

convolve

CPSC 501 A4 Report

The convolve project encompasses multiple versions of a convolution algorithm, each tagged in the Git repository. The initial version (v1.0) implements a straightforward time-domain convolution algorithm. Subsequent versions include algorithm-based optimizations (v2.0), compiler-level optimizations (v3.0), and versions optimized using code tuning, denoted as v4.0, v4.1, v4.2, v4.3, and v4.4. The optimizations lead to significant improvements in execution time, as evidenced by timing results, where v2.0 outperforms v1.0 and v3.0 introduces compiler-level optimizations. Profiling information, including flamegraphs, offers insights into the performance characteristics of each version. Regression testing, executed through `audiodiff.py`, ensures consistency in the output across versions, affirming that the convolution results remain unchanged. The project emphasizes systematic improvement, with all versions and profiling results meticulously documented and accessible in the Git repository.

Build and Run Instructions

Please note that C++11 or greater is required.

1. `cmake -S . -B build.`
2. `cmake --build build.`
3. `./build/src/convolve ./build/src/guitar.wav ./build/src/big_hall_mono.wav ./tests/output.wav`

v1.0 - Baseline Program

The baseline program implements a convolution algorithm directly in the time domain.

```
std::vector<float> Convolver::convolve(const std::vector<float> &x,
                                       const std::vector<float> &h) {
    std::vector<float> y(x.size() + h.size() - 1, 0.0);
    std::cout << "Convolving input signal and impulse response" <<
    std::endl;

    int n, m;

    for (n = 0; n < y.size(); n++) {
        y[n] = 0.0;
    }

    for (n = 0; n < x.size(); n++) {
        for (m = 0; m < h.size(); m++) {
            y[n + m] += x[n] * h[m];
        }
    }

    return y;
}
```

v2.0 - Algorithm-Based Optimization

The algorithm-based optimized program re-implements the convolution algorithm using a frequency-domain FFT convolution algorithm.

```
std::vector<float> Convolver::convolve(const std::vector<float> &x,
                                       const std::vector<float> &h) {
    std::cout << "Convoluting input signal and impulse response" <<
std::endl;

    size_t outputSize = x.size() + h.size() - 1;
    size_t nn = pow(2, ceil(log2(x.size() * 2)));
    size_t k, real, imag;

    std::vector<float> X(nn * 2 + 1, 0.0f);
    std::vector<float> H(nn * 2 + 1, 0.0f);
    std::vector<float> Y(nn * 2 + 1, 0.0f);

    for (k = 0; k < x.size(); k++) {
        real = (k * 2) + 1;
        imag = real + 1;
        X[real] = x[k];
        X[imag] = 0.0f;
    }

    for (k = 0; k < h.size(); k++) {
        real = (k * 2) + 1;
        imag = real + 1;
        H[real] = h[k];
        H[imag] = 0.0f;
    }

    fft(X, nn, 1);
    fft(H, nn, 1);

    for (k = 0; k < nn; k++) {
        real = (k * 2) + 1;
        imag = real + 1;
        float Xr = X[real], Xi = X[imag];
        float Hr = H[real], Hi = H[imag];
        Y[real] = (Xr * Hr) - (Xi * Hi);
        Y[imag] = (Xr * Hi) + (Xi * Hr);
    }

    fft(Y, nn, -1);

    for (k = 0; k <= nn; k++) {
        real = (k * 2) + 1;
        Y[real] /= static_cast<float>(nn);
    }
}
```

```
std::vector<float> result(outputSize);
for (size_t k = 0; k < outputSize; k++) {
    real = (k * 2) + 1;
    result[k] = Y[real];
}

return result;
}
```

v3.0 - Compiler-Level Optimization

The compiler-level optimization version compiles source code with the `-O3` compiler flag set.

v4.0 - Code Tuning Optimization - Vector Initialization

Using `reserve()` to allocate memory for vectors enhances runtime performance and efficiency. The code snippets below illustrate vector initialization before and after implementing this optimization.

Vector initialization prior to optimization:

```
std::vector<float> X(nn * 2 + 1, 0.0f);
std::vector<float> H(nn * 2 + 1, 0.0f);
std::vector<float> Y(nn * 2 + 1, 0.0f);
```

Vector initialization after optimization:

```
std::vector<float> X(0.0f);
X.reserve(nn * 2 + 1);
std::vector<float> H(0.0f);
H.reserve(nn * 2 + 1);
std::vector<float> Y(0.0f);
Y.reserve(nn * 2 + 1);
```

v4.1 - Code Tuning Optimization - Multithreaded Vector Initialization

Vector initialization was optimized by utilizing multithreading during the initialization of vectors X and H.

```
const size_t numThreads = std::thread::hardware_concurrency();
size_t chunkSizeX = x.size() / numThreads;
size_t chunkSizeH = h.size() / numThreads;

std::vector<std::thread> threadsX(numThreads);
for (size_t i = 0; i < numThreads; i++) {
    threadsX[i] = std::thread(parallelLoop, std::ref(X), std::cref(x),
                             i * chunkSizeX, (i + 1) * chunkSizeX);
}
```

```

for (auto &thread : threadsX) {
    thread.join();
}

std::vector<std::thread> threadsH(numThreads);
for (size_t i = 0; i < numThreads; i++) {
    threadsH[i] = std::thread(parallelLoop, std::ref(H), std::cref(h),
                             i * chunkSizeH, (i + 1) * chunkSizeH);
}

for (auto &thread : threadsH) {
    thread.join();
}

```

v4.2 - Code Tuning Optimization - Precomputing Division for Scaling

The code's runtime performance is improved by precomputing the division for scaling and substituting division with multiplication. The following code snippets demonstrate the scaling before and after these optimizations.

Scaling prior to optimization:

```

for (k = 0; k <= nn; k++) {
    real = (k * 2) + 1;
    Y[real] /= static_cast<float>(nn);
}

```

Scaling after optimization:

```

float inv_nn = 1.0f / static_cast<float>(nn);
for (k = 0; k <= nn; k++) {
    real = (k * 2) + 1;
    Y[real] *= inv_nn;
}

```

v4.3 - Code Tuning Optimization - Computing Next Power of 2

Using bitwise operations rather than `cmath`'s provided `pow()` and `log()` functions improves runtime efficiency when computing the next power of 2 from $2 * N$, where N is the size of the vector x .

```

size_t nn = x.size() * 2;
nn--;
nn |= nn >> 1;
nn |= nn >> 2;
nn |= nn >> 4;

```

```
nn |= nn >> 8;
nn |= nn >> 16;
nn |= nn >> 32;
nn++;
```

v4.4 - Code Tuning Optimization - Minimizing Work Performed Inside Loops

Shifting constant computations outside of loops enhances runtime efficiency. In the following code snippet, loop optimization not only minimizes operations inside the loop but also substitutes division with multiplication, leading to further runtime improvements.

```
float scalingFactor = 1.0f / 32768.0f;
for (size_t i = 0; i < numSamples; i++) {
    audioData[i] = static_cast<float>(audio[i]) * scalingFactor;
}
```

Timing

All programs were timed using `time ./build/src/convolve ./build/src/guitar.wav ./build/src/big_hall_mono.wav ./tests/output<version>.wav`.

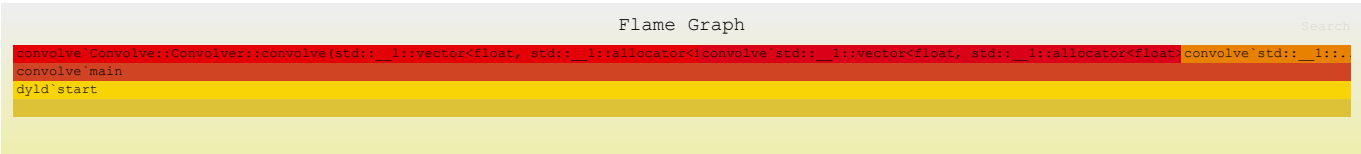
Version	Time
v1.0	830.50s user 0.22s system 99% cpu 13:53.90 total
v2.0	3.56s user 0.02s system 99% cpu 3.592 total
v3.0	1.43s user 0.03s system 83% cpu 1.735 total
v4.0	1.42s user 0.02s system 99% cpu 1.452 total
v4.1	1.43s user 0.03s system 100% cpu 1.448 total
v4.2	1.43s user 0.04s system 81% cpu 1.787 total
v4.3	1.43s user 0.03s system 100% cpu 1.460 total
v4.4	1.42s user 0.03s system 83% cpu 1.742 total

Profiling

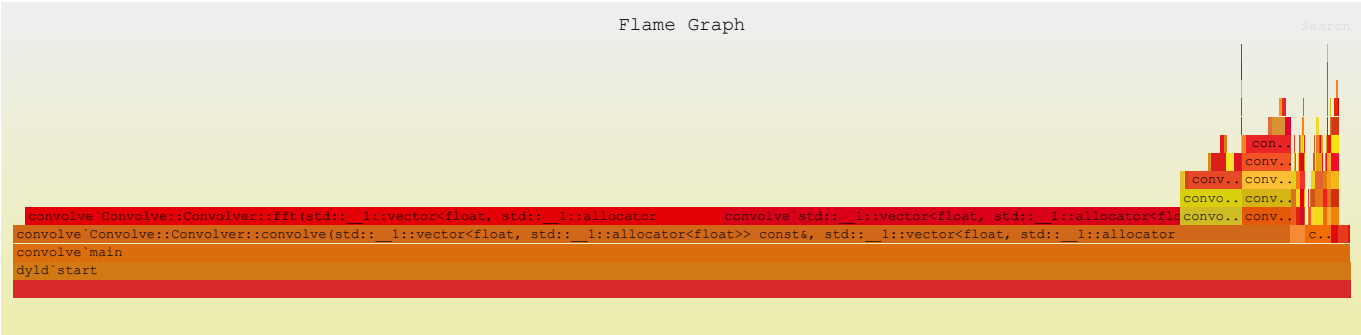
Flamegraphs were generated for all programs using the command `flamegraph --output ./flamegraphs/flamegraph<version>.svg --root -- ./build/src/convolve ./build/src/guitar.wav ./build/src/big_hall_mono.wav ./tests/output<version>.wav`.

Flamegraphs are a visualization tool for profiling, allowing us to understand the most resource-intensive parts of a program. Each box in the graph represents a function, with the width indicating the amount of time spent in that function. The y-axis shows the stack depth, with the topmost box being the currently executing function.

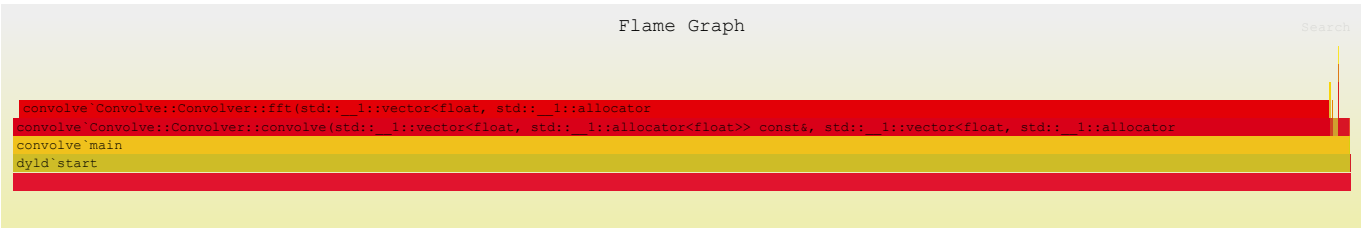
v1.0



v2.0

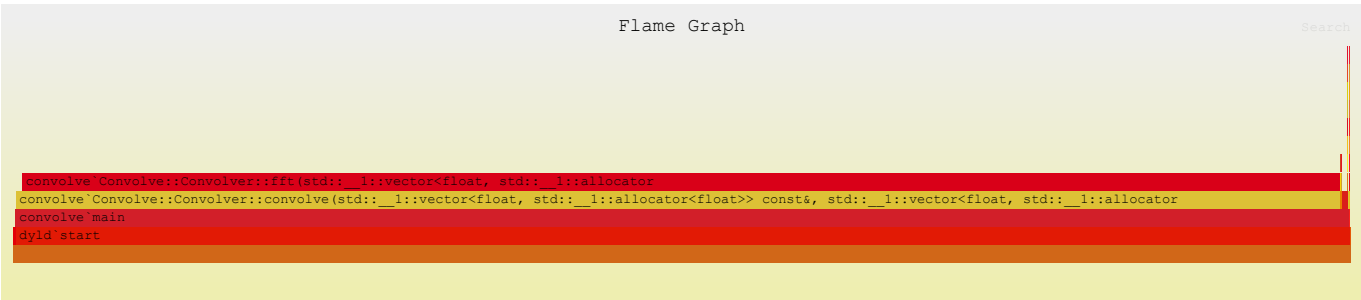


v3.0

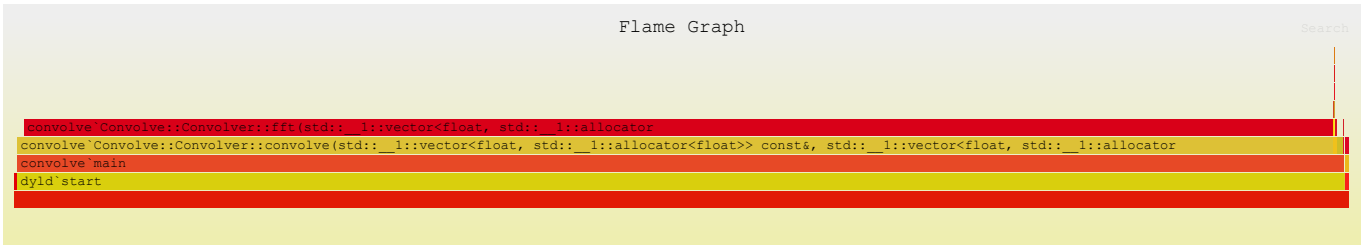


v4.0

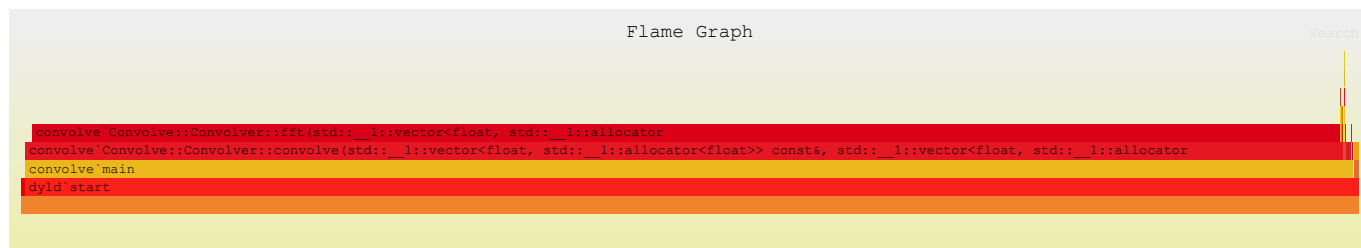
Despite applying code tuning techniques in versions v4.0-v4.4, significant improvements may not be apparent in the flamegraphs. This is because the program is already highly optimized due to algorithmic efficiencies and compiler-level optimizations. However, in theory, these code tunings should enhance the program's performance.



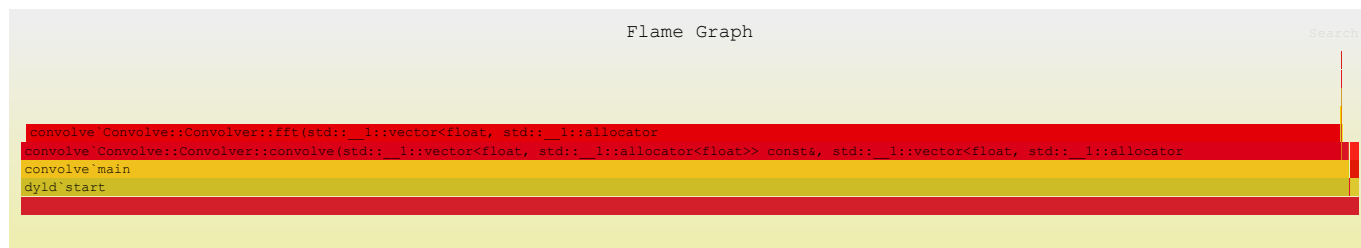
v4.1



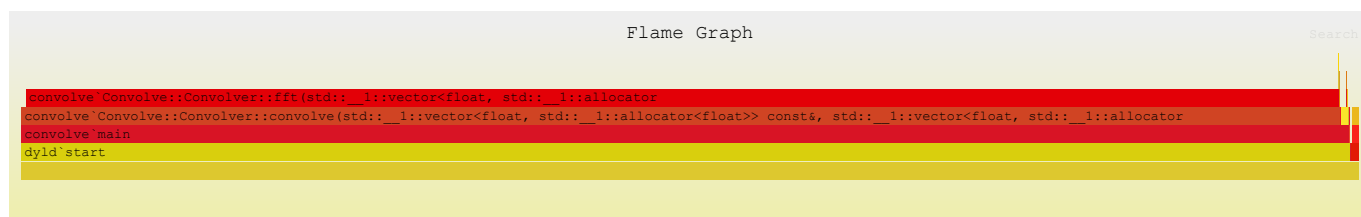
v4.2



v4.3



v4.4



Regression Testing

Regression testing is done by comparing **output.wav** files across versions to verify consistent results from **convolve()**. Execute **python3 ./tests/audiodiff.py** to compare frames from all **convolve()** versions. The output from **audiodiff.py** is shown below.

```
> python3 ./tests/audiodiff.py
```

```
./tests/outputv1.0.wav
```

```
Number of channels 1
```

```
Sample width 2
```

```
Frame rate 44100
```

```
Number of frames 817508530
```

```
./tests/outputv2.0.wav
```

```
Number of channels 1
```

```
Sample width 2
```

```
Frame rate 44100
```

```
Number of frames 817508530
```

```
./tests/outputv3.0.wav
```

```
Number of channels 1
Sample width 2
Frame rate 44100
Number of frames 817508530
```


```
./tests/outputv4.0.wav
Number of channels 1
Sample width 2
Frame rate 44100
Number of frames 817508530
```

```
./tests/outputv4.1.wav
Number of channels 1
Sample width 2
Frame rate 44100
Number of frames 817508530
```

```
./tests/outputv4.2.wav
Number of channels 1
Sample width 2
Frame rate 44100
Number of frames 817508530
```

```
./tests/outputv4.3.wav
Number of channels 1
Sample width 2
Frame rate 44100
Number of frames 817508530
```

```
./tests/outputv4.4.wav
Number of channels 1
Sample width 2
Frame rate 44100
Number of frames 817508530
```

```
Output wav files are equal
-----
All tests passed 
```