Exercise I

Design an experiment to measure the speed up of this vectorize code.

```
// 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];
    }
}
```

Solution

- Chip: Apple M2
- SIMD: ARM NEON
- Array Size: 10M elements (40MB)
- Iterations: 100 iterations/run
- Runs: 3 runs
- Compiler: `clang` with `-O3 -arch arm64 -fno-vectorize`
  - `-fno-vectorize` flag is used to disable auto-vectorization by the compiler. (if not disabled, the speedup will approximately be 0.95x)

```
#include <arm_neon.h>
#include <stdio.h>
#include <stdlib.h>
#include <mach/mach_time.h>

// Scalar (non-vectorized) version
void add(int size, int *a, int *b) {
```

```
    for (int i = 0; i < size; i++) {
        a[i] += b[i];
    }
}

// ARM NEON vectorized version (4 elements per iteration)
void add_neon(int size, int *a, int *b) {
    int i = 0;
    for (; i <= size - 4; i += 4) {
        int32x4_t av = vld1q_s32(a + i);
        int32x4_t bv = vld1q_s32(b + i);
        av = vaddq_s32(av, bv);
        vst1q_s32(a + i, av);
    }
    // Handle remaining elements
    for (; i < size; i++) {
        a[i] += b[i];
    }
}

int main() {
    const int SIZE = 10000000;  // 10 million elements
    const int ITERS = 100;
    const int RUNS = 3;
    mach_timebase_info_data_t timebase;
    mach_timebase_info(&timebase);

    // Allocate aligned memory for NEON
    int *a, *b, *a_backup;
    posix_memalign((void **)&a, 16, SIZE * sizeof(int));
    posix_memalign((void **)&b, 16, SIZE * sizeof(int));
    posix_memalign((void **)&a_backup, 16, SIZE * sizeof(int));

    if (!a || !b || !a_backup) {
        fprintf(stderr, "Memory allocation failed\n");
        return 1;
    }

    // Initialize with random values
    for (int i = 0; i < SIZE; i++) {
        a[i] = arc4random();
        b[i] = arc4random();
        a_backup[i] = a[i];  // Backup for resetting
    }

    double scalar_times[RUNS];
```

```
double neon_times[RUNS];

for (int run = 0; run < RUNS; run++) {
    // Benchmark scalar version
    double total_scalar = 0;
    for (int i = 0; i < ITERS; i++) {
        // Reset array
        for (int j = 0; j < SIZE; j++) a[j] = a_backup[j];

        uint64_t start = mach_absolute_time();
        add(SIZE, a, b);
        uint64_t end = mach_absolute_time();
        total_scalar += (end - start) * (double)timebase.numer / timebase.denom;
    }

    // Benchmark NEON version
    double total_neon = 0;
    for (int i = 0; i < ITERS; i++) {
        // Reset array
        for (int j = 0; j < SIZE; j++) a[j] = a_backup[j];

        uint64_t start = mach_absolute_time();
        add_neon(SIZE, a, b);
        uint64_t end = mach_absolute_time();
        total_neon += (end - start) * (double)timebase.numer / timebase.denom;
    }

    // Store results for this run
    scalar_times[run] = total_scalar / ITERS / 1e6;  // ms
    neon_times[run] = total_neon / ITERS / 1e6;
}

// Calculate final averages
double avg_scalar = 0, avg_neon = 0;
for (int i = 0; i < RUNS; i++) {
    avg_scalar += scalar_times[i];
    avg_neon += neon_times[i];
}
avg_scalar /= RUNS;
avg_neon /= RUNS;
double avg_speedup = avg_scalar / avg_neon;

// Print individual run results
printf("Benchmark Results (10M elements, 100 iterations/run):\n");
for (int i = 0; i < RUNS; i++) {
    printf("Run %d: Scalar=%.2f ms, NEON=%.2f ms, Speedup=%.2fx\n",
```

```
            i + 1, scalar_times[i], neon_times[i],
            scalar_times[i] / neon_times[i]);
    }

    // Print final averages
    printf("\nAverage across %d runs:\n", RUNS);
    printf("Scalar: %.2f ms\n", avg_scalar);
    printf("NEON:   %.2f ms\n", avg_neon);
    printf("Speedup: %.2fx\n", avg_speedup);

    free(a);
    free(b);
    free(a_backup);
    return 0;
}
```

```
clang -O3 -arch arm64 -fno-vectorize -o ex1 ex1.c
./ex1
```

```
Benchmark Results (10M elements, 100 iterations/run):
Run 1: Scalar=4.79 ms, NEON=1.82 ms, Speedup=2.64x
Run 2: Scalar=4.75 ms, NEON=1.57 ms, Speedup=3.02x
Run 3: Scalar=4.76 ms, NEON=1.58 ms, Speedup=3.02x

Average across 3 runs:
Scalar: 4.76 ms
NEON:   1.65 ms
Speedup: 2.88x
```

Conclusion

The vectorized implementation using ARM NEON SIMD instructions demonstrated a significant performance improvement over the scalar implementation, achieving an average speedup of 2.88x.

Exercise II

Design an experiment to measure the speed up of this in numpy (and cupy it you have a GPU)

```
a=list(range(6400000))
b=list(range(6400000))
for i in range(size):
    a[i]+=b[i]
```

```
import numpy as np
na=np.random.randint(1,1000, 6400000)
```

```
nb=np.random.randint(1,1000, 6400000)
na+=nb
```

Solution

Setup of experiment is in `config`.

```
import timeit
import numpy as np
import sys

def main():
    # Benchmark configuration
    config = {
        'size': 6_400_000,      # Number of elements
        'dtype': np.int64,      # Data type
        'repeats': 5,           # Number of benchmark repeats
        'numpy_iterations': 1000,  # Iterations per numpy run
        'random_seed': 42       # For reproducibility
    }

    # System information
    print(f"Python {sys.version}\nNumPy {np.__version__}")
    print(f"Benchmarking {config['size']:,} elements ({config['dtype'].__name__})")

    # Initialize data with fixed seed
    np.random.seed(config['random_seed'])
    a_np = np.random.randint(1, 1000, config['size'], dtype=config['dtype'])
    b_np = np.random.randint(1, 1000, config['size'], dtype=config['dtype'])
    a_list = a_np.tolist()
    b_list = b_np.tolist()

    # Benchmark 1: Python list iteration
    list_times = timeit.repeat(
        stmt='for i in range(len(a)): a[i] += b[i]',
        setup='a = list_a.copy(); b = list_b.copy()',
        globals={'list_a': a_list, 'list_b': b_list},
        number=1,
        repeat=config['repeats']
    )

    # Benchmark 2: NumPy vectorized operation
    numpy_times = timeit.repeat(
        stmt='a += b',
        setup='a = np_a.copy(); b = np_b.copy()',
        globals={'np_a': a_np, 'np_b': b_np},
```

```
        number=config['numpy_iterations'],
        repeat=config['repeats']
    )

    # Calculate statistics
    avg_list_ms = (sum(list_times) / len(list_times)) * 1000  # Convert s to ms
    avg_numpy_ms = (sum(numpy_times) / (len(numpy_times) * config['numpy_iterations'])) * 10
    speedup = avg_list_ms / avg_numpy_ms

    # Print results
    print("\nBenchmark Results (milliseconds):")
    print(f"{'Run':<5} {'List time (ms)':<15} {'NumPy time (ms)':<15}")
    for i, (lt, nt) in enumerate(zip(list_times, numpy_times)):
        list_ms = lt * 1000
        numpy_ms = (nt / config['numpy_iterations']) * 1000
        print(f"{i+1:<5} {list_ms:<15.3f} {numpy_ms:<15.3f}")

    print("\nSummary:")
    print(f"Average List Time:    {avg_list_ms:.3f} ms")
    print(f"Average NumPy Time:   {avg_numpy_ms:.3f} ms")
    print(f"Speedup Factor:       {speedup:,.0f}x")

if __name__ == '__main__':
    main()
```

```
Python 3.11.11 (main, Dec 11 2024, 10:25:04) [Clang 14.0.6 ]
NumPy 1.26.4
Benchmarking 6,400,000 elements (int64)

Benchmark Results (milliseconds):
Run   List time (ms)  NumPy time (ms)
1     265.813         2.723
2     201.187         2.739
3     198.033         2.762
4     197.740         2.826
5     197.282         2.710

Summary:
Average List Time:    212.011 ms
Average NumPy Time:   2.752 ms
Speedup Factor:       77x
```

Conclusion

The NumPy vectorized operation demonstrated a significant performance improvement over the Python list iteration, achieving an average speedup factor

of 77x. This highlights the efficiency of vectorized operations in NumPy for element-wise array computations.

Exercise III

While vectorization is powerful, please explain a situation when it may not be beneficial.
Hint 1 - Compiler support
Hint 2 - Vectorizability of Software

Solution

Vectorization may not always provide performance benefits, especially in the following scenarios:

1. **Lack of Compiler Support**: If the target platform's compiler lacks robust support for vectorization or SIMD instructions (e.g., ARM NEON, AVX), the code may not benefit from vectorization. This is particularly true for older compilers or architectures that do not support the necessary instruction sets.
2. **Non-Vectorizable Code**: Certain algorithms inherently cannot be vectorized due to dependencies between iterations, such as recursive computations, dynamic memory access patterns, or irregular data structures (e.g., linked lists). In such cases, vectorized instructions cannot be applied effectively, and scalar operations remain more suitable.
3. **Small Data Sizes**: When the input data is too small, the overhead of setting up vectorized operations can outweigh the performance gains, making scalar operations faster.
4. **Memory Alignment**: Vectorized instructions often require data to be aligned in memory for optimal performance. Misaligned or fragmented data can lead to penalties, negating the benefits of vectorization.
5. **Branching Code**: Code with heavy branching (e.g., conditional statements inside loops) can disrupt vectorization, as SIMD instructions operate on uniform sets of data. Divergent execution paths hinder parallelism and reduce the effectiveness of vectorized instructions.