SPP Assignment-1 Report

Chirag Dhamija
2022101039

March 2024

1 Introduction

Efficiency is a cornerstone in the realm of software development, especially when it comes to computational tasks, underscoring the paramount significance of performance optimization. This comprehensive report undertakes an in-depth examination of performance evaluation within the context of vector multiplication. Its primary objective is to delve into a variety of optimization techniques, thereby offering a nuanced understanding of their respective impacts on overall performance.

2 Setup

The investigation took place on a system housing a 10th Generation Intel i5-1035G1 processor, clocked at 3.60 GHz and equipped with 4 cores and 8 threads. Optimization procedures were conducted using the GCC version 11.4. compiler. Each optimization technique underwent assessment with a variety of compiler flags and manual vectorization methods employing compiler intrinsics. Additionally, OpenMP was applied for parallelization purposes, enhancing computational efficiency.

3 Flags Used

1. -O0 : This flag turns off all optimizations. It produces unoptimized code, which makes debugging easier as the generated code closely corresponds to the original source code.

2. -O1 : This flag enables basic optimizations that don't increase compilation time significantly. It includes optimizations such as removal of unused variables and functions, simplification of expressions, and basic control flow optimizations.

3. -O2 : This flag enables more aggressive optimizations compared to -O1. It includes optimizations like loop unrolling, function inlining, and instruction scheduling. This may increase compilation time but can result in faster executing code.

4. -O3 : This flag enables even more aggressive optimizations compared to -O2. It may include optimizations such as vectorization and more aggressive loop transformations. This can significantly improve performance but may also increase compilation time substantially.

5. -Ofast : This flag enables optimizations beyond -O3 level. It may sacrifice compliance with certain language standards or IEEE floating point precision rules for the sake of performance. Use it with caution as it might produce unexpected results in some cases.

6. -mavx2 : This flag enables the use of AVX2 (Advanced Vector Extensions 2) instructions, which are extensions to the x86 instruction set architecture designed for enhanced performance of floating-point and integer vector operations.

7. -fopenmp : This flag enables support for OpenMP (Open Multi-Processing), which is an API for parallel programming in C, C++, and Fortran. It allows developers to write code that can be executed concurrently on multiple processors or cores.

8. -march=native : This flag tells the compiler to optimize the generated code for the specific CPU architecture on which the compiler is running. It allows utilizing the full capabilities of the CPU, including instruction sets and features specific to that architecture.

4 Results

Table 1: Result

Technique/Optimization	GFLOPSs/s	Wall Clock Time (ms)	Speedup
O0	0.524934	3.810	-
O1	1.156738	1.729	2.20x
O2	1.212121	1.650	2.31x
O3	1.257862	1.590	2.4x
Ofast(other flags)	3.007519	0.665	5.73x
Manual Vectorization + O0	1.569859	1.274	3x
Manual Vectorization + O1	2.125399	0.941	4.05x
Manual Vectorization + O2	4.210526	0.475	8.02x
Manual Vectorization+ O3	4.329004	0.462	8.25x
OpenMp Parallelization + O0	2.522068	0.793	4.8x
OpenMp Parallelization + O1	3.527337	0.567	6.72x
OpenMp Parallelization + O2	3.546099	0.564	6.755x
OpenMp Parallelization + O3	3.809524	0.525	7.26x
Parallelization+ Vectorization+O0	2.582755	0.774	4.92x
Parallelization+ Vectorization+O1	3.133077	0.638	5.96x
Parallelization+ Vectorization+O2	3.641289	0.549	6.94x
Parallelization+ Vectorization+O3	4.012463	0.498	7.64x

(a) OpenMp Parallelization is compiled using -fopenmp flag

(b) Manual Vectorization is compiled using -mavx2 flag

(c) Vectorization is done using -march=native flag

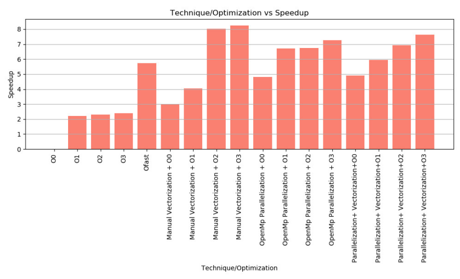

Figure 1: Techniques vs Speedup

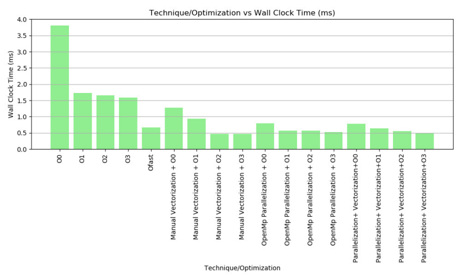

Figure 2: Techniques vs Wall Clock Time

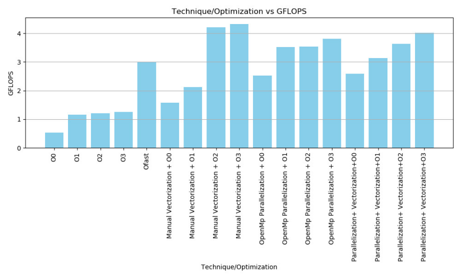

Figure 3: Techniques vs GFLOPS

However, despite the expected increase in GFLOPs due to parallelization and vectorization in O3 aided by manual intrinsics, was less than manual vectorization in O3. This can be attributed to a considerable rise in memory overhead associated with parallel execution. As the number of threads increased, so did the memory overhead, ultimately impacting performance negatively. This situation highlights the critical importance of carefully managing memory resources in parallel computing to mitigate adverse effects on performance.

5 Compute vs Memory Bound

(a) -O0 : This approach is primarily constrained by computational resources. Without any optimizations, the computation extends, resulting in a compute-bound scenario where computation time becomes the dominant factor in overall execution time.

(b) -O1,-O2,-O3 : The optimization levels -O1, -O2, and -O3, ranging from basic to aggressive, primarily address compute-bound scenarios. Compiler optimizations enhance computational efficiency, leading to improved performance. Nonetheless, absent supplementary details, it is logical to infer that computational tasks remain the predominant factor influencing execution time, thereby characterizing these techniques as compute-bound.

(c) Other flags (eg : -march=native, -Ofast, -fopenmp, -mavx2) : While these flags can indirectly impact memory utilization, their primary focus lies in improving computational efficiency. Therefore, they are more closely associated with compute-bound optimizations rather than memory-bound optimizations. However, the overall impact on memory usage may vary depending on the specific characteristics of the code and the underlying hardware architecture.

(d) Manual Vectorization + Flags : This methodology is constrained by computational limitations. Employing manual vectorization through Intel intrinsics facilitates explicit vectorization of the code, thereby enhancing computational efficiency. When coupled with compiler flags, this strategy prioritizes computational optimization, rendering the process compute-bound.

(e) OpenMP Parallelisation : Based on the empirical findings, it is evident that this particular technique exhibits a memory-bound characteristic. The provided report highlights a reduction in GFLOPs alongside an escalation in wall clock time with the implementation of OpenMP parallelization. Such trends indicate that the overhead associated with parallelization, notably in terms of memory utilization, emerges as a bottleneck, thereby inducing a scenario where memory constraints dominate performance.

(f) Vectorization + OpenMP : This approach is prone to being constrained by memory limitations. Despite the objective of vectorization to enhance computational efficiency, the incorporation of parallelization through OpenMP can introduce the possibility of memory bottlenecks, as evidenced in previous instances. The integration of both vectorization and parallelization may not sufficiently alleviate the memory overhead, thereby leading to a situation where memory constraints become prominent.

6 Observations

We employed a range of compiler flags such as -O3, -march=native, -mavx2, -fopenmp, -Ofast for vector multiplication, resulting in a noticeable boost in performance. Additionally, manual vectorization using Intel intrinsics played a significant role in enhancing performance by allowing multiple operations to be executed simultaneously, thereby maximizing throughput.