

Flake Encoder



Performance Tuning Project
Technion Softlab

Submitted by:

Ami Schwartzman

Supervisor:

Liat Atsmon

Table of Contents

1	Introduction.....	4
1.1	Project Goals.....	4
1.2	Introduction to Flake.....	4
2	Original version.....	5
2.1	Flow.....	5
2.2	Performance.....	5
3	Tools.....	6
3.1	Hardware.....	6
3.2	Software.....	6
3.3	Threading library.....	6
4	Benchmark.....	7
4.1	Running Flake.....	7
4.2	Number of threads selection.....	7
5	Thread optimizations, first version.....	8
5.1	New program flow.....	8
5.2	Results.....	9
5.2.1	Processing time.....	9
5.2.2	IO time.....	10
5.2.3	Total time.....	10
6	Thread optimizations, second version.....	11
6.1	Flow.....	11
6.2	Results.....	12
6.2.1	Processing time.....	12
6.2.2	Static versus dynamic scheduling.....	12
6.2.3	IO Time.....	13
6.2.4	Total time.....	13
7	Third version, architectural optimizations.....	14
7.1	Memory alignment.....	14
7.2	VTune.....	14
7.2.1	compute_autocorr.....	14
7.2.2	encode_residual_ipc.....	14
7.3	Results.....	18
7.3.1	Function speedup.....	18
7.3.2	Overall.....	18
8	Move to Intel Compiler.....	20
8.1	Results.....	20
9	Overall improvement.....	22
10	Sources used.....	24

Table of Figures

Figure 1: First version program flow	8
Figure 2: First version results	9
Figure 3: Second version program flow.....	11
Figure 4: Second version results	12
Figure 5: Third version results	19
Figure 6: Fourth version results	21
Figure 7: Final version results.....	22

1 Introduction

1.1 Project Goals

The goal of the Performance Tuning project done in Softlab is as follows:

- 1) Choose an open source application.
- 2) Optimize its performance using Intel based tools and by using architecture specific features found on Intel CPUs.
- 3) Return the software back to the open source community.

In this project I've chosen a software called Flake. Flake was an ideal candidate for this project since it contains no optimizations what so ever, and it's an audio encoding program, meaning plenty of continues CPU and memory activity.

1.2 Introduction to Flake

Flake is an open source Audio encoding program. The standard it encodes in is called FLAC – Free Lossless Audio Codec. The makers of FLAC have released a library which encodes into their format, and Flake was made as an alternative.

Flake is able to reach the same compression rate as the FLAC encoder, and doing so in less time¹.

As mentioned before, flake was an ideal candidate for this project since it's open source and without optimizations. Another reason for it be ideal is that it comes with a frontend tool which is able to read in wave files, call Flake's libraries in the proper manner and write the result back to the disk. This prevented me from looking for another utility which will be able to work with Flake's libraries.

¹ Based on <http://flake-enc.sourceforge.net/benchmarks.html>

2 Original version

The version I've based all my work on is 0.11. At the time of writing these lines, there wasn't a newer version available.

Flake has many parameters that affects it's encoding. The only must parameter is an input wave file to be encoded. Without any other parameters it encodes with its default settings. In the course of working with the application, I've ran flake with two parameter options:

- 1) Default parameters.
- 2) "-12" flag. This flag sets flake to its most rigorous running parameters. I refer to this setting as "hard" during the course of this book.

2.1 Flow

The flow of the original version is as follows:

- 1) Read a frame from the disk (usually 4 Kbytes).
- 2) Encode it.
- 3) Write the result to the result file (.FLAC file).

Since this flow is serial, there's no way to know where future frames will be written to, so it's impossible to parallelize this version unless the flow is changed.

2.2 Performance

Benchmarking will be discussed in a later chapter, but until then I will say that the original program ran at speeds of 18 seconds / 198 seconds with an IO time of 0.2 seconds.

3 Tools

3.1 Hardware

The entire project was done on a Linux machine. The HW was a Core i7 2.66GHz CPU with 4GBs of memory.

Since this processor has 4 real cores, whenever I've tested the program with less than 8 threads I've disabled HyperThreading in the BIOS to prevent a situation in which two threads would arrive to the same core.

3.2 Software

The compilers used were GCC (GNU C compiler) and ICC (Intel C Compiler). I've used GDB as the debugger and Valgrind as a memory leak checker.

Intel profiling tools were VTune and PTU (Performance Tuning Utility). Intel Thread Checker was used as well.

3.3 Threading library

The chosen threading library was OpenMP. The reasons for choosing it were its ease of use and portability. Also it's very easily integrated into GCC and ICC.

OpenMP is used by using "*#pragma*"s and its own API inside C code. In this project I've used it to:

1. Get wall time. This was used to benchmark the application, to get how much time each part of the code takes.
2. Get the number of available threads and set this number accordingly.
3. Break a "for" statement into multiple threads and control the way it distributes the work load.

4 Benchmark

The benchmark used in this program is time based. I've separated the benchmark into:

- 1) IO time which is the time it takes to read or write data to and from the disk, and
- 2) Raw processing time which is the time used for encoding.

Since there are many caching schemes available to be used such as processor caching, disk caching and etc, every measurement was averaged over 10 iterations, without taking into account the first iteration.

Input files were 300-500 MB wave files.

As previously mentioned, each version was benchmarked in two manners:

- 1) "Default" which is running flake with its default parameters, and
- 2) "Hard" which is running flake with its most rigorous settings. This setting causes flake to search for a match as much as possible.

4.1 Running Flake

If the current working directory is the Flake build directory, the command line for running flake with its default parameters is:

```
flake/flake ~/waves/beethoven.wav
```

The command line for running with the rigorous setting is:

```
flake/flake ~/waves/beethoven.wav -12
```

(this assumes that "~/waves/beethoven.wav" is the location of the input file. The output file will be named and placed in "~/waves/beethoven.flac")

4.2 Number of threads selection

I've inserted a custom parameter which allows the user to select the number of threads Flake will use.

For example, in order to force Flake to use 2 threads, the command line should be:

```
flake/flake ~/waves/beethoven.wav -f 2
```

If this parameter is not given, then it would default to the number of logical cores available. For example, the default for Core i7 processor is 8.

5 Thread optimizations, first version

Since the original version didn't include any threads use, the first thing I did to optimize the performance of this application was to parallelize it.

After studying the original program, I've decided on a way to parallelize the program. It seems that it would be best to try and split the encoding of different frames among different threads since there is no dependency among them.

5.1 New program flow

In order to split the encoding of frames among threads, a new flow for the program had to be used. Since the old flow was serial in regards to the encoding of frames, there is no way to know in which location a future frame (that another thread is working on) should be written to the disk. For this reason a new flow was proposed:

1. Read the entire input file (.WAV) into memory.
2. Divide the file into frames.
3. Divide the frames into different threads and encode them.
4. Keep the results in memory.
5. After all encoding is finished write the results back to the disk.

The following figure illustrates this:

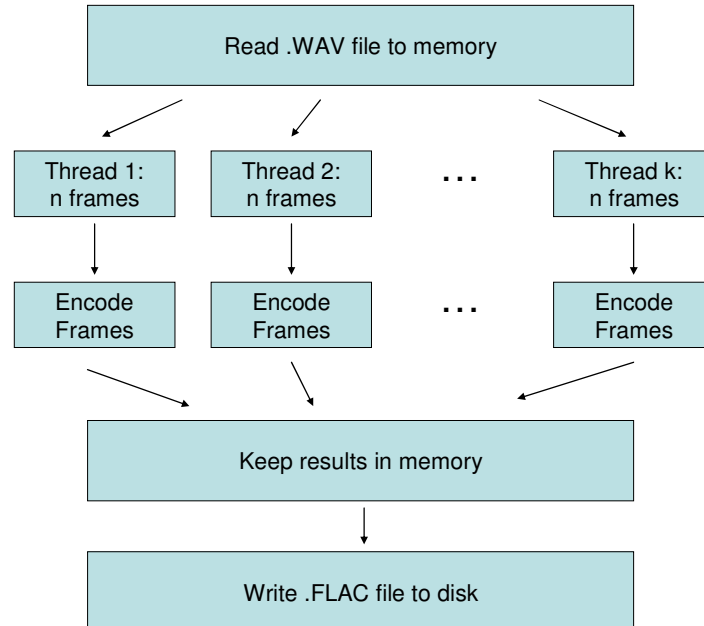


Figure 1: First version program flow

In order to accomplish this, several functions had to change, but most importantly, the main encoding function **encode_file**.

One main problem with this version is its high memory requirements since it's required to hold in memory the input file (in this project the biggest file size was 540 MBs). In case this was run a system with low available memory, the OS would resort to use virtual memory which would mean keeping data on the disk instead of memory which in turn would worsen processing time, or even worse, just fail to allocate the memory.

5.2 Results

5.2.1 Processing time

The processing time in this version had improved dramatically – reduced almost by the number of threads. The results are:

- 1 thread: 18 seconds / 200 seconds
- 2 threads: 9.5 seconds / 103 seconds
- 4 threads: 5.2 seconds / 53.4 seconds
- 8 threads: 4.6 seconds / 48 seconds

The following chart summarizes the results:

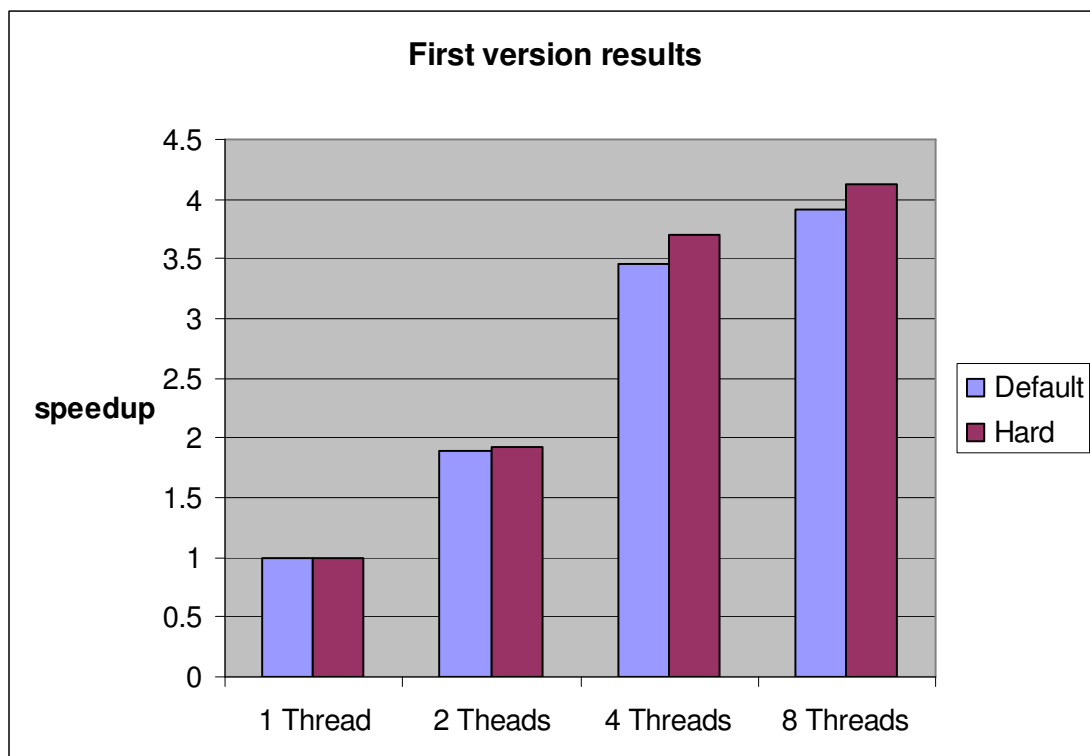


Figure 2: First version results

5.2.2 IO time

IO time in this version worsened in comparison to the original version – 5.5 seconds. This is hard to explain, but can be related to the fact that for a large file (~500 MB) the different disk caching algorithms that are present in HW and OS aren't as effective as constantly reading and then writing small consecutive blocks.

5.2.3 Total time

Since we had an increase in IO time and in 1 thread there's no noticeable change in comparison to the original version, total time for 1 thread increased. This is to be expected since the only optimization introduced in this version is threading, and only 1 thread is used and the disk access algorithm is found to be less effective. However, whenever there is more than 1 thread working, overall time is greatly reduced.

6 Thread optimizations, second version

One of the drawbacks of the first version is its high memory demands. After consulting with Koby, he suggested using smaller chunks to work with, instead reading in the entire file.

Another reason for choosing chunk size of 2MB is to better utilize the 3rd level cache which is available to all cores.

6.1 Flow

In this version a new flow is used:

1. Read a chunk of the input file (.WAV) into memory (typically 2MBs).
2. Divide the chunk into frames.
3. Divide the frames into different threads and encode them.
4. Keep chunk results in memory.
5. Write the results back to the disk.
6. If there is more to read from the input file, go back to 1.

The following figure helps to illustrate this:

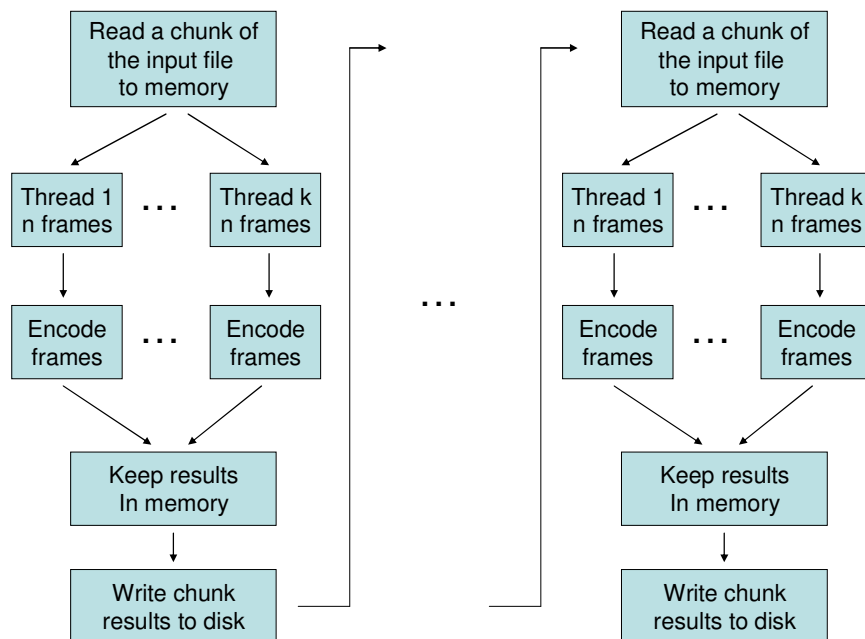


Figure 3: Second version program flow

The chunk size is configured by a define declared in **flake.c**.

6.2 Results

6.2.1 Processing time

Processing time for 1 and 2 threads didn't change and it could be explained by the fact that the better utilization of the 3rd level cache doesn't affect this small number of threads.

Surprisingly, the processing time for 4 and 8 threads increased but could be explained by the fact that it's harder to synchronize when increasing the number of threads. The new times are:

- 4 threads: 6.8 seconds / 55 seconds
- 8 threads: 5.5 seconds / 50.4 seconds

These results can be better illustrated as follows:

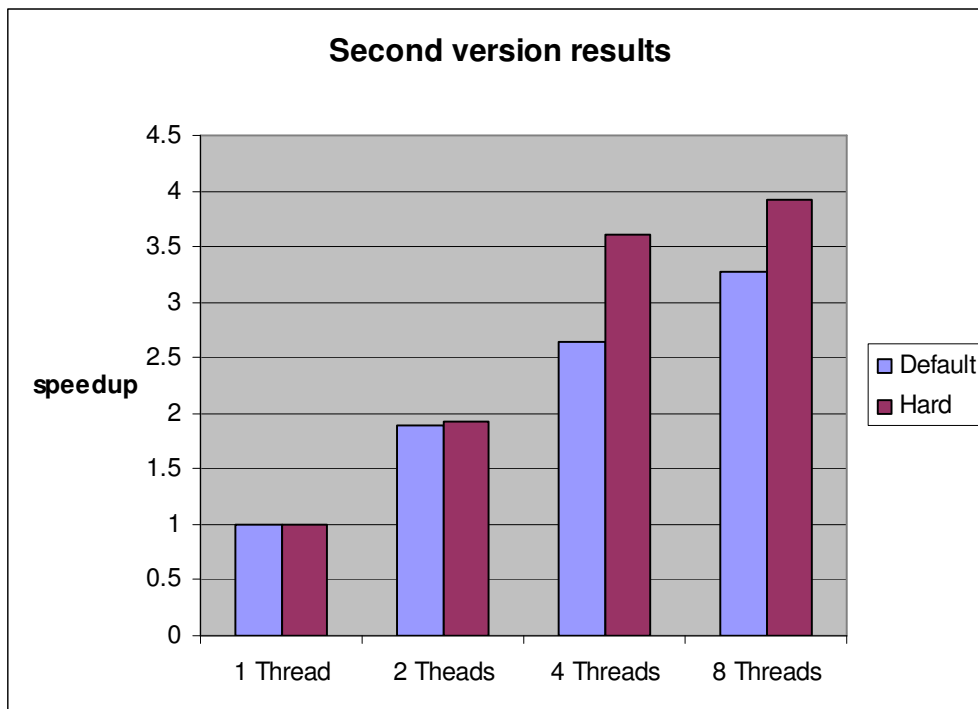


Figure 4: Second version results

6.2.2 Static versus dynamic scheduling

While dividing the frames among threads two possible approaches were tested: static and dynamic scheduling which refers to the way OpenMP allows for the division of for loop iterations among threads. Static scheduling means initially dividing the loops among threads. For example, if 4 threads are used and there are 1000 iterations then thread 1 will receive iterations 0-249, thread 2 will receive 250-499 and so on. Dynamic scheduling means whenever a thread finishes it receives the next iteration regardless of order.

Theoretically dynamic scheduling should give better performance since a situation could occur in which the first 250 iterations finish faster than the next 250. In this scenario the first thread would finish before the second one and will remain idle. In dynamic scheduling this would never happen. In real life testing, I've found no measurable difference between the two with a slight advantage to dynamic scheduling.

6.2.3 IO Time

A move to this version reduced IO time dramatically, almost to the time it was on the original version – 0.7 seconds. This is to be expected since the sizes of file reads and writes resemble more the original version than the first version and so the efficiency of the disk caching algorithms also look more similar to the first version.

6.2.4 Total time

Even though there's no improvement for 1 and 2 threads and actually worse results with 4 and 8 threads, since there's a big improvement with IO time, total time was reduced with all threads. At the end there was an improvement in between 10% to 40% over the previous version.

7 Third version, architectural optimizations

Since it seemed that there could be no performance to be gained by tweaking threads use, I've turned to using architectural advantages to increase performance.

In this version the program flow didn't change.

7.1 Memory alignment

In every possible location in the code, I've substituted conventional mallocs with aligned mallocs and the alignment was to a 64 Kbyte boundary since a cache line is 64 Kbytes long. The reason for this was this: if the compiler knows that there are memory arrays which are aligned it might choose to use instructions that demand aligned memory addresses. These instructions usually give smaller latency than instructions that use unaligned memory access.

Another alignment issue was with multi-dimensional arrays. Data structures inside Flake are complex and inside some of them there are multi-dimensional arrays on which Flake is doing most of its computations. I used directives which forced GCC and ICC to align these arrays. In some cases there was a need to rearrange the data structures and extend the length of some to be compatible with a 64 Kbyte boundary.

7.2 VTune

I have used VTune to find in what functions in a typical run Flake spends most of its time.

7.2.1 compute_autocorr

One of the functions found by VTune was a function to compute the autocorrelation of segments. To optimize this function I've rewritten the function to use inline assembly & SSE instructions. The results showed the compiler's own optimized version gave better run time than the version that I've written. The reason for this has several explanations. First of all, using inline assembly inside C code forces the compiler to use this assembly code and thus prevents from optimizing. Also, do to this I'm restricting the use of specific registers since I have to declare which registers I'm working with. Another reason is that the compiler probably understood what this function was trying to do and was able to better optimize it. Since I've ran the compilers with a flag to do inter-procedural optimizations, I've greatly restricted its efforts.

7.2.2 encode_residual_ipc

Another function found to be a hot spot by VTune proved to be more promising for manual optimizations. Below is the function's code:

```

static void encode_residual_lpc(int32_t res[], int32_t smp[], int n, int order,
                               int32_t coefs[], int shift)
{
    int i;
    int32_t pred;

    for(i=0; i<order; i++) {
        res[i] = smp[i];
    }
    for(i=order; i<n; i++) {
        pred = 0;
        // note that all cases fall through.
        // the result is in an unrolled loop for each order
        switch(order) {
            case 32: pred += coefs[31] * smp[i-32];
            case 31: pred += coefs[30] * smp[i-31];
            case 30: pred += coefs[29] * smp[i-30];
            case 29: pred += coefs[28] * smp[i-29];
            case 28: pred += coefs[27] * smp[i-28];
            case 27: pred += coefs[26] * smp[i-27];
            case 26: pred += coefs[25] * smp[i-26];
            case 25: pred += coefs[24] * smp[i-25];
            case 24: pred += coefs[23] * smp[i-24];
            case 23: pred += coefs[22] * smp[i-23];
            case 22: pred += coefs[21] * smp[i-22];
            case 21: pred += coefs[20] * smp[i-21];
            case 20: pred += coefs[19] * smp[i-20];
            case 19: pred += coefs[18] * smp[i-19];
            case 18: pred += coefs[17] * smp[i-18];
            case 17: pred += coefs[16] * smp[i-17];
            case 16: pred += coefs[15] * smp[i-16];
            case 15: pred += coefs[14] * smp[i-15];
            case 14: pred += coefs[13] * smp[i-14];
            case 13: pred += coefs[12] * smp[i-13];
            case 12: pred += coefs[11] * smp[i-12];
            case 11: pred += coefs[10] * smp[i-11];
            case 10: pred += coefs[ 9] * smp[i-10];
            case  9: pred += coefs[ 8] * smp[i- 9];
            case  8: pred += coefs[ 7] * smp[i- 8];
            case  7: pred += coefs[ 6] * smp[i- 7];
            case  6: pred += coefs[ 5] * smp[i- 6];
            case  5: pred += coefs[ 4] * smp[i- 5];
            case  4: pred += coefs[ 3] * smp[i- 4];
            case  3: pred += coefs[ 2] * smp[i- 3];
            case  2: pred += coefs[ 1] * smp[i- 2];
            case  1: pred += coefs[ 0] * smp[i- 1];

```

```

        case 0:
        default: break;
    }
    res[i] = smp[i] - (pred >> shift);
}
}

```

This time I didn't use inline assembly for the reasons mentioned before, and so I've used SSE instructions intrinsics instead. These intrinsics are specified in Intel's instruction set reference and basically give a C function for every instruction. This time C syntax isn't broken by assembly code and so the compiler is free to optimize as much as it can. Another reason to use instruction intrinsics here instead of inline assembly is GCC's compatibility to ICC. Since these functions are basically declared for use with ICC, GCC made an effort to be completely backward compatible to ICC by means of libraries and functions. This way no special work had to go into when converting the project to ICC.

Another thing I've done in this function to accelerate its operation is loop unrolling. In this function there are 2 nested loops. While unrolling each loop 4 times, I've unrolled the entire function by a factor of 16. The new function's code can be seen below:

```

static void encode_residual_lpc(__m128i res[], __m128i smp[], int n, int order,
                               __m128i coefs[], int shift)
{
    int i,j;
    int i2,j2;
    __m128i pred1, pred2, pred3, pred4;
    __m128i a, b, c, d, e;
    int32_t *t1, *t2;
    int32_t p1, p2, p3, p4;

    for(i=0; i<(order/4)+1; i++) {
        i2 = i*4;
        res[i] = smp[i];
    }

    for(i=(order/4); i<(n/4); i++) {
        pred1 = _mm_xor_si128(pred1, pred1);
        pred2 = _mm_xor_si128(pred2, pred2);
        pred3 = _mm_xor_si128(pred3, pred3);
        pred4 = _mm_xor_si128(pred4, pred4);
        p1 = p2 = p3 = p4 = 0;

        for (j=0; j<(order/4); j++) {
            i2 = i*4;
            j2 = j*4;
            a = _mm_load_si128(coefs+j);

```

```

b = _mm_load_si128(smp+i-j-1);
c = _mm_load_si128(smp+i-j);

d = _mm_shuffle_epi32(b,27);
e = _mm_mullo_epi32(a,d);
pred1 = _mm_add_epi32(e, pred1);

d = _mm_alignr_epi8(c,b,4*1);
d = _mm_shuffle_epi32(d,27);
e = _mm_mullo_epi32(a,d);
pred2 = _mm_add_epi32(e, pred2);

d = _mm_alignr_epi8(c,b,4*2);
d = _mm_shuffle_epi32(d,27);
e = _mm_mullo_epi32(a,d);
pred3 = _mm_add_epi32(e, pred3);

d = _mm_alignr_epi8(c,b,4*3);
d = _mm_shuffle_epi32(d,27);
e = _mm_mullo_epi32(a,d);
pred4 = _mm_add_epi32(e, pred4);
}

t1 = coefs;
t2 = smp;

if ((order & 0x3) != 0) {
    if (order<4) i2 = i*4;
    for (j2=(order&0xffffffc);j2<order; j2++) {
        p1 += (i2 -j2-1) < 0 ? 0 : t1[j2] * t2[i2 -j2-1];
        p2 += (i2+1-j2-1) < 0 ? 0 : t1[j2] * t2[i2+1-j2-1];
        p3 += (i2+2-j2-1) < 0 ? 0 : t1[j2] * t2[i2+2-j2-1];
        p4 += (i2+3-j2-1) < 0 ? 0 : t1[j2] * t2[i2+3-j2-1];
    }
}

pred1 = _mm_hadd_epi32(pred1,pred1);
pred1 = _mm_hadd_epi32(pred1,pred1);
p1 = (p1 + _mm_extract_epi32(pred1, 0)) >> shift;

pred2 = _mm_hadd_epi32(pred2,pred2);
pred2 = _mm_hadd_epi32(pred2,pred2);
p2 = (p2 + _mm_extract_epi32(pred2, 0)) >> shift;

pred3 = _mm_hadd_epi32(pred3,pred3);

```

```

pred3 = _mm_hadd_epi32(pred3,pred3);
p3 = (p3 + _mm_extract_epi32(pred3, 0)) >> shift;

pred4 = _mm_hadd_epi32(pred4,pred4);
pred4 = _mm_hadd_epi32(pred4,pred4);
p4 = (p4 + _mm_extract_epi32(pred4, 0)) >> shift;

t1 = res;

if (!(i==(order/4))&&((order&0x3)>=1))
    t1[i2] = t2[i2] - p1;
if (!(i==(order/4))&&((order&0x3)>=2))
    t1[i2+1] = t2[i2+1] - p2;
if (!(i==(order/4))&&((order&0x3)>=3))
    t1[i2+2] = t2[i2+2] - p3;

    t1[i2+3] = t2[i2+3] - p4;
}
}

```

7.3 Results

7.3.1 Function speedup

In order to estimate the speedup in the updated function alone, I've used VTune to see the value of CPU_CLK_UNHALTED before and after the improvement. The result was that before CPU_CLK_UNHALTED was 108 and after it is 101. In percentage before it was 14.96% and afterwards it is 14.17%.

7.3.2 Overall

IO performance didn't change in this version and we didn't expect it to change since nothing IO-wise was done. Processing time was improved as a result of all optimizations:

- 1 thread: 17.1 seconds / 181.5 seconds
- 2 threads: 9 seconds / 92.5 seconds
- 4 threads: 6.4 seconds / 48.5 seconds
- 8 threads: 5.2 seconds / 42.7 seconds

The following chart summarizes the results:

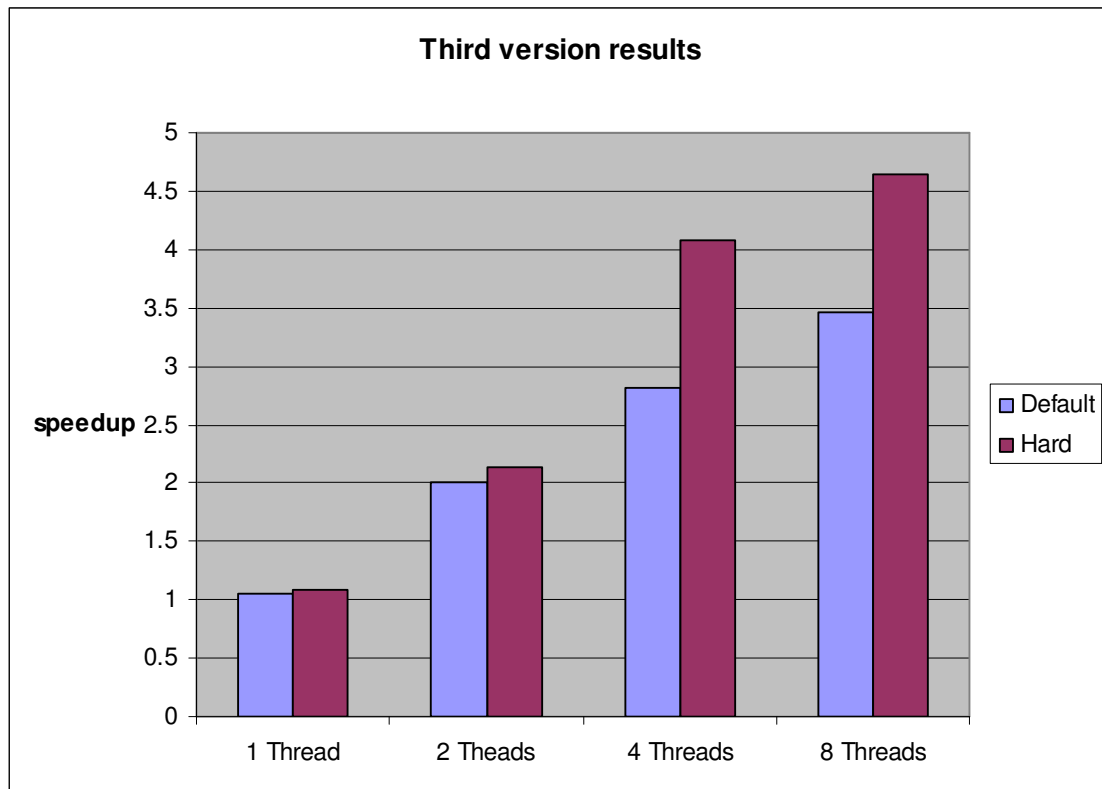


Figure 5: Third version results

This time it can be seen that every version including with and without threads and “hard” and default benchmarks favored with these optimizations. Looking at the results it can be seen that the “hard” benchmark results improved more than the default benchmark and the reason for this is because the optimized function is operated with a larger dataset in this benchmark. This means that the loop unrolling and SSE instructions operate more time than with a smaller dataset.

Overall there is an improvement between 5% to 13.5% over the previous version.

8 Move to Intel Compiler

The last thing I did in this project is to convert the project to use the Intel compiler. This change meant changing the directives that force the compiler/linker to allocate arrays on aligned addresses and to change the Makefiles.

Using the Intel compiler produced mixed results: while the default benchmark showed improvement, the hard benchmark took an extreme turn for the worse. Running the application with the hard benchmark took so long that there was no point in taking the results for performance measurement. This is probably attributed to a bug in the Intel compiler.

8.1 Results

IO time didn't change with a move to the Intel compiler, while the processing time did improve:

- 1 thread: 14.9 seconds
- 2 threads: 8.3 seconds
- 4 threads: 5.6 seconds
- 8 threads: 4.8 seconds

It should be noted that while the processing time for 8 threads improved over 4 threads, the IO time actually increased so overall there was no improvement moving to 8 threads. It's hard to explain the reason for this, but it might be related to the fact that the processing time went to as low as it possibly can and now it's not the bottleneck of the software, but rather IO operations.

The following illustrates the results:

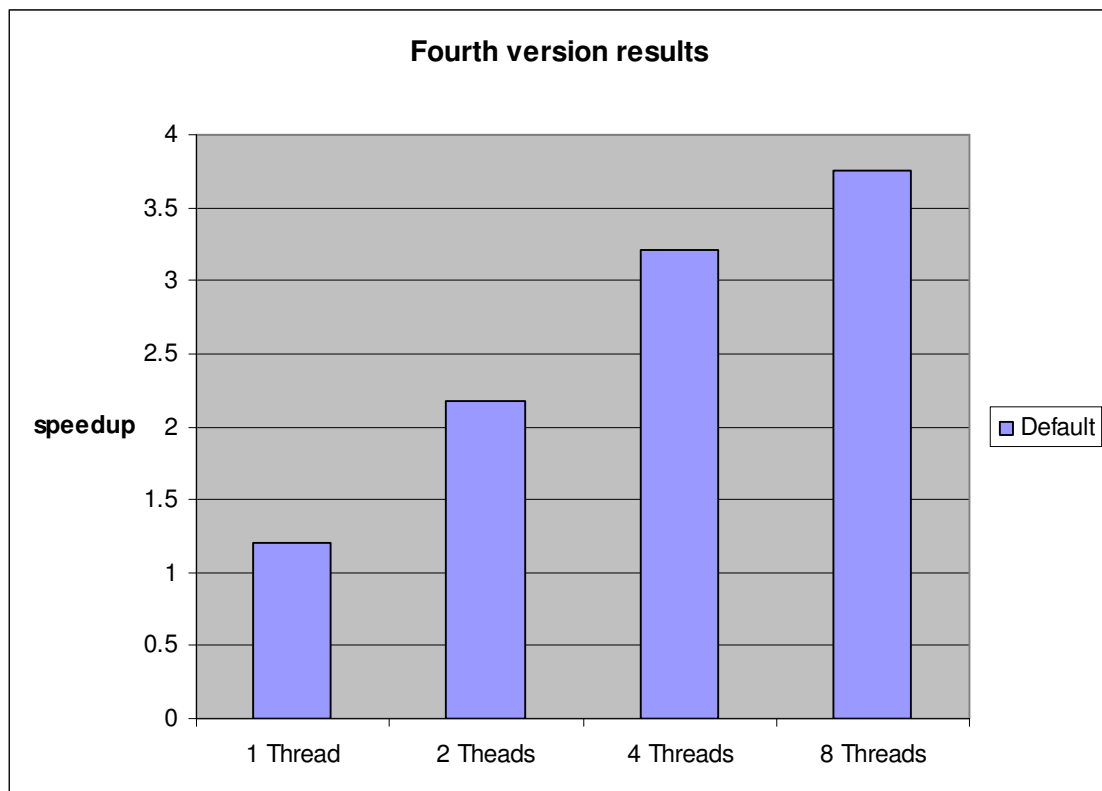


Figure 6: Fourth version results

Overall, when there was improvement it was between 8.5% to 14.7% over the previous version.

Note: even though 8 threads gave smaller processing time, IO time increased in this version so the total time increased.

9 Overall improvement

When combining everything together we can see that there was a big improvement:

Default benchmark (compiled with ICC):

- 1 thread: x1.2 speedup
- 2 threads: x2.17 speedup
- 4 threads: x3.21 speedup
- 8 threads: x2.85 speedup

Hard benchmark (compiled with GCC):

- 1 thread: x1.09 speedup
- 2 threads: x2.14 speedup
- 4 threads: x4.08 speedup
- 8 threads: x4.64 speedup

The following chart illustrates the speedups:

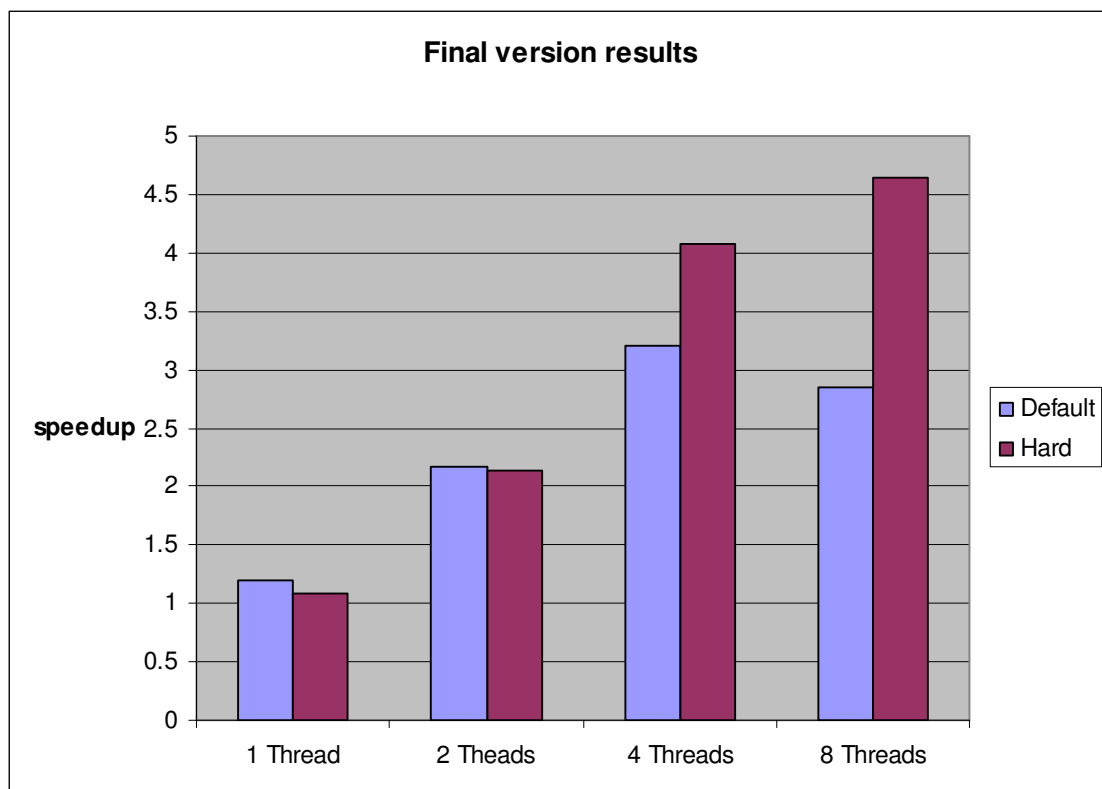


Figure 7: Final version results

Note: The resulting .FLAC files of the final version (and all minor versions as well) are 100% bitwise compatible to the original version.

10 Sources used

1. Flake home directory: <http://flake-enc.sourceforge.net/>
2. Inline assembly for x86 in Linux:
<http://www.ibm.com/developerworks/linux/library/l-ia.html>
3. GCC Inline Assembly HOWTO: <http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html>
4. X86 Built-in Functions (part of GCC manual): http://gcc.gnu.org/onlinedocs/gcc-3.4.6/gcc/X86-Built_002din-Functions.html
5. Quick Reference Guide to Optimizations with Intel Compilers Version 11:
http://cache-www.intel.com/cd/00/00/22/23/222300_222300.pdf&ei=0dngSYXeCsihjAf9ldDUDQ&usg=AFQjCNEEmHrLbO-8JyDakWzVquW1jZKEAA
6. Intel 64 and IA-32 Architectures Software Developer's Manuals:
Volume 2A & 2B: Instruction Set Reference:
<http://www.intel.com/products/processor/manuals/>
7. Intel SSE4 Programming Reference:
<http://softwarecommunity.intel.com/isn/Downloads/Intel%20SSE4%20Programming%20Reference.pdf>
8. Intel 64 and IA-32 Architectures Optimizations Reference Manual:
<http://www.intel.com/products/processor/manuals/>
9. Using Intel VTune Performance Analyzer Events/Ratios & Optimizing Applications
White Paper: <http://software.intel.com/en-us/articles/using-intel-vtune-performance-analyzer-events-ratios-optimizing-applications/>
10. Using Streaming SIMD Extensions 2 (SSE2) in a Double-precision 3D Transform:
<http://software.intel.com/en-us/articles/sse2-instructions-in-a-double-precision-3d-transform/>