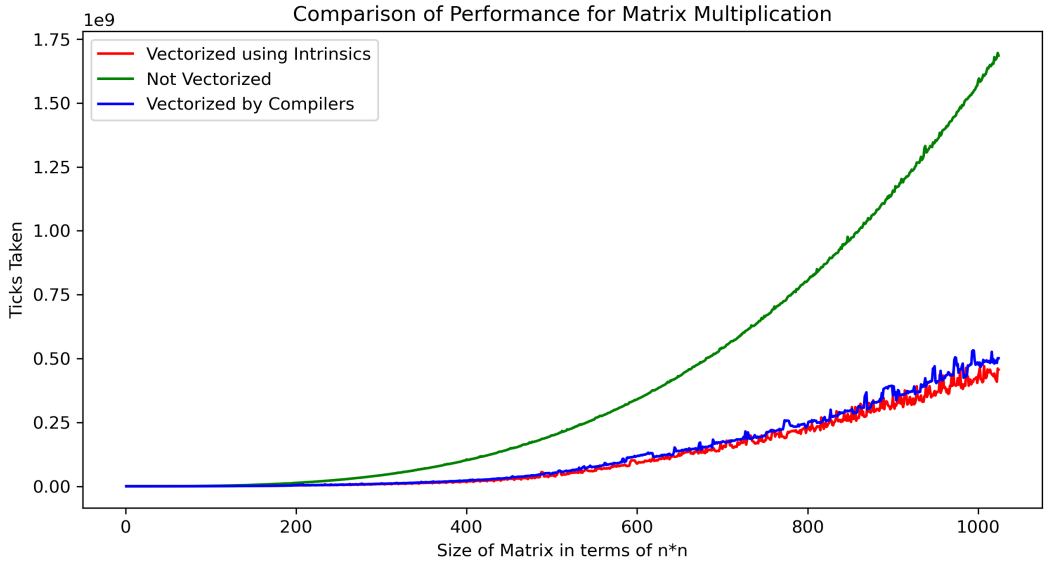
(2.3-1 Result) For comparisons we prepared three sets of data: (1)Vectorized code compiled without compiler's vectorization flags set. (2)Baseline code compiled without compiler's vectorization flags set, and (3)Baseline code compiled with compiler vectorization. The discussion below is for the first (optimized) implementation I proposed.

Our vectorization involves using `__m256i` intrinsic to parallelly manage **eight** `int32_t` integers for both multiplication and addition tasks. Assuming SIMD operations and individual `int32_t` operation takes equal processing time, the anticipated outcome was for the vectorized execution to be approximately one-eighth as time-consuming as the non-vectorized baseline. However, in practice, the vectorized implementation completed in a time frame that was merely between one-fifth to one-sixth compared to the non-vectorized baseline. When compared against the compiler-optimized baseline code, the graph shows that our manually vectorized code displays a noticeably higher incidence of outliers, almost forming a band-like visualization. Yet, for smaller values, the

performance of our manually vectorized code aligns closely with that of the compiler-optimized baseline, which correlates linearly with the size of the vector to the tick count. See graph below:



In addition to testing on multiplying over randomly-generated vector with varying sizes, we also performed tests on their performances on multiplications with randomly generated matrices. Remarkably, we discovered that our manual vectorization method slightly outperforms the compiler-optimized baseline code when dealing with matrix multiplication, even though our vectorized code shows less speed for multiplying single vectors. See graph below:

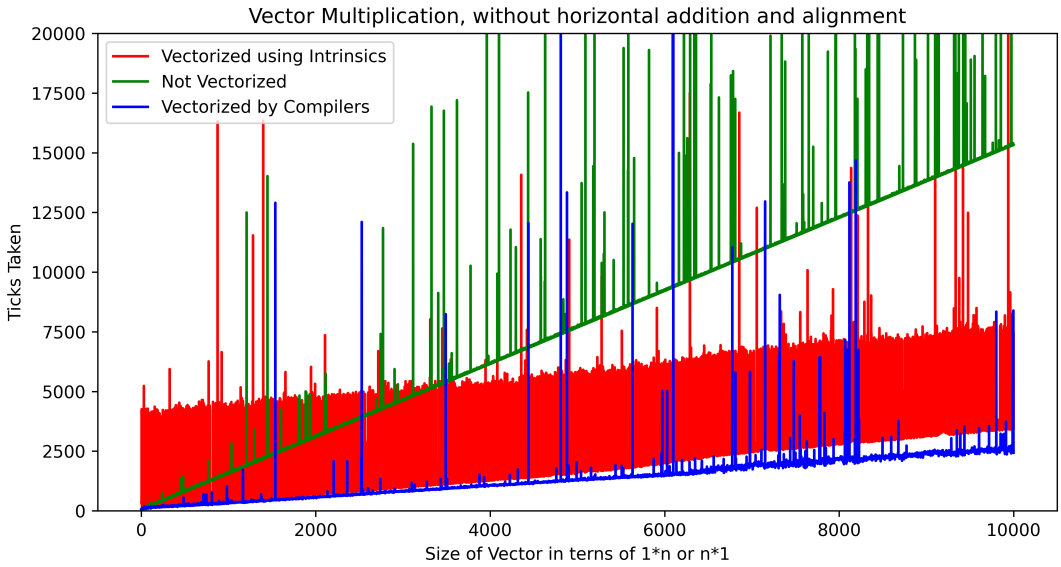


(2.3-1 Analysis) The anticipated eight-fold speed improvement from `__m256i` may not be realized due to the assumption that SIMD operations and individual operation takes equal processing time is wrong, latencies and throughputs associated with SIMD instructions might be different in nature. The interplay between vectorization overhead and instruction latency and throughput may offset the speedup, and may be critical in why the actual performance is lower than predictions.

We also noticed that our vectorized code performs better than compiler-optimized baseline code in matrix multiplication, while this is not the case in single vector multiplication, we think there might be two reasons:

- **Overhead in Intrinsic Setup:** For operations on single vectors, which complete at most in the order of thousands ticks, the setup cost for SIMD operations may overshadow the computational time saved through parallel execution(and we think is likely the source of the band-like visualization). In contrast, matrix multiplication operations, which can run into the range of hundreds of millions of ticks, makes the overhead per operation significantly smaller in proportion..
- **Compiler Optimization:** Compiler optimizations may treat single-pointer (`int*`) and double-pointer (`int**`) operations differently, leading to different performance during matrix multiplications compared to vector multiplications. Also the compiler may be using different instruction sets. We used `AVX2` for our vectorized code, the compiler may used a different instruction set which may perform better in vector multiply but not in matrix multiply.

(2.3-2) Result We conducted similar tests for the alternative, less optimal implementation for vector multiplication that doesn't need memory alignment of input data. Noticeably, the lower boundary of the performance 'band' for **(2.3-2)** remains consistently above that of the compiler-vectorized baseline code, indicating slower performance compared to both the baseline code and the optimized **(2.3-1)** code. Graphs are shown below:



(2.3-2 Analysis) The slowdown of code in **(2.3-2)** mainly comes from the `_mm256_loadu_si256` intrinsic used. It allows the loading of data from any memory address, regardless of its alignment. Yet unaligned memory accesses are typically slower because they may require additional processing to handle memory that is not aligned to the expected boundaries. If unaligned loads cross a cache line or a page boundary, multiple memory accesses may be required to fetch the complete vector data, further reducing efficiency. Despite it being slower, no memory alignment requirement means it can be more flexible in usage than **(2.3-1)**. And this is why I am presenting this code as well despite its inferior performance.

Supplementary Materials

For reference, all source codes, raw data, and executable files are included in the supplementary zip file named 4824-HW3_Code.zip. Please refer to README.txt in the zip file for more information.