Accelerating software radios by means of SIMD Instructions. A case for the AVX2 and AVX512 Extensions

Damian Miralles DAMIAN.MIRALLES@AERO.ORG
The Aerospace Corporation, 2310 E. El Segundo Blvd. El Segundo, CA 90245-4609 USA

Jessica Iwamoto JESSICA.IWAMOTO@AERO.ORG
The Aerospace Corporation, 2310 E. El Segundo Blvd. El Segundo, CA 90245-4609 USA

Abstract

Current computer architecture trends are moving towards parallelization by means of node replication and data parallelization, which optimize the execution speed of a given application. Increasing the number of nodes is constrained by the hardware platform in use; however, effective data parallelization techniques can improve processing speeds by leveraging existing resources of the platform. This paper presents the AVX2 and AVX512 instruction addition to several kernels in the VOLK library. We discuss the capabilities of the new extensions and their interaction with the VOLK library. Finally, we show profiling results of the speed enhancements added to the library for AVX capable machines.

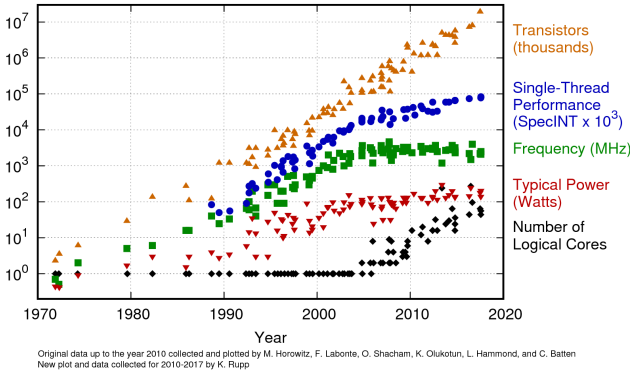

Figure 1. CPU performance over time (Karl Rupp, 2018)

1. CPU Performance and Evolution

The trend where Central Processing Unit (CPU) frequencies were increasing significantly over time has been halted due to concerns of power consumption and an increase in device operating temperatures. Even though the CPU frequencies between modern processors are not drastically different, new computers seem to perform better each year. To achieve this, modern CPUs rely heavily on parallelism. As the plateau in CPU frequency becomes more evident, the number of logical cores in devices will increase. This trend is illustrated in Figure 1, which builds upon previous book keeping records and extend it to reflect the changes up to present. The result is a detailed description of the trend of CPU frequencies, power consumption, and number of logical cores, among others (Karl Rupp, 2018).

Looking at computer architecture, instructions are performed in terms of CPU clock cycles, with every new tick being a chance to perform another operation. Keeping with that analogy, it is safe to assume that in certain applications, the faster the clock ticks, the faster the job will get done. The decay of CPU frequency then seems to be a fundamental mistake of computer architectures; however, this analysis becomes more complex when we take into account two critical variables in the equation: power consumed and heat dissipated. Intel calls the speed/power tradeoff a "fundamental theorem of multicore processors" (The Intel Corporation, 2017), as these two metrics must be balanced to achieve the most efficient platform. A solution that has been utilized over the last decade is the use of multiple cores inside a single CPU chip. The increase of logical cores allows instructions to be parallelized so the device performs better. As a matter of fact, Intel reports that underclocking a single core by 20% saves half the power while sacrificing just 13% of the performance. That means that if you divide the work between two cores running at an 80% clock rate, you get 73% better performance for the same power (Philip E. Ross, 2008).

Given the previous analysis, a discussion of the proper techniques for parallelization is in place. Work developed in (Fernandez et al., 2016) divided the parallelization strategies into three main groups as follows:

Proceedings of the 1st GNU Radio Conference, Copyright 2016 by the author(s).

- **Instruction-level parallelism**: A model in which compilers put considerable effort into the organization of the programs such that functional units and paths to memory are busy with useful work. Unfortunately, several studies showed that typical applications are not likely to contain more than three or four different instructions that can be fed to the computer at a time in this way, limiting the reach of this approach (Fernandez et al., 2016).
- **Task parallelism** A fundamental model of architectural parallelism is found in shared-memory parallel computers, which can work on several tasks at once by parceling tasks out to different processors by executing multiple instruction streams in an interleaved way in a single processor.
- **Data parallelism**: A model in which instructions can be applied to multiple data elements in parallel, thus exploiting data parallelism. This computer architecture extension is known as Single Instruction, Multiple Data (SIMD).

Work developed in this paper will focus on the exploitation of data parallelization techniques in processors through SIMD instructions.

2. Intel SIMD Extensions

The Intel 64 and IA-32 Architectures are some of the most dominant computer architectures in the world. Some reports list them as being used in 80% of the personal computers in the world today. This becomes more relevant when another competitor in the market, Advanced Micro Devices (AMD), has an architecture that is compatible with Intel's development. Data parallelism in the Intel and AMD architectures happens through the SIMD extension for each processor. These circumstances create a scenario where layers of software are placed on top of the specific instructions to create generic implementations as seen per the developer.

The first extension of Intel supporting SIMD was the Intel MMX, unoficially known as Multi Media Extension (MMX). The extensions were initially released to support graphics and multimedia operations (Greene, 1997), but given the resemblance of operations, they represented an ideal set of instructions for Digital Signal Processing (DSP). After MMX, Intel released the SSE instructions. The SSE instructions introduced eight 128-bit registers, that were used for floating point and integer operations across its many versions.

Intel's AVX instruction set was the first of its kind in supporting parallel operations of 256-bit size by promoting the legacy 128-bit SIMD instruction sets that operate on

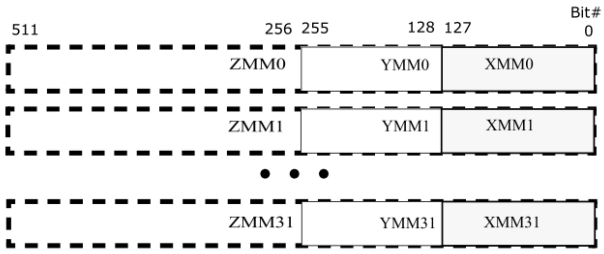

Figure 2. AVX512 ZMM registers for SIMD operations as an extension of the AVX YMM register and SSE XMM registers (The Intel Corporation, 2017)

XMM registers to the YMM registers (The Intel Corporation, 2017). AVX extended most of the 128-bit size operations into its 256-bits extension to support a three-operand syntax. As an enhancement to Intel AVX, Intel also introduced the Fused Multiply Add (FMA) extensions to provide high-throughput, arithmetic capabilities covering fused operations combining multiplication with addition and subtraction. Intel AVX2 extends the AVX instructions by promoting most of the 128-bit SIMD integer instructions with 256-bit numeric processing capabilities. Intel AVX2 instructions follow the same programming model as AVX instructions. In addition, AVX2 provides enhanced functionalities for broadcast/permute operations on data elements, vector shift instructions with variable-shift count per data element, and instructions to fetch non-contiguous data elements from memory (The Intel Corporation, 2017).

The latest extension to SIMD operations released by Intel was AVX512. The AVX512 extension family comprises a collection of 512-bit SIMD instruction sets to accelerate a diverse range of applications. Intel AVX512 instructions provide a wide range of functionalities that support programming in 512-256-128-bit vector registers, plus support for opmask registers and instructions operating on opmask registers.

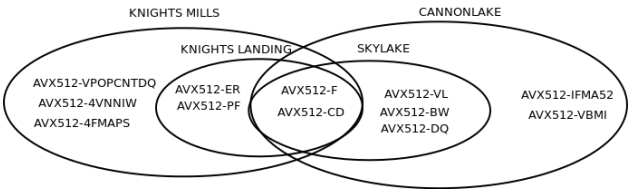

Figure 3. AVX512 Instructions available on a range of Intel processors (Wikipedia contributors, 2018)

2.1. AVX-512 Instructions

The Intel AVX512 Instructions have around 12 subset instructions; some are supported by the Intel Xeon Phi processors starting with the Intel Knights Corner (KNC) architecture [1], and some will be supported by Intel Xeon processors. The distribution of supported instructions per processor type is shown in Figure 3.

2.1.1. AVX512-F

The "Foundation" instruction set is available on both processor types (Intel Xeon Phi and Intel Xeon). It contains vectorized arithmetic operations, comparisons, type conversions, data movement, data permutation, bitwise logical operations on vectors and masks, and miscellaneous math functions like min/max (The Intel Corporation, 2017). This is similar to the core feature set of the AVX instruction set, with the difference of wider registers and additional support in double precision and integer data types.

2.1.2. AVX512-CD

The "Conflict Detection" instruction set is available on both processor types (Intel Xeon Phi and Intel Xeon). It contains vectorized operations for memory conflict detection, counting leading zeros count and performing broadcast operations.

2.1.3. AVX512-ER

The "Exponential and Reciprocal" instructions are designed to help implement transcendental operations and are only available on Xeon Phi type of processors. This set contains instructions for base 2 exponential functions (i.e., 2^x), reciprocals, and inverse square roots. These instructions are available in both single and double precision, with rounding and masking options.

2.1.4. AVX512-PF

The "Pre-fetch" instruction set contains pre-fetch instructions for gather and scatter instructions. Even though these instructions provide software pre-fetch support, Knights Landing (KNL) processors have a much heavier emphasis on hardware pre-fetching.

2.1.5. AVX512-BW

The "Byte and Word" instructions allow support to 8 and 16-bit arithmetic and memory handling operations for the 512-bit registers.

[1] KNC was the first generation of Intel Xeon Phi processors

2.1.6. AVX512-DQ

The "Double-word and Quad-word" instructions support arithmetic operations on Intel 64 and IA-32 Architecture's double-word (32-bits) and quad-word (64-bits) data types.

2.1.7. AVX512-VL

The "Vector Length" instructions allow for the AVX-512 to operate on XMM (128-bits) and YMM (256-bits) registers

2.1.8. AVX512-IFMA

The "Integer FMA" instructions support integers FMA operations with 512-bit precision resolution.

2.1.9. AVX512-VBMI

The "Vector Byte Manipulation Instructions" instructions add additional capabilities not present in AVX512-BW and extend operational data types.

2.2. Intel Xeon Phi

The Intel Xeon Phi processors are highly parallel processors with the Intel Many Integrated Core (MIC) architecture (The Intel Corporation, 2017). Parallelism is present in these processors at two levels: task parallelism and data parallelism. Task parallelism comes from the multiple number of cores in the processor while data parallelism comes from support for the SIMD instructions.

Work presented here was tested in the Intel KNL [2], which was released in 2016 and supports some of AVX512 extensions discussed before. However, KNL, was not the first processor to support the 512-bit instruction set. Back in 2012, Intel released its KNC processor, which was the first Intel architecture processor to support 512-bit vectors. The KNL architecture has a peak theoretical performance of 6 Tera Floating Point Operations per Second (TFLOP) in single precision, which is triple of what KNC had. This performance gain is partly due to the presence of two Vector Processing Units (VPU) per core, doubled compared to the previous generation (Zhang, 2016).

2.3. Intel Intrinsics Instructions

The Intel Intrinsic Instructions are C-style functions that provide access to the SIMD instructions that were previously only accessible through assembly code. The speed up presented in this work is by means of the intrinsics available to the CPU.

The function calls available on the intrinsics are mapped into assembly code by the compiler before the program is run. When an operation uses the intrinsics, it will try to

[2] KNL was the second generation of Intel Xeon Phi processors

take advantage of the CPU resources through parallelization. The concept of maximum resource utilization is also a common practice by compilers when parsing code. As such, it is worth asking if there a use for the intrinsics instructions. Can compilers make use of the SIMD instructions when compiling code? How do the two approaches compare?

To answer these questions, an analysis of the compiler autovectorization tools is necessary. Autovectorization happens when compilers try to parallelize the code of a given application using several techniques, including register mapping, software pipelining, and loop unrolling (Naishlos, 2004). Compilers are actually quite good at performing the latest techniques and can, on some occasion, return better or similar results to manual optimization (Rondeau et al., 2013). However, for more complex operations, compilers are not able to fully parallelize the code to avoid mixing the logic in code originally intended by the developer. As this paper will show, Software Defined Radio (SDR) signal processing has a plethora of operations where intrinsics show better performance than the autovectorization tools of a compiler. This paper, however, does not use inline assembly to accomplish the speed up process, it rather make efficient use of intrinsics programming through Vector Optimized Library of Kernels (VOLK). This approach is ideal because of code simplicity, decreased development time, and portable efficiency.

3. VOLK Implementation

VOLK as a library is an abstraction designed to fix these problems of autovectorization and portability. The autovectorization dilemma is fixed by means of efficient loop unrolling techniques that follows the pseudocode of Listing 1. Each input vector is divided in data chunks that fit in the targeted register size, then operations are performed in a parallel fashion before being stored in the output memory space selected by the calling function. Most functions will finish then with the equivalent serial operations applied to the data elements that do not fit in the register size.

The issue of portability is fixed by providing a platform-agnostic interface called a kernel for each conceptual execution unit subject to SIMD vectorization (Rondeau et al., 2013). At its base, VOLK has a set of proto-kernels designed for particular platforms, SIMD architecture versions, or run-time conditions. The VOLK library compiles all possible proto-kernels supported by the compiler toolchain. At run-time, during the first call to an abstract kernel, VOLK resolves the kernel to a specific proto-kernel. VOLK tests the run-time platform for its capabilities to ensure the resolved proto-kernel will run correctly and employs a dynamic rank-ordering to select the best possible proto-kernel (Rondeau et al., 2013).

Listing 1. Generic protokernel pseudocode with loop unrolling

```
#ifdef LV_HAVE_ARCH

#include <extension include files>

static inline void
volk_in-tag_kernel-desc_out-tag_align_proto-kernel-desc
(input_params){

  // Variable declarations
  .....

  // Loop unrolling
  // Data parallelization over SIMD register
  for() {
    // Load data from memory to register
    .....

    // Perform kernel operations
    .....

    // Store data from register to memory
    .....
  }

  // Serial loop
  // Items remaining after parallelization
  for () {
    // Perform kernel operations
    .....

    // Store data to memory
    .....
  }
}
#endif
```

Listing 2. Sample code for multiply proto-kernel using AVX512-F Intrinsics Extensions

```
#ifdef LV_HAVE_AVX512F
#include <immintrin.h>

static inline void
volk_32f_x2_multiply_32f_u_avx512f(
float* cVector, const float* aVector,
const float* bVector, unsigned int num_points)
{
  // Variable declarations
  unsigned int number = 0;
  const unsigned int sixteenthPoints = num_points / 16;
  float* cPtr = cVector;
  const float* aPtr = aVector;
  const float* bPtr= bVector;
  __m512 aVal, bVal, cVal;

  // Loop unrolling
  // Data parallelization over SIMD register
  for(;number < sixteenthPoints; number++){
    // Load data from memory to register
    aVal = _mm512_loadu_ps(aPtr);
    bVal = _mm512_loadu_ps(bPtr);

    // Perform kernel operations
    cVal = _mm512_mul_ps(aVal, bVal);

    // Store the results back into the C container
    _mm512_storeu_ps(cPtr,cVal);
    aPtr += 16;
    bPtr += 16;
    cPtr += 16;
  }

  // Serial loop
  // Items remaining after parallelization
  number = sixteenthPoints * 16;
  for(;number < num_points; number++){
    *cPtr++ = (*aPtr++) * (*bPtr++);
  }
}
#endif /* LV_HAVE_AVX512F */
```

3.1. Programming Model

As mentioned before most of the code development in VOLK follows the pseudo-code shown in Listing 1. Line **1** is a safeguarded to ensure that only supported architectures are processed. Line **6** is reserved for the function name. As a rule, all VOLK proto-kernels will start with the word *volk*, after which the input tags will follow. The *Input tags* field in the VOLK environment refer to the bit count and data types the function will receive, e.g. `32fc` defines a 32 bit floating point complex number. The *kernel descriptor* field names the kernel being implemented with the architecture. This is simply the name of the operation the function will perform like multiply, divide, etc. The *output tag* field, similar to its counter part for the input parameter, describes the bit count and data types of the results of the operation. The *alignment* field hold the type of alignment used for the data loaded from memory. Finally the *proto-kernel descriptor* field identifies the specific architecture being targeted by the kernel, i.e. `sse`, `avx`, `avx2`, `avx512f`, etc.

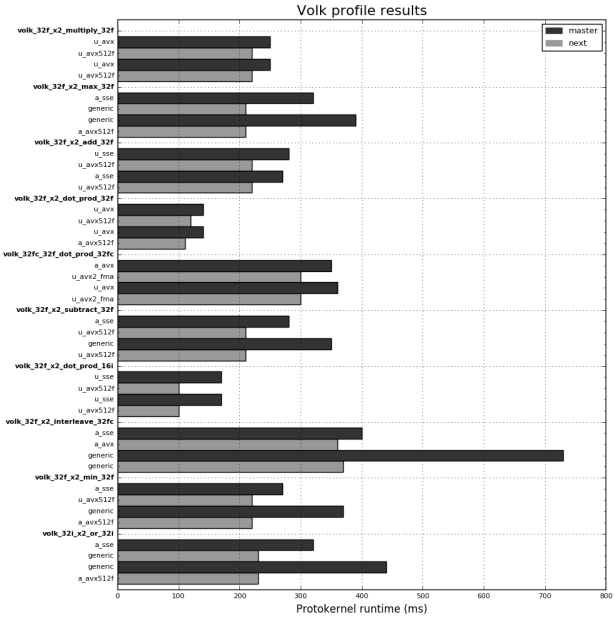

Figure 4. VOLK Profiling results for Xeon Phi KNL, highlighting AVX512-F improvements on the `32f_x2` kernels

Line **14** starts the process of data parallelization with SIMD registers. Initially the incoming data is loaded into the register type targeted by the proto-kernel. Starting with the SSE extensions, a dedicated set of intrinsics exists to perform "load" operations, handling internally the alignment of the input data (The Intel Corporation, 2017). Line **18** will start the chain of operations needed to complete the proto-kernel functionality. Most operations, when targeting `x86` systems will be completed by means of the intrinsics extensions available to the system (The Intel Corporation, 2012). The online guide presented in (The Intel Corporation, 2012) organize the intrinsics by extensions and is in the author opinion the most updated document available for the code development. Line **21** will store the results in the output variables dictated in the function call after the operations required by the kernel are completed. In similar fashion to the loading procedure, moving data from the dedicated registers to the output variables is managed internally by the intrinsics available to the processor.

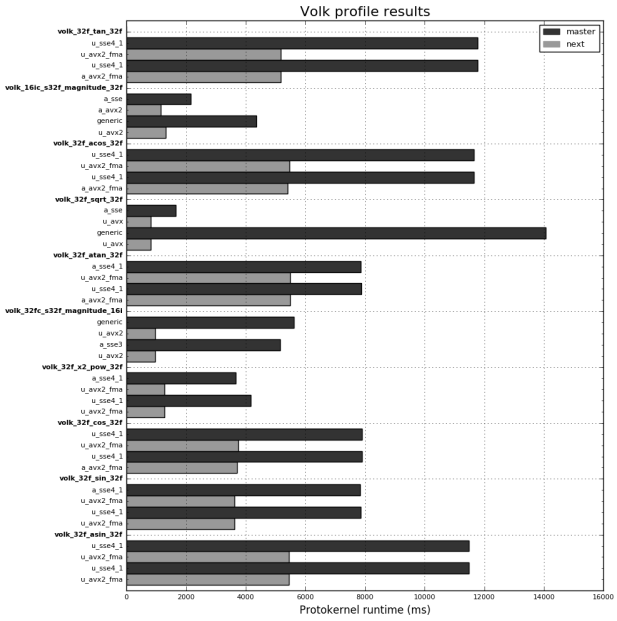

Figure 5. VOLK Profiling results for Xeon Phi KNL, showing AVX2 and FMA improvments on the 32f kernels

A final piece of the proto-kernel operation is to perform the operation on the remaining data elements that were not able to be parallelized by the loop unrolling techniques as shown in Line **27**. Code in this section will follow the generic implementation with standard library function calls.

Listing 2 shows sample code for one of the proto-kernels developed during this work. The code shows the same guidelines offered in the generic pseudo code listing shown before. Line **1** shows the targeted architecture of the proto-kernel, AVX512-F in this case. Line **5** shows the function's name following the guidelines described before where:

- *volk*: required key word `volk` in use to start each proto-kernel.
- *input tags*: defined as `32f_x2`, indicating that it receives two (x2) input vectors of 32 bits floats values (32f)

- *kernel descriptor*: defined as `multiply`, indicates that a multiply operation is to be performed
- *output tags*: defined as `32f`, indicating that results of multiplication is to be saved in a single 32 bits float value (32f)
- *alignment tag*: input data is to be handle as unaligned (u)
- *proto-kernel descriptor*: defined as `avx512f`, indicates that proto-kernel targets AVX512-F extensions

Starting with line **10**, all variable declarations are performed, including the special types defined by the compiler to handle the intrinsics operation. Line **19** starts the loop unrolling process followed by the loading, multiplication and storage operation. For vector sizes not multiple of the register size targeted, a serial operation is performed in the remaining data elements as shown in Line **36**. Listing 2 is just one of the many kernels added during this work and should be available in the VOLK GitHub page in the near future.

4. Profiling Results

A total of 82 kernels were improved by adding AVX, AVX2, FMA, and/or AVX512 capabilities. The AVX and AVX2 intrinsics were added to most of the kernels, while AVX512-F, AVX512-CD was only added to a subset due to the limited set of instructions/operations available in the extension set. Specifically, the lack of permute, xor, and shuffle operations limited the kernels that could be improved using AVX512-F and the Intel KNL architecture.

Figures 4-7 show the profiling results for a selection of kernels. For each kernel, the profiling time for the fastest aligned and unaligned versions of the kernel is plotted and compared between the `master` branch of VOLK and the `next` branch on which the AVX2 and AVX512 improvements were made. It is important to note that although the absolute proto-kernel runtimes seem long, we are only concerned with the relative differences between the runtimes of comparable kernels. The runtimes are slower because the KNL processor was used to perform profiling, which uses slower CPU frequencies. The majority of the VOLK kernels show a much faster runtime with the AVX2/AVX512-F improvements. Before these improvements, most of the kernels used the generic or SSE instruction sets as their fastest implementation. The generic implementation is generally especially slow because it does not take advantage of any parallelization or use the Intel intrinsics. However, with the AVX, AVX2, and AVX512-F additions, many of the kernels are faster now than their previous implementation.

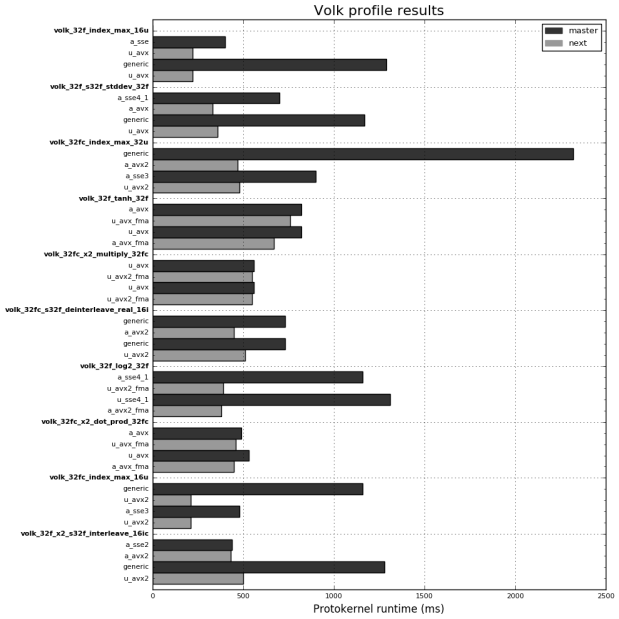

Figure 6. VOLK Profiling results for Xeon Phi KNL, showing AVX2 and FMA improvments on several 32fc and `32f` kernels

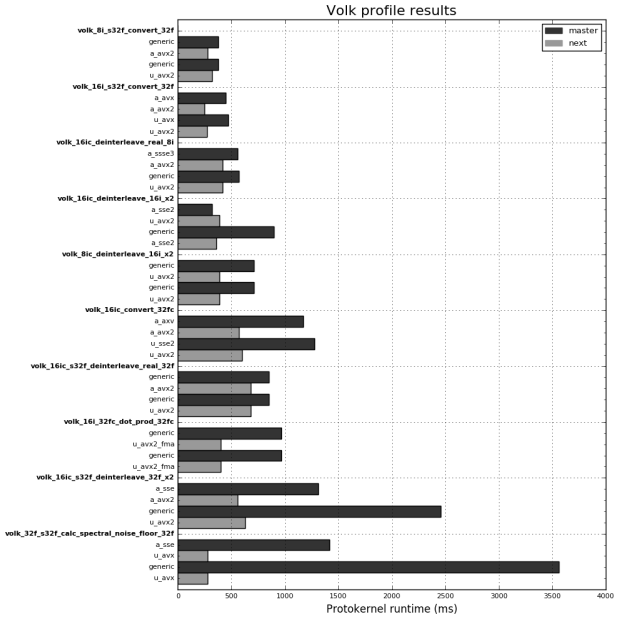

Figure 7. VOLK Profiling results for Xeon Phi KNL, highlighting AVX2 improvments on some of the convert and deinterleave kernels

The `volk_32f_sqrt_32f` kernel is one of the functions that shows the most improvement (Figure 5), from 14070 ms using generic to 820 ms using AVX, a 94% improve-

ment in speed. Many of the other kernels show performance improvement of 50% or more, especially many of the more complex 32f math operations such as sine and power. Most of the significant improvements in speed come from an upgrade from generic or SSE to AVX2 (mostly in Figures 5-7). Smaller performance improvements were observed between AVX2 and AVX2 with FMA, and between AVX2 and AVX512-F (Figure 4). Greater performance improvements were also realized on kernels that required more mathematical processing, as these used more instructions, allowing more improvement when the parallel processing was increased from 128-bit to 256-bit or 512-bit.

5. Conclusions

The results in this paper show that the addition of AVX2 and AVX512 to the VOLK Library kernels will provide great improvement to the software using its mathematical capabilities such as GNU Radio or Global Navigation Satellite System Software Defined Radio (GNSS-SDR)(Fernández–Prades et al., 2011). These improvements were only experimented on Xeon Phi `x86` processors, but cross compatibility with similar architecture like that of Xeon processors should allow the results to be easily usable. In addition because the AVX512 capabilities are only present in `x86` processors, we did not focus on the implementation of similar technologies in embedded architectures, particularly those provided by Advanced RISC Machines (ARM) processors.

Given the evolution of Intel processors and the proliferation of the processors in consumer markets that support this set of advanced extensions, it is only natural to assume that further work will be focused on supporting the remaining AVX512 flavors, i.e AVX512-BW, AVX512-DQ, etc with the intention to further improve DSP speeds in SDR applications.

6. Software and Data

Code developed for this work was submitted to the VOLK repository as a pull request. At the moment of this writing the pull request was accepted and pending a merge process with the master branch. Details of the pull request can be found at `https://github.com/gnuradio/volk/pull/190`

Acknowledgments

The authors would like to thank Eric K. Lai and the Digital Communications Implementation department at The Aerospace Corporation for their contributions to this work. They would also like to thank the GPS Directorate Advanced Technology Branch for sponsoring this investigation of software radios.

References

Fernandez, Carles, Arribas, Javier, and Closas, Pau. Accelerating GNSS Software Receivers. In *Proceedings of the 29th International Technical Meeting of The Satellite Division of the Institute of Navigation (ION GNSS+ 2016)*, pp. 44–61, Portland, OR, 2016.

Fernández–Prades, C., Arribas, J., Closas, P., Avilés, C., and Esteve, L. GNSS-SDR: An open source tool for researchers and developers. In *Proc. of the ION GNSS 2011 Conference*, Portland, Oregon, Sept. 2011.

Greene, M. A. Pentium(r) processor with mmx/sup tm/ technology performance. *Proceedings IEEE COMPCON 97. Digest of Papers*, pp. 263–267, 1997.

Karl Rupp. 42 Years of Microprocessor Trend Data, 2018. URL `https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/`.

Naishlos, Dorit. Autovectorization in GCC. *Proceedings of the 2004 GCC Developers Summit*, pp. 105–118, 2004.

Philip E. Ross. Why CPU Frequency Stalled -, 2008. URL `https://spectrum.ieee.org/computing/hardware/why-cpu-frequency-stalled`.

Rondeau, T, McCarthy, N, and O'Shea, T. SIMD Programming in GNU Radio: Maintainable und User-Friendly Algorithm Optimization with VOLK. *WinnForum's SDR conference*, 86:101–110, 2013. URL `http://50.19.239.22/redmine/attachments/422/volk.pdf`.

The Intel Corporation. Intel Intrinsics Guide, 2012. URL `https://software.intel.com/sites/landingpage/IntrinsicsGuide/{#}techs=AVX,AVX2{&}expand=3924,3894,2758`.

The Intel Corporation. Intel ® 64 and IA-32 Architectures Software Developer ' s Manual Volume 1 : Basic Architecture. Technical Report 253665, The Intel Corporation, 2017.

Wikipedia contributors. Avx-512 — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=AVX-512&oldid=837564204`, 2018. [Online; accessed 23-April-2018].

Zhang, Bonan. Guide to Automatic Vectorization with Intel AVX-512 Instructions in Knights Landing Processors. Technical report, Colfax International, 2016. URL `https://colfaxresearch.com/knl-avx512/`.