Final Project Testing Report

Hisen & Wyatt, Spring 2023

Test Environment

OS Ubuntu 18.04.5 LTS
Memory 23.3 GiB
Processor AMD Ryzen73800X8-core processor×8
Graphics SVGA3D;build:RELEASE;LLVM;
GNOME 3.28.2
OS type 64-bit
Virtualization Oracle
Disk 42.0 GB

Test Command

Compilation

```
g++ benchmark.cpp -o build/benchmark -O3 -DBENCHMARK -mavx -pthread
-lpulse-simple -lpulse
```

SIMD and vanilla method testing

```
build/benchmark DURATION
```

The numpy and scipy testing

```
python3 benchmark.py DURATION
```

Observation and Analysis

We benchmarked our methods with vanilla implementation and several other common ones. Figure 1 below suggests about 4 times speed boost over a range of signal length (sampled at 44100 Hz). We may conclude our result has achieved significant gain over vanilla. Interestingly, numpy and scipy perform similar to our vanilla method despite they are commonly used python packages. The python foundation probably left it non-optimized for better compatibility on a range of devices.

The parallel method based on Figure 2 using a threadpool was performing less sound than SIMD alone. The smaller the chunk size, the higher the performance penalty, despite the only change in convolution by code is the “+=” operator at chunk boundary. This observation indicates that there’s a potential penalty in memory access pattern compared to pure SIMD, probably due to the extra memory read operations caused by "+=" disrupting the high-speed SIMD CPU bound operations.

However, there's still an opportunity to further speedup the computation. In this project we implemented Inner loop vectorization (VIL). Instead we may adopt Outer loop vectorization (VOL), or a mixture named InnerOuter loop vectorization (VOIL). At a larger scale, it is possible to conduct the convolution task in a distributed system, as the signal is broken into chunks and they are computed independently.

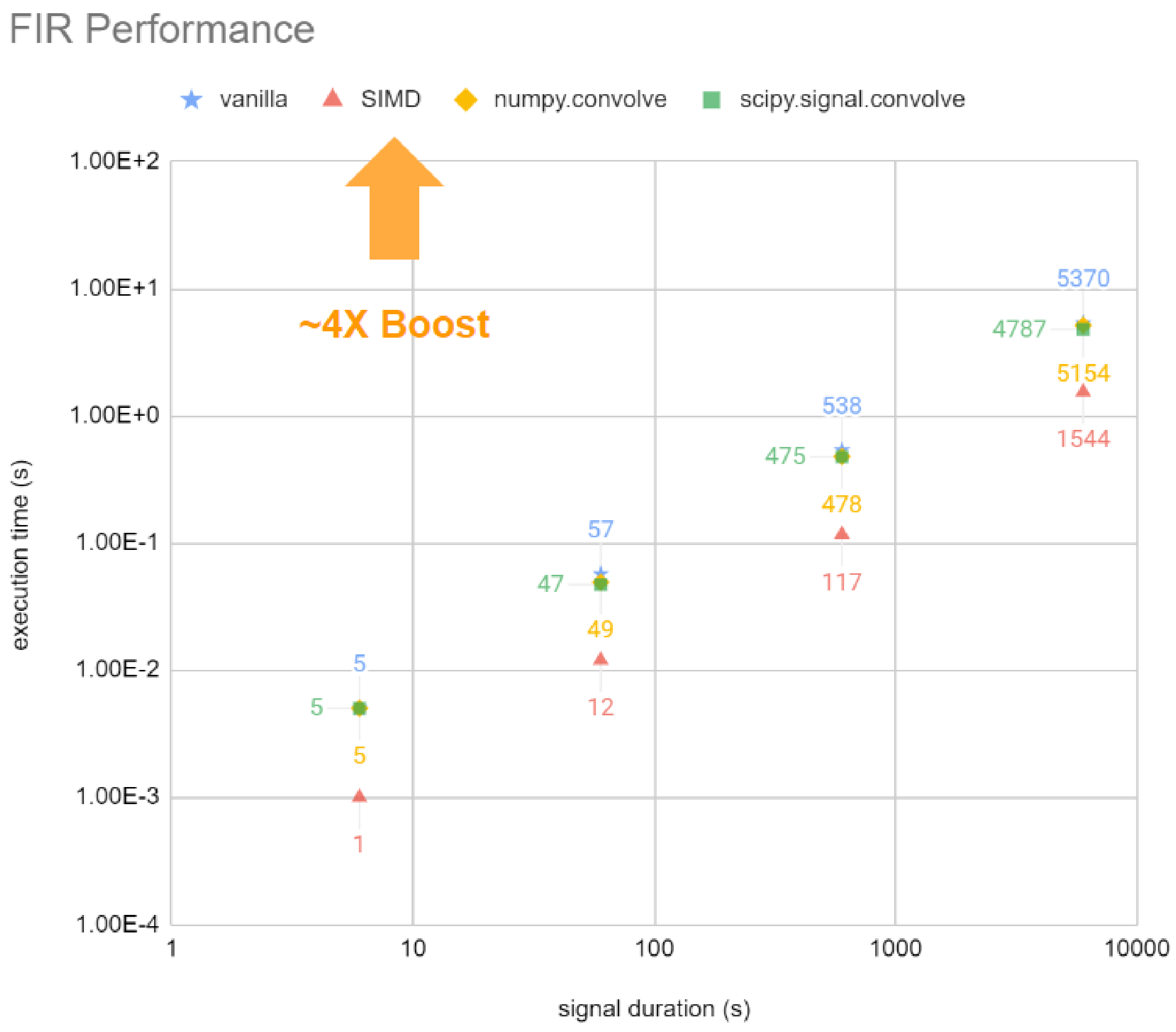
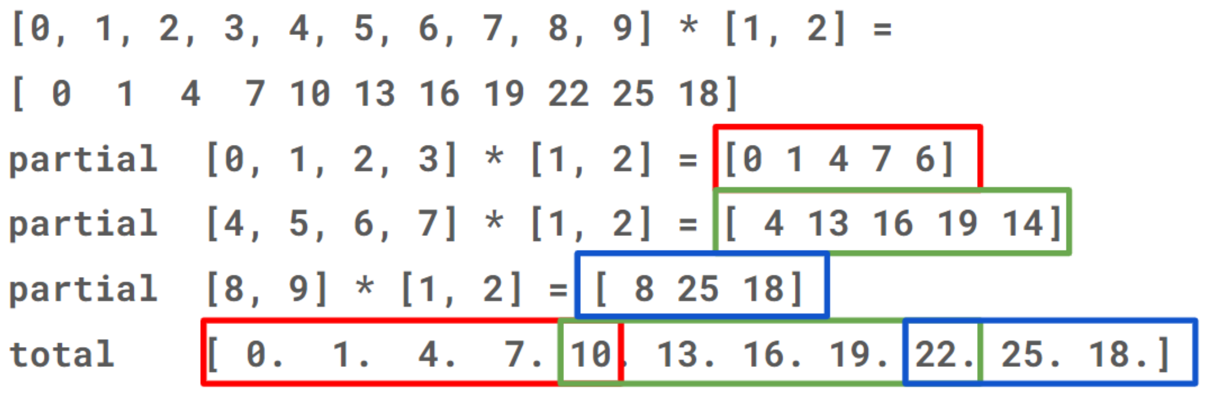


Figure 1. Performance Benchmark



```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9] * [1, 2] =
[ 0  1  4  7 10 13 16 19 22 25 18]
partial  [0, 1, 2, 3] * [1, 2] = [0 1 4 7 6]
partial  [4, 5, 6, 7] * [1, 2] = [ 4 13 16 19 14]
partial  [8, 9] * [1, 2] = [ 8 25 18]
total    [ 0.  1.  4.  7. 10. 13. 16. 19. 22. 25. 18.]
```

Figure 2. Parallel Convolution