Lecture 3 Exercise - SIMD & Vectorization

Exercise 1

To measure the speedup of the vectorized code in C, I experimented with 5 run on each version of the code, Remove minimum and maximum result of that 5 run and calculate the geometric mean of the remaining 3 run. The result is shown in the table below.

Code | Result

Version	Time (s)	Speedup
Scalar (No AVX)	0.056684	1.00
Vector (w/ AVX2)	0.009861	5.75

Exercise 2

To measure the speedup of the vectorized code in Python, I experimented with 5 run on each version of the code, Remove minimum and maximum result of that 5 run and calculate the geometric mean of the remaining 3 run. The result is shown in the table below.

Code | Result

Version	Time (s)	Speedup
Scalar (Python)	0.556130	1.00
Vector (Numpy)	0.049630	11.21

Exercise 3

Vectorization may not be beneficial in these situations:

- Some compilers might not be able to vectorize the code automatically, so we have to explicitly write the vectorized code with knowledge of the computer architecture.
- Some programs are not suitable for vectorization, for example, programs that have a lot of conditions or branches, or programs that have a lot of data dependencies, vectorization becomes complex.

Appendix

Exercise 1

```c
#include <stdio.h>
#include <time.h>
```

```
#include <math.h>
#include <smmintrin.h>
#include <immintrin.h>

// No AVX
void add(int size, int *a, int *b)
{
    for (int i = 0; i < size; i++)
    {
        a[i] += b[i];
    }
}

// with AVX2
void add_avx(int size, int *a, int *b)
{
    int i = 0;
    for (; i < size; i += 8)
    {
        // load 256-bit chunks of each array
        __m256i av = _mm256_loadu_si256((__m256i *)&a[i]);
        __m256i bv = _mm256_loadu_si256((__m256i *)&b[i]);
        // add each pair of 32-bit integers in chunks
        av = _mm256_add_epi32(av, bv);

        // store 256-bit chunk to a
        _mm256_storeu_si256((__m256i *)&a[i], av);
    }
    // clean up
    for (; i < size; i++)
    {
        a[i] += b[i];
    }
}

float benchmark(void (*f)(int, int *, int *), int num_test)
{
    float results[num_test], min = 100., max = 0.;

    const int SIZE = 1e6;
    int a[SIZE], b[SIZE];

    for (int i = 0; i < num_test; i++)
    {
        float startTime = (float)clock() / CLOCKS_PER_SEC;
        (*f)(SIZE, a, b);
        float endTime = (float)clock() / CLOCKS_PER_SEC;
        float timeElapsed = endTime - startTime;
        results[i] = timeElapsed;

        if (timeElapsed < min)
            min = timeElapsed;

        if (timeElapsed > max)
            max = timeElapsed;
    }

    float result = 1;
    for (int i = 0; i < num_test; i++)
    {
        if (results[i] == min)
            continue;
        if (results[i] == max)
            continue;
        result *= results[i];
    }

    return pow(result, 1. / (num_test - 2));
}

int main()
{
    float add_result = benchmark(add, 5);
    float add_avx_result = benchmark(add_avx, 5);
    printf("add benchmark=%f s\n", add_result);
    printf("add_avx benchmark=%f s\n", add_avx_result);
    printf("speedup=%f\n", add_result / add_avx_result);

    // Write to `ex1.txt`
    FILE *fptr;

    fptr = fopen("ex1.txt", "w");

    fprintf(fptr, "add benchmark=%f s\n", add_result);
    fprintf(fptr, "add_avx benchmark=%f s\n", add_avx_result);
    fprintf(fptr, "speedup=%f\n", add_result / add_avx_result);

    fclose(fptr);

    return 0;
}
```

Exercise 2

```python
import numpy as np
import time
import mlx.core as mx


def add_python(size):
    a = list(range(size))
    b = list(range(size))

    start = time.time()
    for i in range(size):
        a[i] += b[i]
    end = time.time()

    return end - start


def add_numpy(size):
    na = np.random.randint(1, 1000, size)
    nb = np.random.randint(1, 1000, size)

    start = time.time()
    na += nb
    end = time.time()

    return end - start


def add_mlx(size):
    ma = mx.array(np.random.randint(1, 1000, size))
    mb = mx.array(np.random.randint(1, 1000, size))

    start = time.time()
    mx.add(ma, mb, stream=mx.cpu)
    end = time.time()

    return end - start


def benchmark(func, size, num_tests=5):
    results = []

    for i in range(num_tests):
        start = time.time()
        func(size)
        end = time.time()
        results.append(end - start)

    reported_results = results.copy()
    reported_results.remove(max(results))
    reported_results.remove(min(results))

    return np.prod(reported_results) ** (1.0 / len(reported_results))


def main(args):
    SIZE = 6400000
    add_python_result = benchmark(add_python, SIZE)
    add_numpy_result = benchmark(add_numpy, SIZE)

    print(f"add_python={add_python_result} s")
    print(f"add_numpy={add_numpy_result} s")
    print(f"speedup={add_python_result/add_numpy_result}")

    f = open("ex2.txt", "w")
    f.write(f"add_python={add_python_result} s\n")
    f.write(f"add_numpy={add_numpy_result} s\n")
    f.write(f"speedup={add_python_result/add_numpy_result}\n")

    if args.extra:
        add_mlx_result = benchmark(add_mlx, SIZE)
        print(f"add_mlx={add_mlx_result} s")
        print(f"speedup={add_python_result/add_mlx_result}")

        f.write(f"add_mlx={add_mlx_result} s\n")
        f.write(f"speedup={add_python_result/add_mlx_result}\n")

    f.close()


if __name__ == "__main__":
    import argparse

    parser = argparse.ArgumentParser(description="Benchmark addition")
    parser.add_argument("--extra", action="store_true", help="Run extra benchmarks")

    args = parser.parse_args()

    main(args)
```