1 A SIMD Intrinsic Correlator Library for GNSS Software Receivers

2 **Damian Miralles • Dennis M. Akos**

4 **Abstract**

5 An open source implementation of a Code Division Multiple Access (CDMA) software
6 correlator library that leverages Single Instruction Multiple Data (SIMD) is presented. We
7 initially discuss the key aspects involved in the correlation operation for software radio
8 applications. Afterward, we present the state of the art Application Programming Interface
9 (API) that provides SIMD capable methods for each of the components in a correlation
10 operation, including the first of its kind parallelized code and carrier generation using lookup
11 tables and SIMD instructions. The library was developed using SIMD Intrinsic Instructions,
12 which are a C type nomenclature offering access to the assembly instructions originally
13 designed for the SIMD extensions in the processor. This design paradigm presents an advantage
14 in terms of readability and simplified code development to accommodate future modifications.
15 Recorded data were used with a standalone Global Navigation Satellite System (GNSS)
16 software receiver where the methods hereby presented were tested and profiled to validate
17 theoretical assumptions.

18

19 **Keywords**

20 GNSS Correlation • SIMD Instructions • software radio

21

22 D. Miralles (✉) • D. M. Akos

23 Ann and H.J. Smead Aerospace Engineering Sciences • University of Colorado at Boulder

24 Email: damian.miralles@colorado.edu • ORCID ID: <https://orcid.org/0000-0001-5820-9569>

25

26 **Introduction**

27 Software Defined Radio (SDR) is an emerging technology that seeks to build flexible, multi-
28 service, multi-band, multi-standard, reconfigurable, and reprogrammable through software,
29 radio systems. This definition, borrowed from Principe et al. (2011), describes very well the
30 key idea of the SDR design paradigm. More specifically, an ideal SDR transceiver, sketched in
31 Fig. 1, is characterized by a Radio Frequency (RF) Front End (FE) that is connected to a Digital
32 to Analog Converter (DAC) and/or an Analog to Digital Converter (ADC) depending on the
33 operation being performed. The device normally features a Universal Software Bus (USB)
34 adapter to allow communication with the host processor, although network interfaces have also
35 been used. At the end of the chain, a Programmable Processing Unit (PPU) serves as a data
36 processing module in charge of baseband conversion, modulation, etc. Software is then written
37 in the processor to execute a given task depending on the application at hand.

38

Fig. 1 Software radio design diagram

The effectiveness and the advantage of a fully SDR platform like the one shown before with respect to a fully hardware equivalent design can be synthesized in three key points (Principe et al. 2011).

1. *Configurability*: Within a specific communication technology, this means that a unique transceiver for different applications can be factory-configured without any hardware change.
2. *Upgradeability*: Since the core is software based, the receiver can be updated and upgraded in case of new and improved algorithms or amendments to a standard, while maintaining the same hardware components.
3. *Flexibility*: The same transceiver, with the same hardware, can be used for different communication technologies, simply by reconfiguring or upgrading the software.

Most hardware devices are usually constrained and confined to certain applications. Mitola (1995) first stated that the direct consequence of this limitation is the need to use multiple hardware solutions for different technologies, and the possible need of hardware replacements with the change in standards or technologies.

GNSS has witnessed the advent of several new systems and technologies after the landmark design and development of the Global Positioning System (GPS). In this framework, an SDR-based receiver has no difficulty in performing the necessary functions of data processing to accommodate the new signals broadcast by GNSS constellations.

The received digitized signal by the RF front end, as defined by Psiaki (2001) follows the following model:

$$y_k = Ad(t_k)c[(1 + \eta)(t_k - \tau_s)]\cos[(\omega_{IF}t_k - (\omega_D t_k + \varphi_0))] + v_k \quad (1)$$

where y_k is the sampled output from RF front end at time t_k , A is the signal amplitude, $d(t_k)$ is the navigation symbols, $c(t)$ refers to the ranging code transmitted, η is the fractional perturbation due to Doppler shift in the ranging code chipping rate, t_s is the start time of the ranging code, ω_{IF} is the nominal intermediate frequency after down conversion, ω_D is the carrier frequency's Doppler shift, φ_0 is the initial carrier phase, and v_k is the un-modeled noise in the system. Equation (1), due to mathematic tractability assumes an ideal clock but that not need be the case for the processing of the signal in real case scenarios.

Typical processing of GNSS signals is divided into four main stages, regardless of the constellation of origin. First, the acquisition stage oversees detecting the targeted signal with rough estimates of its base parameters: code phase (τ_s) and Doppler frequency (ω_D). Second, a tracking stage starts with the acquisition estimates and uses those to further refine the

estimations of the code phase (τ_s), Doppler frequency (ω_D), and carrier phase (φ). Third, a telemetry decoding stage oversees obtaining the ephemeris, almanac, and clock parameters broadcasted in the navigation message by the satellite. Finally, the decoded information and time delay parameters are used to compute a position solution through a process called trilateration.

At the heart of the tracking and acquisition stage, lays the correlation operation. Due to its coarse nature, the acquisition stage can also be implemented using Fast Fourier Transform (FFT) operation. However, the tracking stage is inherently dependent on the correlation operation, which is typically implemented in a hardwired Application-Specific Integrated Circuit (ASIC) and is used to keep track of acquired signals. The correlation operations is defined as:

$$Z_I = \sum_{k=0}^{N-1} y_k c[(1 + \tilde{\eta})(t_k - \tilde{\tau}_s)] \cos[(\omega_{IF} - \tilde{\omega})t_k] \quad (2)$$

$$Z_Q = \sum_{k=0}^{N-1} y_k c[(1 + \tilde{\eta})(t_k - \tilde{\tau}_s)] \sin[(\omega_{IF} - \tilde{\omega})t_k] \quad (3)$$

where Z_I and Z_Q are the correlator outputs for the in phase and quadrature channels, N is the sampled code size, and the remaining terms are as defined in (1). Based on the equation above, it is safe to assume that at its very base, assuming a generic GNSS signal, the correlation can be decomposed into:

1. Carrier wave generation at a given carrier frequency (normally combination of IF and Doppler frequency), carrier phase remainder, and sampling frequency.
2. Pseudorandom code generation at a given code frequency, code phase, and sampling frequency.
3. Down-conversion of the received signal by multiplication with a carrier wave generated in step 1 above.
4. Multiplication of baseband signal with a local replica of pseudorandom code generated.
5. Accumulation of resulting signal to generate the correlation value of the sampled period.

The process previously described is normally repeated (except steps 1 and 2) for each of the correlators the receiver uses; typical applications for GPS L1 C/A signal tracking will use three correlators each with an in-phase and a quadrature component per channel. If combined in a 12-channel single-frequency receiver, we will end up having a total of 72 correlators operating each millisecond. Couple this result with one second of processing time and this quickly becomes a computationally intensive operation where 72,000 correlations are needed per second of data processing. Even more, with the presence of multiple signals sharing the spectrum, it is common to have the need to support multi-constellation, multi-band frequencies, increasing the number of correlations needed for processing. The demands of the correlation operation, especially when implemented in a software receiver, place a huge burden on computational resources.

Over the past decade, there has been a shift in the design paradigm of Central Processing Unit (CPU) manufacturing. Figure 2 shows that the steady trend, for more than 25 years, to

increase CPU operating frequency for speed improvements has slowed. The paradigm-shift nowadays is in the rise of the number of logical cores, which brings speed improvements and a reduction in the power consumption of the device.

Fig. 2 Microprocessor Trend Data for 42 years. Obtained with permission from the author. (Kupp 2018).

Given the previous analysis, a discussion of the proper techniques for parallelization is worthwhile. Fernandez et al. (2016) divided the parallelization strategies into three main groups as follows: (1) Instruction-level parallelism, (2) Task parallelism and, (3) Data parallelism. This work will focus on data parallelism, which is a model in which instructions can be applied to multiple data elements simultaneously. This computer architecture extension is known as SIMD.

Previous Work

Advancing software receivers by means of SIMD extensions is not new to the GNSS community. Heckler and Garrison (2006) first presented a SIMD based correlation library that provided support for multiplication, addition, and accumulation operations. At the time of its writing, and due to the technology available, this work only provided support for the Multi-Media Extension (MMX) and Streaming SIMD Extensions (SSE) extensions. With the advancement of Intel architectures and the release of new extensions, new efforts to use those

for GNSS receivers were developed. Xu et al. (2013) presented a correlator algorithm that made use of the Advanced Vector Extension (AVX) when processing complex data in GNSS software receivers. The aforementioned work did not develop an algorithm for sinusoidal wave generation and was more focused on the development of arithmetic operations for multiplication and addition.

However, the first effort to standardize the use of the SIMD extensions for SDR applications came with the Vectorized Library of Kernel (VOLK) library which provided an abstraction layer for code development of SIMD applications using software intrinsic. The VOLK library presented an abstraction layer when supporting the intrinsic type by developing a platform-agnostic design that selected the most efficient architectures in real-time for the system being used (Rondeau et al. 2012). VOLK has now become the de facto math library for the GNU Radio platform and its use has revolutionized software radio applications. Further work has been developed to keep VOLK and/or inherited modules up to date with the newest extension additions in Intel, Advanced Micro Devices (AMD), or Advanced RISC Machines (ARM) CPUs. Such is the case of the intrinsic support for GNSS-SDR in operations specific to GNSS technologies, see Fernandez et al. (2016) or the updates to the VOLK library and its missing extensions for AVX2 and AVX512 (Miralles and Iwamoto 2018).

Intel SIMD Extensions

Data parallelism in the two most popular computer architectures in the world, Intel and AMD, happens through the SIMD extensions. However, since estimates place the Intel market share of the CPU business at approximately 80%, an evolution of the Intel SIMD extension over time is discussed and shown in Figure 3. The first extension of Intel supporting SIMD was the MMX. These extensions were initially released to support graphics and multimedia operations (Greene 1997), but given the resemblance of operations, it represented an ideal set of instructions for digital signal processing.

Fig. 3 SIMD Evolution in the Intel Architecture with highlights of relevant work on SDR applications with SIMD support

After MMX, Intel released the SSE instructions as an extension of the SIMD execution model introduced with the MMX technology. The SSE instructions introduced eight 128-bit registers, denoted as XMM. These instructions evolved with further versions across the line that added support for floating point and integer operations.

Intel's AVX instructions were the first of its kind in supporting parallel operations of 256-bits size. Figure 4 shows how AVX extended most of the 128-bit size operations into its 256-bit extension to support three-operand syntax. Shortly after, Intel also introduced the Fused Multiply Add (FMA) extensions to provide high-throughput arithmetic capabilities covering fused operations combining multiplication with addition and subtraction.

Fig. 4 Intel SIMD registers size and extension methods

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 Intel SIMD extension is the AVX-512 family which is designed as a set of instructions that expand across several Intel architectures as shown in Figure 5. Table 1 further describes the extensions AVX512-F, -CD, -VL, -BW, and -DQ, which served as the testing platform on the Skylake processor used in this work.

Fig. 5 AVX512 Family support across multiple Intel CPU architectures (Miralles and

Table 1 Description of instructions supported by Skylake processors (The Intel Corporation 2017).

Instruction	Description
AVX512-F	The “Foundation” instruction set 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.
AVX512-CD	The “Conflict Detection” instruction set contains vectorized operations for memory conflict detection, counting leading zeros count and performing broadcast operations.
AVX512-BW	The “Byte and Word” instructions allow support to 8- and 16-bit arithmetic and memory handling operations for the 512-bit registers.
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.
AVX512-VL	The “Vector Length” instructions allow for the AVX-512 to operate on XMM and YMM registers

SIMD Data Type Support

With the evolution of the SIMD instructions on the Intel architecture, an evolution of the data type support has also occurred. The highly-efficient instructions have been created to solve the needs of industries dealing with computer graphics and multimedia, scientific computing, etc. The use of these instructions for GNSS operations is then a natural extension given the resemblance of the applications in terms of data processing.

The Intel SIMD instructions support multiple operations; however, listed in Table 2 the operations that can be used for GNSS signal processing are showcased. The data types listed uses the following notation

1. pi8/pu8, pi16/pu16, pi32/pu32, pi64/pi64: Vectors containing 8, 16, 32, and 64 bit signed and unsigned integers.
2. ps, pd: Vector containing 32, 64 bit single and double precision floating point numbers.

In the case of GNSS signal processing, work developed by Pany (2010) showed that signal detection and processing can happen with quantization levels of 1-4 bits without a significant loss in the received signal power. This then translates to the ability to effectively perform GNSS

signal processing with 8 or fewer bit data types. Previous work even used 1-bit quantization levels and performed software correlation by means of bit-wise parallel algorithms (Ledvina et al. 2004). However, such low levels of data quantization are impractical and non-flexible for software radios since fewer operations are supported natively on the processor. This works then leverages the SIMD instructions and its default data types to perform the sub-operations in the correlation module of the receiver. Such an approach ends up offering flexible operations that support 32-bit integers and single precision floats.

Intel Intrinsic Instructions

The Intel Intrinsic Instructions are C-style functions that provide access to the SIMD instructions previously only accessible through assembly code. The C-style nomenclature offers two advantages over inline assembly development: (1) simplicity and decreased development time, and (2) portable efficiency. The function calls available on the intrinsic are mapped into assembly code by the compiler before the program is run. Use of the intrinsic by any given operation will try to gain advantage of the CPU resources by means of parallelization. A similar approach is followed by compilers when processing code in elevated levels of optimization through a process called auto-vectorization. Auto-vectorization happens when compilers will try to parallelize the code of a given application using several techniques, including register mapping, software pipelining, and loop unrolling. Compilers can in some occasion return better or similar results to manual optimization (Rondeau et al. 2012). However, for more complex operations, compilers are not able to fully parallelize code to avoid the order the user initially selected. As we will show, GNSS signal processing has a plethora of operations where intrinsic instructions show better performance than the auto-vectorization tools of a compiler.

Table 2 Supported data types and operation types for Intel Intrinsic as it relates to GNSS correlation. (A) defines addition, (M) defines multiplication, (X) defines not supported, and (F) defines data fetching operations.

Technology	pi8/pu8	pi16/pu16	pi32/pu32	pi64/pu64	ps	pd
MMX	A	A,M	A,M	X	X	X
SSE	X	X	X	X	A,M	X
SSE2	A	A,M	A,M	A,M	A,M	A,M
SSE3	X	X	X	X	X	X
SSSE3	X	X	X	X	X	X
SSE4.1	X	A,M	A,M	X	X	X

SSE4.2	X	X	X	X	X	X
AVX	X	X	X	X	A,M	A,M
FMA	X	X	X	X	A,M	A,M
AVX2	A	A,M	A,M,F	A,M,F	A,M,F	A,M,F
AVX512	A	A,M	A,M,F	A,M,F	A,M,F	A,M,F

Parallelized Lookup Table (PLUT) Method

The Intel AVX2 and AVX512 extensions introduced for the first time the ability to fetch in parallel data from memory in a non-continuous manner. Such fetching operations, as it applies to the correlation operation, were identified by extension type in Table 2. These sets of instructions opened a new set of possibilities for parallelization of lookup table implementation for GNSS correlations. The new instructions were used to target key components of the correlation, (1) the parallelization of the carrier wave generation, (2) the parallelization of the pseudorandom code and (3) the multiplication/accumulation operations. This work builds upon the multiplication/accumulation techniques by means of SIMD instructions developed by Heckler and Garrison (2006), and as such, are not discussed in detail here. Additional sources that further describe the multiplication/accumulation approach is discussed in detail by Rondeau et al. (2012).

Lookup Table Carrier Wave Generation

Phase or frequency controlled sinusoidal wave generation is critical for signal demodulation and Doppler correction applications. Traditionally sine and cosine generation have been a very expensive operation for software radio and multiple approaches have been considered to reduce the computational complexity of the operation. In the GNSS community, a very popular option is the direct lookup table (DLUT) method (Matic et al. 1998), which is based on the reading of precomputed samples from memory. The precomputed samples typically represent the values of one sinusoidal function equally spaced across a single wave period.

The DLUT method can be better described by Figure 6, where the DLUT vector has $N = 2^n$ entries, with n being the number of address bits generated by the accumulator, b being the fractional-phase accumulator, L being the output bits, and the step size μ determines how quickly the address is accumulated. The DLUT approach is affected by a reduction in the phase resolution of the desired wave. However, internally generated sine/cosine carriers can be represented by a 1-bit amplitude, causing only a loss of 0.9 dB (Pany 2010), making this method an excellent option for wave generations in GNSS correlation. A detailed description of the DLUT method is outside the scope of this paper. However, the topic has been discussed widely, see Matic et al. (1998).

275

276

277

Fig. 6 Generic sine/cosine DLUT system description

278

279 Based on the concepts of a lookup table for carrier wave generation and given the new
 280 support for fetching instructions for AVX2 and AVX512, a novel approach is presented in Fig.
 281 7. The algorithm follows the same principles described before but parallelizes those by means
 282 of the intrinsic instructions. Finding the indexes to address the lookup table is a predictable
 283 procedure that always adds a constant offset to the fixed-point accumulator used. As such, at
 284 initialization, a YMM register is set with the first possible offset for the counter. It is followed
 285 by a right shift operation that obtains the index needed to access the table. The length of the
 286 shift and the index size will be dependent on the bit size resolution used during the lookup table
 287 design procedure. For our implementation, a [8:24] system is used. Once the index is properly
 288 computed in the register, a non-continuous memory access occurs based on the indexes values.
 289 The latest memory fetching operation represents a huge advantage over previous
 290 implementations. Access to memory, when using 32-bits variables, happens eight samples at a
 291 time using YMM registers or sixteen samples at a time using ZMM registers. After access to
 292 memory is complete the accumulation counter is recomputed, adding the constant offset
 293 selected. Using the AVX extension, this process happens simultaneously in the register. The
 294 parallelized loop finishes when the obtained data values are stored into the output array.

295

Fig. 7 AVX2 PLUT method for carrier wave generation implementation.

Lookup Table Pseudorandom Code Generation

A similar approach can be used for the pseudorandom code implementation by means of a combination of lookup tables and the fetch instructions for the SIMD intrinsic. Using a DLUT approach for the code generation does not incur in-phase truncation errors commonly seen in sine/cosine generation given the square-wave nature of the pseudo-random code. However, if using an LUT with the original code length, e.g. 1023 for GPS L1 C/A, 511 for GLONASS L1OF, a process of re-sampling is normally required to generate the sampled code. Multiple implementations exist, but a traditional approach follows the design of Fig. 8.

Fig. 8 Sampled code resampling generation based on DLUT

Assume that a code chip rate $f_c = 1.023 \text{ MHz}$, chip width $t_c = \frac{1}{f_c} = 0.9775 \mu\text{s}$,
 sampling frequency $f_s = 4 \text{ MHz}$, sample interval $t_s = \frac{1}{f_s} = 0.25 \text{ ms}$, code length $L =$
 1023 chips , number of samples M , and the time vector $t = 0, t_s, 2t_s, 3t_s, \dots, (M-1)t_s$.
 Then, the process of resampling can be achieved by $idx = \text{floor}\left(\frac{t}{t_c}\right)$, which eventually will
 map chip to sample values as $samples = code[idx]$.

Based on the concepts described before and given the new support for fetching instructions
 for AVX2 and AVX512, a novel approach is presented in Fig. 9. The algorithm follows the
 same principles described before but parallelizes those by means of the intrinsic instructions.
 Finding the indexes to address the lookup table is a predictable procedure that always adds a
 constant offset to the accumulator used.

Fig. 9 AVX2 PLUT method for pseudorandom code generation implementation.

Results and Analysis

This addition to the GPS Toolbox is an x86 SIMD library useful to perform faster correlation operations in software radio applications. The resulting library includes methods that perform arithmetic operations in YMM and ZMM registers and supports common data types like floats and signed integers. Some of the methods of the library, especially those dealing with code and carrier generation can be easily extrapolated to other fields where CDMA is in use. In order to illustrate how to use the SIMD instructions in x86 platforms, examples are discussed before analyzing the performance improvements of the proposed library.

Carrier Wave Generation

Table 3 shows code snippets from the carrier wave generation using the PLUT approach with AVX512 support. The code follows the paradigm for kernel development used in VOLK, allowing for ease of portability if adding to the VOLK library. After function initializations lines 44-45 perform a fixed-point operation by shifting each of the 32-bit indexes 24 times to the right, resulting in an 8-bit index used later to address the lookup table. Line 48 fetches the value from memory in a non-continuous way thanks to the new fetching instructions supported

by AVX2 and AVX512. Line 52 updates the base counter to use in the next data chunk. The vector containing the data for the segment is then stored in memory in line 55. Notice the pointer update that happens immediately after in line 58. This brief segment of code presents the parallelized version of the carrier generation, where samples are processed in batches of 512bits at a time. After the parallelized version ends, a serial loop is placed to account for remaining data samples that did not fit in the AVX registers.

Table 3 Carrier wave generation using the PLUT method with AVX512 support. Code snippet only includes parallel loop operation

```

11 void avx512_nco_fl32(float *sig_nco, const float *lut, const int blk_size,
12                     const double rem_carr_phase, const double carr_freq,
13                     const double samp_freq) {
14     .
15     .
42   for (inda = 0; inda < sixteenth_points; inda++) {
43       // Obtain integer index in 8:24 number
44       carr_idx = _mm512_srli_epi32(carr_phase_base, 24);
45       carr_idx = _mm512_and_si512(carr_idx, hex_ff);
46
47       // LUT Access
48       nco = _mm512_i32gather_ps(carr_idx, lut, 4);
49       // Delta step
50       carr_phase_base = _mm512_add_epi32(carr_phase_base, carr_step_offset);
51       // Store values
52       _mm512_storeu_ps((float *)sig_nco, nco);
53       // Update pointers
54       sig_nco += 16;
55   }
56   .
57   .
65   //Serial implementation
66   for (; inda < blk_size; ++inda) {
67       nom_carr_idx = (nom_carr_phase_base >> 24) & 0xFF;
68       *sig_nco++ = lut[nom_carr_idx];
69       nom_carr_phase_base += nom_carr_step;
70   }
71 }
72 }
73 }
74 }

```

An analysis of the latency cycles and throughput count of each operation follows. The latency is how many cycles are required to fully complete the instruction, whereas the throughput is how many cycles are required to wait before introducing the next loop in the pipeline (Heckler and Garrison 2006). When performing a sine or cosine function call, it takes around 120 latency cycles (Fog 2017). This needs to be repeated for every sample of the vector

before demodulation. Assuming a sampling frequency of 4 MHz, then the operation needs to be repeated 4000 times for a single sinusoidal generation. Combining the operation repetition times and latency cycles then a sine/cosine wave generation takes around $120 \times 4000 = 480000$ latency cycles. This cycle count analysis does not consider the latency generated when initializing the function parameters or the multiplication operations needed to compute the angle value, where sin/cos will be called.

When inspecting the PLUT approach, and assuming the same sampling frequency as before of 4 MHz, we end up having 33 cycles per parallel operation. If using the ZMM (512-bit) registers then the parallel cycles count will be repeated $4000/16 = 250$ times, which will end up using $250 \times 33 = 8250$ latency cycles. The latency cycles added for function initialization could be dismissed since they account for a grand total of 8270 cycles.

Table 4 Cycle Latency and Throughput count for carrier wave generation

Intrinsic	Instruction	Latency	Throughput
_mm512_set1_epi32	vpbroadcastd zmm {k}, r32	3	1
_mm512_add_epi32	vpadd d zmm {k}, zmm, zmm	2	0.5
_mm512_srli_epi32	vpsrld zmm {k}, zmm, imm	1	0.5
_mm512_and_si512	vpand d zmm {k}, zmm, zmm	1	0.5
_mm512_i32gather_ps	vgatherdps zmm {k}, vm32z	30	9.75
_mm512_add_epi32	vpadd d zmm {k}, zmm, zmm	1	0.5
_mm512_storeu_ps	vmovups m512 {k}, zmm	1	0.5
_mm512_extracti32x8_epi32	vextracti32x8	5	1
_mm256_extract_epi32	N/A	3	1

Pseudorandom Code Generation

The code generation followed a model similar to that of Table 3, and source code is provided with the library for reference when inspecting the PLUT approach. Assuming the same sampling frequency as before (4 MHz), we end up having 36 cycles per parallel operation per correlator. If using the ZMM (512-bit) registers then the parallel cycles count will be repeated $4000/16 = 250$ times, which will end up costing $250 \times 36 = 9000$ latency cycles. The latency cycles added for function initialization could be dismissed, since they account for a grand total of 9020 cycles.

380

Table 5 Cycle Latency and Throughput count for code generation

Intrinsic	Instruction	Latency	Throughput
_mm512_cvtps_epi32	vcvtps2dq zmm {k}, zmm	4	0.5
_mm512_i32gather_ps	vgatherdps zmm {k}, vm32z	30	9.75
_mm512_add_ps	vpaddq zmm {k}, zmm, zmm	1	0.5
_mm512_storeu_ps	vmovups m512 {k}, zmm	1	0.5

381

382

383 Profiling Results Analysis

384 To test the efficiency of the generated code, self-contained tracking executables were
 385 developed and profiled in an Intel(R) Core(TM) i9-7900X CPU @ 3.30GHz processor with
 386 support for AVX2 and AVX512. This platform also had installed the Intel Vtune Amplifier to
 387 serve as the profiling tools of the option. The testing/profiling procedure consisted of

- 388 1. Running multiple executables that make calls to the API depending on its category, i.e.,
 389 executables checking for AVX512 performance were only making function calls
 390 supporting the AVX512 function calls.
- 391 2. Dividing the test into two groups according to data type: (1) source files using single
 392 precision float for the correlation operation and (2) source files using 32-bit integers.
- 393 3. Dividing the test pool into three groups depending on the carrier and code generation:
 394 (1) regular code and carrier generation using C standard function calls, (2) nominal
 395 implementation with the DLUT method and (3) AVX implementation with the PLUT
 396 method for code and carrier generation. This last group also subdivides into AVX2 and
 397 AVX512 implementations.

398

Fig. 10 Profiling results for si32 implementation in an Intel Skylake processor

Fig. 10-11 displays the execution times of running the executables fifty times in the targeted Skylake processor. All executables were compiled with the same optimization flags of -O3, guaranteeing the maximum level of optimization possible while keeping a safe code generation. The executables in questions were labeled as

1. reg_standalone: source code created using exclusively C function calls. This should mimic a generic implementation of a tracking algorithm and software correlator chip.
2. avx_si32_avx2: source code created using function calls that use AVX2 extensions and with code and carrier generation using the PLUT method with AVX2 support.
3. nom_si32_avx2: source code created using function calls that use AVX2 extensions for multiplication and accumulation but a nominal code and carrier generation using the DLUT method.
4. avx_si32_avx512: source code created using function calls that use AVX512 extensions and with code and carrier generation using the PLUT method and AVX512 support.
5. nom_si32_avx512: source code created using function calls that use AVX512 extensions for multiplication and accumulation but a nominal code and carrier generation using the DLUT method.

Fig. 11 Profiling results for fl32 implementation in an Intel Skylake processor

Table 6 Average execution time for one millisecond correlation

Implementation	Average Time (s)
reg_standalone	6.581746666667e-05
avx_si32_avx2	1.652866666667e-06
nom_si32_avx2	5.806866666667e-06
avx_fl32_avx2	1.598733333333e-06
nom_fl32_avx2	5.440533333333e-06
avx_si32_avx512	1.488400000000e-06
nom_si32_avx512	5.321000000000e-06
avx_fl32_avx512	1.391866666667e-06
nom_fl32_avx512	4.943400000000e-06

Validation of Tracking Results

Validation of all code created was done by adding the function call from the API to the self-contained tracking executables. Fig. 12 shows the tracking results when running a highly efficient code that makes use of the AVX512 extensions for multiplication, addition, and carrier-code generation. The upper half of the figure displays the navigation data bits of a GPS L1 C/A signal ready to be decoded while the second half displays the correlation results of the operation.

Fig. 12 Tracking results for PRN 22 using a real data set in a test scenario. Results include AVX512 carrier wave generation, AVX512 pseudorandom code generation, and parallelization of multiplication and addition operations. The top figure in the plot shows the decoded data bits in the navigation message while the bottom figure shows the correlation results.

Conclusion

This work provides a state of the art library for GNSS correlations using the latest SIMD extensions to accelerate the processing. It also offers the first of its kind parallelized code and carrier generation based on lookup tables and SIMD extensions. Finally, the library provides an extension to the methods originally presented by Heckler and Garrison (2006) to perform common arithmetic operations (addition, multiplication, etc.) but with an elevated data type such as integers and single precision floats that offers flexibility in the operations supported when targeting popular SDR front ends.

The cycle study of each implementation highlights that SIMD extensions drastically reduced the cycle count required to complete one correlation operation. This study did not consider the pipeline architecture of the targeted platform because of its diversity across CPU lines. However, CPU pipeline will simply further improve the current implementation processing time as the total cycle count will not be a strict addition of the latency cycles, but rather a combination of the latency cycles, throughput delay, and next operation in order.

Data and Source Code

Code developed for the library was done following the model introduced by the VOLK Library (Rondeau et al. 2012), and some of its AVX512 methods (multiplication and addition) follow the model developed by Miralles and Iwamoto (2018). However, methods developed for carrier and code generation have been exclusively designed for this work.

The source code available is organized as follows:

1. src: directory with source code for library functions including multiplication, accumulation, and code and carrier generation.
2. doc: directory with Doxygen style documentation of available functions in HTML format, with links to function parameters and return values if applicable.
3. prof: sample code on how to use the library functions in a simulated, standalone tracking scenario.
4. plot: binary files generated by prof source code to display tracking results values and validate new function additions or usage.
5. build: build directory for application using standard makefile components.
6. install: directory with generated executables from prof source code and utility functions to profile the code
7. data: directory hosting data dependencies to run the executables generated.

Code is available for download/inspection at <https://github.com/dmiralles2009/gnss-intrinsics> and at <https://www.ngs.noaa.gov/gps-toolbox>

Conflict of Interest

The authors declare that they have no conflict of interest.

Acknowledgments

This material is based upon work partially supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE 1144083.

484 **References**

- 485 Fog A (2017) Instruction tables: Lists of instruction latencies, throughputs and micro-operation
486 breakdowns for Intel, AMD and VIA CPUs. Manual 4, 2017.
- 487 Fernández-Prades C, Arribas J, Closas P (2016) Accelerating GNSS Software Receivers. Proc.
488 ION GNSS 2016, Institute of Navigation, Portland, Oregon, USA, September 12-16, 44-61.
- 489 Greene MA (1997) Pentium (R) processor with MMX/sup TM/ technology performance. Proc.
490 IEEE COMPCON 97. <https://doi.org/10.1109/CMPCON.1997.584726>
- 491 Heckler GW, Garrison JL (2006) SIMD correlator library for GNSS software receivers. GPS
492 Solutions, 10(4):269–276.
- 493 Ledvina BM, Psiaki ML, Powell SP, Kintner PM (2014) Bit-wise parallel algorithms for
494 efficient software correlation applied to a GPS software receiver. IEEE Transactions on
495 Wireless Communications, 3(5):1469-1473.
- 496 Matic V, Marinkovic-Nedelicki V, Tadic V (1998) Comparison of digital signal processing
497 methods for sine wave signal generation. SBT/IEEE International Telecommunications
498 Symposium. <https://doi.org/10.1109/ITS.1998.713134>
- 499 Miralles D, Iwamoto J (2018) Accelerating software radios by means of SIMD Instructions. A
500 case for the AVX2 and AVX512 Extensions. Proc. of the GNU Radio Conference,
501 Henderson, Nevada, USA, September 17-21, 1-7.
- 502 Mitola J (1995) The software radio architecture. IEEE Communications Magazine, 33(5):26-
503 38.
- 504 Pany T (2010). Navigation Signal Processing for GNSS Software Receivers, Artech House.
- 505 Principe F, Bacci G, Giannetti F, Luise L (2011) Software-Defined Radio Technologies for
506 GNSS Receivers: A Tutorial Approach to a Simple Design and Implementation.
507 International Journal of Navigation and Observation.
508 <http://dx.doi.org/10.1155/2011/979815>.
- 509 Psiaki M (2001) Block Acquisition of Weak GPS Signals in a Software Receiver. Proc. ION
510 GPS 2001, Institute of Navigation, Salt Lake City, Utah, USA, September 11-14, 2838-2850
- 511 Rondeau T, McCarthy N, O'Shea T (2012) SIMD Programming in GNU Radio: Maintainable
512 and User-Friendly Algorithm Optimization with VOLK. Proc. WinnForum's SDR
513 Conference, Brussels, Belgium, June 27-29, 101–110.
- 514 Rupp K (2018) Forty two years of microprocessor trend data. Available
515 <https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/>. Accessed 1
516 January 2019.
- 517 The Intel Corporation (2017) Intel ® 64 and IA-32 Architectures Software Developer's
518 Manual. Volume 1: Basic Architecture. Tech. Rep. 253665, The Intel Corporation, 2017.
519 <https://doi.org/10.1109/MAHC.2010.22>

520 Xu L, Zhang H, Guo W, Zhang D (2013) A New SIMD Correlator Algorithm for GNSS
521 Software Receivers to Process Complex IF Data. Proc. China Satellite Navigation
522 Conference, Wuhan, China, May 15-17, 565-573

523

524 **Author Biographies**

525 **Damian Miralles** is a graduate student in the Department of Aerospace Engineering Sciences
526 at the University of Colorado Boulder. He received a B.S. in Electrical and Computer
527 Engineering from the Polytechnic University of Puerto Rico. His research interests are in
528 GNSS receiver technologies, SDR and digital signal processing.

529 **Dennis M. Akos** completed the Ph.D. degree in Electrical Engineering at Ohio University
530 within the Avionics Engineering Center. He has since served as a faculty member with Luleå
531 Technical University, Sweden, and then as a researcher with the GPS Laboratory at Stanford
532 University. Currently, he is a faculty member with the Aerospace Engineering Sciences
533 Department at the University of Colorado, Boulder and maintains a visiting appointment at
534 Stanford University.

535