Enhancing Execution Speed of White Noise Generation through Parallelization and Vectorization

BOUJIDA Nezar

20/01/2024

This study aims to optimize the execution time of white noise generation and processing through Fast Fourier Transform (FFT) techniques. By employing parallelization with OpenMP and SIMD vectorization with AVX2, we seek to significantly improve the computational efficiency of FFT operations.

Context and Execution Environment

The results presented in this report were obtained following the execution of the programs accompanying this report on an Intel® Core™ i5-11300H processor.

Compilation command: `gcc -fopenmp -Wall -O3 -o code_name code_name.c -lm -mavx2`
Execution command: `./code_name --size <N>` (N: power of 2)
Command to change the number of threads: `export OMP_NUM_THREADS=n`

Contents

1 Introduction **2**

2 Focus Areas for Parallelization **2**

3 Achieved Results **2**

4 Parallelization Methods **3**

4.1 Parallelization with OpenMP (code fft_MP.c) 3

4.2 Optimization through Vectorization (code fft_vect.c) 4

4.3 Parallelization with OpenMP and Vectorization (code fft_MP_vect.c) 4

1 Introduction

In this study, we investigate a code that generates white noise, which is initially inaudible in its raw state. To illustrate this, we include the code for generating the white noise `white_noise.c` along with an auditory sample `white_noise.wav`.

By employing the Fast Fourier Transform (FFT), we transform this noise into the frequency domain. This conversion facilitates precise adjustments, such as reducing specific "sharp" or "noisy" frequencies to achieve an optimal auditory quality and modifying phase relationships to improve the coherence of sound components over time.

Subsequent application of the Inverse Fast Fourier Transform (iFFT) results in a smoother auditory experience, demonstrated in the modified output (output.wav).

We analyze 2^{26} data points and aim to optimize the execution time of our process through the use of parallelization with OpenMP and vectorization techniques.

This report includes three versions of the code:

- **fft_MP:** This version of the code is parallelized using OpenMP directives, aiming to reduce execution time by distributing tasks across multiple processors.
- **fft_vect:** In this sequential version, the `FFT_rec()` function has been vectorized using AVX2 intrinsics to improve computational efficiency by performing operations on multiple data points simultaneously.
- **fft_MP_vect:** This code combines the previous two approaches, integrating both parallelization and vectorization to maximize performance improvements.

This segmentation allows us to clearly observe the contributions of each technique to the code's performance, in addition to verifying their compatibility.

2 Focus Areas for Parallelization

The code uses a specialized pseudo-random function (PRF), influenced by cryptographic techniques, to enhance the complexity and quality of generated randomness. This function takes a block of numbers as input, processes it, and outputs a 64-bit unsigned integer. The output is subsequently normalized to fit within the range of -1 to 1. Our parallelization efforts primarily target the for-loop responsible for this normalization process.

Further processing of the signal is conducted through the Fast Fourier Transform (FFT), wherein our interest in parallelization lies within its recursive execution. Additionally, during the normalization phase—aimed at adjusting the signal's amplitudes and phases—our focus shifts to parallelizing the involved for-loop. These strategic points of parallelization are critical in enhancing the efficiency and speed of our computational processes.

3 Achieved Results

Our optimizations significantly improved the execution time of the initial sequential code, which took 70 seconds. Implementing OpenMP reduced this to 15 seconds, achieving a 4.67x speed-up. Further enhancements through vectorization, and its combination with OpenMP, reduced the time to 10 seconds, corresponding to a 7x speed-up. These results underscore the effectiveness of parallelization and vectorization in computational efficiency.

4 Parallelization Methods

4.1 Parallelization with OpenMP (code fft_MP.c)

The application of OpenMP directives significantly optimizes the efficiency of our signal processing tasks, as outlined below:

- **White Noise Generation:** Using the `#pragma omp parallel for` directive parallelizes the white noise generation by evenly distributing loop iterations across available threads. The `schedule(auto)` clause permits the OpenMP runtime to dynamically select the optimal scheduling method.
- **FFT Calculation:** For the Fast Fourier Transform (FFT), the `#pragma omp task` directive facilitates concurrent execution of independent FFT calculations. This approach allows efficient scheduling and execution across threads, particularly advantageous for recursive parts of the FFT.
- **Fourier Coefficients Adjustment:** This step employs the same parallelization strategy as white noise generation, ensuring even workload distribution and computational efficiency.
- **Normalization:** The `#pragma omp parallel for reduction(max:max)` directive is instrumental in efficiently computing the maximum magnitude across all threads. This maximum value is then used to normalize the magnitudes of the Fourier coefficients, demonstrating the effectiveness of parallel reduction in optimizing processing time.

These optimizations via OpenMP not only streamline the processing pipeline but also significantly improve computational performance by leveraging multi-threading.

This graph illustrates that increasing the number of threads initially boosts performance significantly. However, the rate of improvement diminishes with further increases in thread count. This phenomenon can be attributed to several factors.

Primarily, as parts of the program execute sequentially, doubling the number of threads doesn't halve execution time. Furthermore, overhead from thread creation, synchronization, and management becomes more pronounced with more threads, impacting parallelization efficiency. Despite this, the results are deemed satisfactory, achieving an execution time of 15 seconds compared to 70 seconds for the sequential execution, yielding a 4.67x speedup.

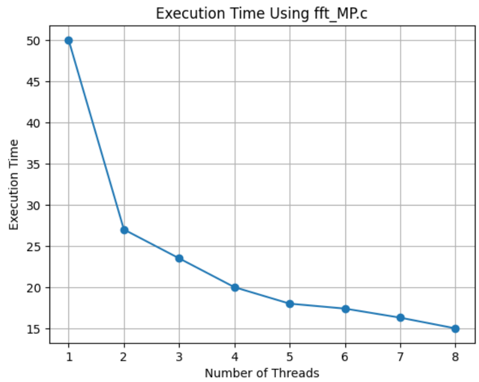

Figure 1: Execution Time of OpenMP Code Across Various Thread Counts

4.2 Optimization through Vectorization (code fft_vect.c)

Given that a significant portion of the execution time was attributed to the FFT and inverse FFT processes, we focused our optimization efforts on vectorizing the `FFT_rec()` function. This involved implementing a vectorized version of `FFT_rec()` and a necessary function, `complex_product()`, for handling complex number multiplication in a vectorized manner.

Complex Product Vectorization

The `complex_product()` function takes two `__m256d` vectors (representing two complex numbers), extracts their real and imaginary parts, performs the complex multiplication, and combines the results to obtain the final complex product. The used data type and functions include:

- **Data Type**: `__m256d` - a vector of 4 doubles.
- **Functions**:
  - `_mm256_add_pd` - adds two `__m256d` vectors.
  - `_mm256_sub_pd` - subtracts two `__m256d` vectors.
  - `_mm256_unpacklo_pd` - extracts the lower half of a `__m256d` vector.
  - `_mm256_unpackhi_pd` - extracts the upper half of a `__m256d` vector.
  - `_mm256_shuffle_pd` - performs a shuffle operation between two `__m256d` vectors.

FFT_rec Vectorization

The vectorized `FFT_rec` function follows the same logic as the original sequential version but adapted for vector operations. It employs the same add and subtract functions as `complex_product`, along with:

- `_mm256_set_pd` - initializes a `__m256d` vector with four doubles.
- `_mm256_loadu_pd` - loads a `__m256d` vector from memory.
- `_mm256_storeu_pd` - stores a `__m256d` vector into memory.

Through vectorization, the execution time was reduced to 13.1 seconds from the original 70 seconds in the sequential code, achieving a speed-up factor of approximately 5.34.

4.3 Parallelization with OpenMP and Vectorization (code fft_MP_vect.c)

This code is essentially a combination of the two previous codes. Our goal here is to test the efficiency and compatibility of vectorization and parallelization with OpenMP. Therefore, we have added the vectorized `FFT_rec()` function to the parallelized code with OpenMP, making sure to include OpenMP directives in the vectorized `FFT_rec()` function which did not initially have them (we based these on the same directives added to the function in the sequential code).

This graph can be interpreted in a manner similar to that of the `fft_MP` section: the increase in the number of threads leads to a significant initial boost in performance.

However, the magnitude of improvement diminishes due to the sequential part of the code and the overhead associated with thread management. Despite these challenges, we successfully reduced the execution time from 70 to 10 seconds, achieving a 7x speed-up. This performance is superior to that of the code utilizing merely MP or solely vectorization techniques.

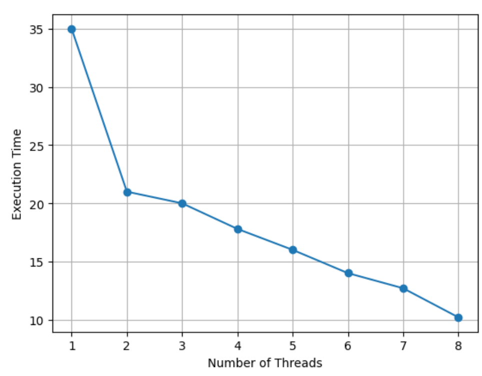

Figure 2: Execution Time of OpenMP Code Across Various Thread Counts