Mason Riley
Homework 1

Optimization of Matrix-Vector Multiplication in C

Introduction

In computer science, the optimization of matrix operations plays a pivotal role in achieving high performance and efficient use of system resources. This study set out to improve the performance of matrix-vector multiplication by applying optimization techniques and evaluating their efficacy. Starting with unoptimized code, improvements such as cache aware optimizations, gcc optimization flags, and Intel SIMD Intrinsics were incrementally added to achieve the greatest performance.

Methods

Design Approach

The design of the experiment was to separate test runs by optimization type - cache aware optimizations, gcc optimization flags, and Intel Intrinsics - and then combine them methodically to produce the best possible performance.

Experimental Setup

Optimization Level	**Description**
Unoptimized	Used as a control/benchmark
Opt1	Cache aware optimizations
Opt2	Cache aware optimizations & gcc optimization flags
Opt3	Intel SIMD Intrinsics
Optimized	Intel SIMD Intrinsics & gcc optimization flags

The above setup resulted in a clear picture of how the various optimizations impact matrix-vector multiplication performance. Runtime and performance for each version was measured using microtime.

Evaluation Strategy

For evaluation, there were three metrics:
1. Execution time (microseconds).
2. Performance (Gflop/s).
3. Correctness of results (C[N/2]).

Implementation Details

Implementing cache aware optimizations and gcc optimization flags were both straightforward. Intel Intrinsics, however, proved challenging at first because I attempted to use 16-byte aligned matrices by creating the matrix with `_mm_malloc` and loading the SIMD registers with `_mm_load_ps`, but this resulted in segmentation faults that I could not diagnose. Instead I opted to use `_mm_uload_ps` and left `createMatrix` untouched, which worked but only on matrices that were divisible by 4. My next challenge was determining that the tail of the matrix was not being processed, which lead to the two following code snippets being added to `matVecMult()`:

```
int simd_end = cols - (cols % 4);
```

```
    for (k = simd_end; k < cols; k++)
        C[i] += A[i * cols + k] * B[k];
```

That loop handles the tail of any matrix where:

$$cols \,\%\, 4 \;\neq\; 0$$

This final code snippet allowed me to process matrices of any size (as long as I have enough memory) and finalized the Intel SIMD Intrinsics implementation.

Results

Note: $Efficiency\ Increase = (\frac{Performance\ of\ the\ Unoptimized\ Version}{Performance\ of\ the\ Optimized\ Version} - 1) \times 100$

Version	**Runtime (µs)**	**Performance (Gflop/s)**	**C[N/2]**	**Efficiency Increase**
Unoptimized	452,414	0.447483	0.00152989	-
Cache-awareness	161,427	1.25411	0.00152989	180.25%
Cache-awareness & Compiler Flags	41,373	4.89323	0.00152989	993.5%
Intel Intrinsics	76,893	2.63285	0.00152988	488.4%
Intel Intrinsics & Compiler Flags	19,205	10.5414	0.00152988	2255.7%

Table 1

```
masonriley@pop-os:~/dev/repos/COP5522/homework/homework_1$ ./run_test.sh
Unoptimized Test

Time = 452414 us
Timer Resolution = 1 us
Performance = 0.447483 Gflop/s
C[N/2] = 0.00152989

Opt1 Test - Cache-awareness

Time = 161427 us
Timer Resolution = 1 us
Performance = 1.25411 Gflop/s
C[N/2] = 0.00152989

Opt2 Test - Cache-awareness & Compiler Flags

Time = 41373 us
Timer Resolution = 1 us
Performance = 4.89323 Gflop/s
C[N/2] = 0.00152989

Opt3 Test - Intel Intrinsics

Time = 76893 us
Timer Resolution = 1 us
Performance = 2.63285 Gflop/s
C[N/2] = 0.00152988

Opt4 Test - Intel Intrinsics & Compiler Flags

Time = 19205 us
Timer Resolution = 1 us
Performance = 10.5414 Gflop/s
C[N/2] = 0.00152988
```

Screenshot showing the data found in Table 1

Results

- Cache awareness optimizations alone provided a 2.5-3x performance boost, demonstrating how important it is to account for C's column-major memory access patterns.
- Compiler flags when combined with cache-awareness showed a nearly 10x performance boost, making them the most impactful change to the code, running even better than the Intel Intrinsics optimizations.
- Using Intel Intrinsics alone yielded significant performance improvements (~500%). Though less dramatic than the gcc flags, this was still an enormous performance boost.
- Combining Intel intrinsics with compiler flags (and cache-awareness) gave the highest performance - over 2200% - showing the compounded effects of hardware-specific operations and gcc flag enhancements.

Conclusion

Optimizing matrix multiplication is a challenging task, and the tools at our disposal, such as cache awareness optimizations, gcc optimization flags, and Intel intrinsics, can make a radical difference. The most optimized approach, using all of the tools at our disposal, saw a performance increase from 0.447 Gflop/s to 10.541 Gflop/s, showcasing the potential of combining hardware and software optimizations.