LASER INTERFEROMETER GRAVITATIONAL WAVE OBSERVATORY

LIGO Laboratory / LIGO Scientific Collaboration

LIGO-T2100460-v3 Advanced LIGO 2/10/2022

A Biquad Implementation Using Advanced Vector Extensions

Daniel Sigg

Distribution of this document:
LIGO Scientific Collaboration

This is an internal working note
of the LIGO Laboratory.

California Institute of Technology
LIGO Project – MS 18-34
1200 E. California Blvd.
Pasadena, CA 91125
Phone (626) 395-2129
Fax (626) 304-9834
E-mail: info@ligo.caltech.edu

Massachusetts Institute of Technology
LIGO Project – NW22-295
185 Albany St
Cambridge, MA 02139
Phone (617) 253-4824
Fax (617) 253-7014
E-mail: info@ligo.mit.edu

LIGO Hanford Observatory
P.O. Box 159
Richland WA 99352
Phone 509-372-8106
Fax 509-372-8137

LIGO Livingston Observatory
P.O. Box 940
Livingston, LA 70754
Phone 225-686-3100
Fax 225-686-7189

http://www.ligo.caltech.edu/

1 Introduction

Intel based CPU have implemented advanced vector extensions since a while. We look at AVX2 and AVX-512[1], AVX2 has become common, whereas AVX-512F is newer and less widely available. Our newest set of front-end computers based on Cascade Lake processors, like the Intel® Xeon® W-2245 Processor, support both.

Advanced vector extensions allow to process multiple floating-point operations in parallel: 4 in case of AVX2 and 8 in case of AVX-512F. These operations also use a separate CPU register file. There are 16 registers for AVX2 and 32 for AVX-512.

A single IIR filter and its implementation as a cascaded set of biquad sections cannot easily be parallelized. Instead, we investigate how to implement multiple IIR filters in parallel consisting of a fixed set of 3 biquad sections. The main use case for this parallelization would be the decimation filters that reduce the data rate from a fast ADC running 2^{19} Hz.

2 Thermal Throttling

Vector extensions produce a lot of heat and potentially lead to thermal throttling of the CPU. Figure 1 shows the potential impact on CPU clock for a CPU that is from the same family as the W-2245 with the same core count and similar frequencies (3.9 vs 3.6 GHz). For reference see en.wikichip.org[2], and online articles[3,4].

For our front-end computers we disable turbo mode in the BIOS, so they never run faster than their normal base frequency of 3.9 GHz. From the table below, we can conclude that it is unlikely that thermal throttling will be an issue for AVX2 and maybe effect performance if AVX-512 is run on all cores. In our targeted use case, these operations would only run on the IOP and hence one core. So, **we don't envision any issues due to thermal throttling.**

Mode	Base	Turbo Frequency/Active Cores							
		1	2	3	4	5	6	7	8
Normal	3,600MHz	4,400MHz	4,400MHz	4,300MHz	4,300MHz	4,300MHz	4,300MHz	4,300MHz	4,300MHz
AVX2	3,000MHz	4,000MHz	4,000MHz	3,900MHz	3,900MHz	3,900MHz	3,900MHz	3,900MHz	3,900MHz
AVX512	2,600MHz	3,800MHz	3,800MHz	3,600MHz	3,600MHz	3,500MHz	3,500MHz	3,500MHz	3,500MHz

Figure 1: Thermal Throttling on a Xeon Gold 6244 CPU 3.6GHz with 4.4GHz turbo.

[1] https://en.wikipedia.org/wiki/Advanced_Vector_Extensions

[2] https://en.wikichip.org/wiki/intel/xeon_gold/6244

[3] https://extensa.tech/blog/avx-throttling-part1/

[4] https://lemire.me/blog/2018/09/07/avx-512-when-and-how-to-use-these-new-instructions/

2.1 Implementation

We use the LIGO biquad implementation as a starting point. See the zip file in T2100460 for the C/C++ code that was used. We first slightly rewrite it by using array indices instead of pointer arithmetic. Next, we then implement both AVX2 and AVX-512 versions. The filter coefficients don't need to be identical, but the number of sections needs to be the same. Finally, we write versions that use identical coefficients and exactly 3 sections, implement decimation and work on longer strides of data. The later versions are targeted an ADC data that is arranged in vectors of fixed length corresponding to the number of ADC channels and is stacked up 8 samples deep that need to be filtered and decimated.

A second set of tests implements both a down conversion followed by an IIR filter. We look at two cases: there are multiple inputs that are all down-converted by the same frequency and a single input that is down-converted by multiple frequencies. These routines don't stack up any samples but work clock by clock.

Table 1: Tested Biquad Implementations.

iir_filter_biquad	Copied from fm10Gen.c.
biquad_std	Same as above but using array indices instead of pointer arithmetic.
biquad2_sse3	SSE3 implementation (with FMA) of biquad, working on 2 input samples in parallel.
biquad4_avx2	AVX2 implementation of biquad, working on 4 input samples in parallel.
biquad8_avx2	AVX2 implementation of biquad, working on 8 input samples in parallel, this function uses the same parameters as biquadAVX512 and can be interchanged.
biquad8_avx512	AVX512 implementation of biquad, working on 8 input samples in parallel
biquad_stride1_std	Biquad implementation that works on a longer stride of data and includes a decimation. All filter coefficients are identical.
biquad_stride2_std	Poor man's way to parallelize the above biquad_stride1 working on 2 samples intermixed.
biquad_stride4_std	Working on 4 samples intermixed.
biquad_stride2_section3_sse3	SSE3 implementation (with FMA) of biquad_stride1, working on 2 samples in parallel.
biquad_stride8_section3_sse3	SSE3 implementation (with FMA) of biquad_stride1, working on 2 samples in parallel, but using poor man's way of parallelization, working on 4 vectors intermixed.

biquad _stride4_section3_avx2	AVX implementation of biquad_stride1, working on 4 samples in parallel.
biquad _stride8_section3_avx2	Poor man's way of parallelization, working on 2 vectors intermixed.
biquad_stride16_section3_avx2	Working on 4 vectors intermixed.
biquad_stride32_section3_avx2	Working on 8 vectors intermixed.
biquad _stride8_section3_avx512	AVX-512 implementation of biquad_stride1, working on 8 samples in parallel.
biquad_stride16_section3_avx512	Poor man's way of parallelization, working on 2 vectors intermixed.
biquad_stride32_section3_avx512	Working on 4 vectors intermixed.
demod_dec_stride8_section3_std	Demodulate multiple channels by a single frequency followed by a decimation filter.
demod_dec_stride8_section3_sse3	SSE3 implementation (with FMA) working on 4x2 samples in parallel.
demod_dec_stride8_section3_avx2	AVX2 implementation working on 2x4 samples in parallel.
demod_dec_stride8_section3_avx512	AVX512 implementation working on 8 samples in parallel.
demod_dec_rotation8_section3_std	Demodulate a single channel by multiple frequencies followed by a decimation filter.
demod_dec_rotation8_section3_sse3	SSE3 implementation (with FMA) working on 4x2 samples in parallel.
demod_dec_rotation8_section3_avx2	AVX2 implementation working on 2x4 samples in parallel.
demod_dec_rotation8_section3_avx512	AVX512 implementation working on 8 samples in parallel.

3 Results

We run two version on the Xeon W-2245: once compiled using the RGC compiler flags, "-O -ffast-math -m80387 -msse2 -fno-builtin-sincos -march=native", and once using a higher optimization level, "-O5 -march=native". The native architecture flag has been added to both, otherwise AVX2 and AVX-512 wouldn't be available. In our case native means cascadelake.

Table 2: Test Results Xeon W-2245

Function	RCG flags		Performance			
	Time(s)	Speedup	Time(s)		Speedup	
iir_filter_biquad	11.6	1.0	12.1		1.0	
biquad_std	11.9	1.0	13.7		0.8	
biquad2_sse3	5.28	2.2	5.10		2.3	
biquad4_avx2	2.78	4.2	2.78		4.2	
biquad8_avx2	2.56	4.5	2.58		4.5	
biquad8_avx512	1.76	6.6	1.77		6.6	
biquad_stride1_std	11.6	1.0	10.2		1.1	
biquad_stride2_std	11.0	1.1	8.8		1.3	
biquad_stride4_std	9.7	1.2	8.2	7.7	1.4	1.5
biquad_stride2_section3_sse3	5.3	2.2	4.4	4.4	2.6	2.6
biquad_stride8_section3_sse3	3.58	3.2	2.83	3.4	4.1	3.4
biquad _stride4_section3_avx2	2.98	3.9	2.42	2.4	4.8	4.8
biquad _stride8_section3_avx2	2.45	4.7	1.99	2.1	5.8	5.5
biquad_stride16_section3_avx2	1.87	6.2	1.48	1.79	7.8	6.5
biquad_stride32_section3_avx2	1.77	6.6	1.65	2.81	7.0	4.1
biquad _stride8_section3_avx512	1.96	5.9	1.48		7.8	
biquad_stride16_section3_avx512	1.37	8.5	1.06		10.9	
biquad_stride32_section3_avx512	1.21	9.6	1.02		11.4	
demod_dec_stride8_section3_std	25.6	1.0	19.8		1.3	
demod_dec_stride8_section3_sse3	8.67	3.0	7.54	10.0	3.4	2.6
demod_dec_stride8_section3_avx2	5.11	5.0	4.31	5.53	5.9	4.6
demod_dec_stride8_section3_avx512	4.16	6.2	3.32		7.7	
demod_dec_rotation8_section3_std	27.7	1.0	22.7		1.2	
demod_dec_rotation8_section3_sse3	10.7	2.6	9.31	11.4	3.0	2.4
demod_dec_rotation8_section3_avx2	5.77	4.8	5.11	6.59	5.4	4.2
demod_dec_rotation8_section3_avx512	4.15	6.7	3.32		8.3	

There is an important difference in the SSE3 and AVX2 implementations, if a processor with AVX2 but without AVX-512 capability, such as haswell, is selected. For an AVX-512 capable processor, the compiled code uses 32 internal xmm or ymm registers, whereas only 16 registers are available in the case of haswell. This is because for AVX512 the register set was increased by a factor of 2. Where it matters both times are given, haswell times were generally slower. The biquad functions were repeated many times to get stable execution times.

4 Conclusions

SSE3, AVX2 and AVX-512 can provide a good speedup over the current implementation of the internal decimation filters.

We would profit from being able to compile the front-end code with a higher optimization level.

SSE3, AVX2 and AVX-512 implementations are less dependent on the selected optimization level.

At the high optimization level the speedup by SSE3 is between 2 and 4, whereas for AVX2 it is between 3 and 5, and for AVX-512 it is between 5 and 8. At the RGC optimization level, the SSE3 speedup is between 2 and 3, whereas the AVX2 speedup is between 2.5 and 8, and the AVX-512 speedup is between 7 and 11. The real world speedup of in IOP is of course less, since it has to perform other operations, but with both a fast and a low noise ADC running at 2^{19} Hz the burden of the decimation filters is significant.