juliacon

Tracking.jl: Accelerating multi-antenna GNSS receivers with CUDA

Can Özmaden[1]

[1]Faculty of Electrical Engineering and Information Technology, Chair of Navigation, RWTH Aachen University

ABSTRACT

The use of advanced Global Navigation Satellite System (GNSS) receiver architectures employing multi-antenna and multi-correlator signal processing poses a high computational demand on Software Defined Radio (SDR) modules. The literature provides promising results in offloading computationally burdensome tasks onto Graphics Processing Units (GPUs). Tracking.jl implements the tracking module of a GNSS receiver, responsible for the majority of the computational load. This paper extends the existing codebase with GPU algorithms. A benchmarking comparison between the implemented GPU algorithms and existing parallelized CPU algorithm is carried out with varying optimization strategy combinations and signal input sizes on two platforms: on a conventional mid-grade PC and an NVIDIA Jetson embedded system. Possible performance improvements of GPU-enabled GNSS receivers are highlighted based on empirical findings from simulated data. The preliminary benchmarking results show real-time signal processing capabilities of the developed algorithms.

Keywords

Julia, GNSS, Signal Processing, SDR, CUDA, GPU

1. Introduction

Utilization of antenna diversity via antenna arrays is a proven technique for GNSS receivers to mitigate the effects of Radio Interference (RFI), jamming, and spoofing [17]. Furthermore, multi-correlation allows GNSS receivers to alleviate multipath errors [25]. It also provides a robust estimation of a transfer function of a multi-antenna RF front end as shown in [18]. This aids the receiver to compensate the cross-talk error between the antenna channels. Moreover, multi-constellation and multi-frequency signal processing are increasingly used Advanced Receiver Autonomous Integrity Monitoring (ARAIM) to meet strict demands of Safety of Life (SoL) applications [4]. Advanced modern receiver architectures combining the aforementioned techniques pose high demands on the computational resources of a software-defined receiver, making real-time processing a challenge. Most of the computational load can be attributed to the tracking module of a GNSS receiver, specifically to the correlation operation, which is the focus of this paper.
The literature provides promising results on dealing with the computational load by utilizing various parallelization approaches. These include the use of bit-wise parallelism [15], single instruction multiple data (SIMD) instructions, and multithreading on multicore central processing units (CPUs) [7]. Additionally, the use of hardware acceleration by utilizing application-specific integrated circuits (ASICs) [1], digital signal processors (DSPs) [28], field-programmable arrays (FPGAs)[8] or GPUs [14, 21, 11, 27, 12].
This paper focuses on GPU acceleration. The use of GPUs to accelerate computationally burdensome tasks has become a popular choice in a plethora of applications, such as digital image processing, radar imaging, cryptography, neural networks, and many more. GPUs consist of massive parallel processors able to efficiently deal with a large amount of input data. In comparison with other hardware accelerators, GPUs are affordable, generally easier to program, and readily available in a large amount of existing consumer electronics devices. This makes GPUs highly desirable for SDRs. A significant amount of effort has been made by the vendors and the community to make programming GPU applications easier. Commonly used programming frameworks are NVIDIA's CUDA, AMD's ROCm, and the open-source OpenCL. In this paper, the CUDA framework is used.
Programmers are often challenged with various ways of optimizing the efficiency of GPU applications, especially their execution time. As mentioned above, real-time processing capability is crucial for software-defined GNSS receivers. This paper strives to highlight performance improvements fullfilling the real-time processing criteria based on empirical findings. In this paper, the comparison is carried out as a series of benchmarking experiments. Algorithms for the comparison are developed in a rapid prototyping fashion in Julia, a high-level high-performance language [3]. GPU programming is performed by utilizing the CUDA.jl package which provides an interface for Julia to wrap low-level functionalities of the CUDA framework without sacrificing performance [2]. The developed algorithms extend Tracking.jl, an existing multi-antenna multi-correlator GPU-enabled GNSS SDR receiver module [22]. The tested optimization strategies rely purely on CUDA and are therefore not exclusive to Julia.

Paper Organization. This paper is organized as follows: in Section 2 the signal model is presented, specifically the multi-correlation operation and the generation of replica signals. In Section 3 the experiment setup is introduced and the data acquisition methodology is described. Section 4 introduces the CUDA programming model and describes the implementation of the GPU algorithms, including a description of the texture memory based code replication technique. Section 5 focuses on the analysis of the obtained experimental data, to draw outcomes and guidelines for the best performing optimization strategies. These outcomes are summarized, and a conclusion is presented in Section 6.

2. Signal Model

Following is an introduction to the notation used throughout this paper. Let k denote the k-th satellite from K total satellites, l the l-th correlator out of L total correlators, m the m-th antenna out of M total antennas, n the n-th sample out of N total samples of the received signal. Further, any analytic time-domain signal is denoted as $\underline{x}(t), t \in \mathbb{R}$, and the respective sampled version as $\underline{x}[n], n \in \mathbb{Z}$. Modeling of the signals and the receiver architecture follows and extends those presented in [5, 24].

Multi-Correlator

The output of a multi-correlator $R_{l,m}^{(k)}$ of the k-th satellite, l-th correlator m-th antenna can be expressed as:

$$\underline{R}_{l,m}^{(k)} = \sum_{n=1}^{N} \underline{r}_{\mathrm{IF}}^{(k)}[n,m] \ \mathrm{conj}(\hat{\underline{s}}_{\mathrm{ca}}^{(k)}[n]) \ \hat{\underline{s}}_{\mathrm{co},l}^{(k)}[n], \tag{1}$$

where r_{IF} denotes the received signal downconverted to an Intermediate Frequency (IF), $\hat{s}_{\mathrm{ca}}$ the carrier replica, $\hat{s}_{\mathrm{co}}$ the code replica, and $\mathrm{conj}(\cdot)$ the complex conjugation operation. The correlation has to be performed over N samples and for each of M antennas, L correlators, and K satellites and is a prime example of the need for parallelization. The multiplication of the received signal with the conjugate of the carrier replica is often referred to as "carrier wipe-off". The despreading of the signal by multiplication with the code replica is often referred to as "code wipe-off".

Received Signal

The received signal downconverted to the IF can be expressed as:

$$\underline{r}_{\mathrm{IF}}^{(k)}(t) = \underline{s}_{\mathrm{IF}}^{(k)}(t) + \eta(t), \tag{2}$$

where

$$\underline{s}_{\mathrm{IF}}^{(k)}(t) = c^{(k)}(t) \ \exp\Big\{ j \left(2\pi \left(f_{\mathrm{IF}} + f_{\mathrm{d}}^{(k)} \right) t + \phi_0^{(k)} \right) \Big\}, \tag{3}$$

is the signal transmitted by the k-th satellite and downconverted to the IF-band, $c^{(k)}$ is the CDMA code of the k-th satellite, f_{IF} denotes the intermediate frequency, $f_{\mathrm{d}}^{(k)}$ the Doppler shift of the k-th satellite relative to the receiver, $\phi_0^{(k)}$ the phase delay, and $\eta(t)$ the additive white Gaussian noise (AWGN) term.

Carrier Replica

The tracking module produces an estimate of the carrier of the received signal. This signal is commonly referred as the carrier replica $\hat{s}_{\mathrm{ca}}$ and can be expressed as:

$$\hat{\underline{s}}_{\mathrm{ca}}^{(k)}[n] = \exp\left\{ j \left(2\pi \frac{f_{\mathrm{IF}} + \hat{f}_{\mathrm{d}}^{(k)}}{f_{\mathrm{s}}} n + \hat{\phi}_0^{(k)} \right) \right\}, \tag{4}$$

where $\hat{f}_{\mathrm{d}}^{(k)}$ is an estimate of the Doppler shift of the k-th satellite relative to the receiver, f_{s} denotes the sampling frequency of the receiver, and $\hat{\phi}_0^{(k)}$ denotes the estimated phase delay in radians.

Code Replica

Alongside the carrier replica generation, the tracking module also generates a code replica signal $\hat{\underline{s}}_{\mathrm{co},l}$ shifted by δ_l amount of chips that correspond to Δ_l amount of samples that are required by the l-th correlator. It can be expressed as follows:

$$\hat{\underline{s}}_{\mathrm{co},l}^{(k)}[n] = c^{(k)} \left[\mathrm{mod}\left(\left\lfloor \frac{f_{\mathrm{c}} + \hat{f}_{\mathrm{c,d}}^{(k)}}{f_{\mathrm{s}}} (n + \Delta_l) + \hat{\tau}^{(k)} \right\rfloor, \mathrm{length}(c^{(k)}) \right) \right] \tag{5}$$

where $\hat{f}_{\mathrm{c,d}}^{(k)}$ is an estimate of the Doppler shift of the code of the k-th satellite relative to the receiver, f_{c} denotes the code frequency, and $\hat{\tau}^{(k)}$ denotes the estimated phase delay in chips. The calculated code phase needs to be an integer value $z \in \mathbb{Z}$, therefore a flooring operation is needed. Furthermore, since the sample shift Δ_l pushes the earliest and latest chips out of bounds, a mod operation is needed to wrap the out-of-bounds code phase according to the length of the CDMA code. These two operations are computationally prohibitive and are to be avoided if possible.

3. Methodology

Testing Environment

The work in this paper is conducted as a series of experiments on two different platforms provided in Table 1. Platform #1 is a mid-grade desktop Personal Computer (PC) equipped with a 6-core Intel Core i5-9600K 3.70 GHz CPU and an NVIDIA GeForce GTX 1050 Ti GPU. Two Operating Systems (OSes) are installed on separate solid-state memory units: Microsoft Windows 10 Home (Build 19042) and Linux (Endeavour OS, rolling release distribution with a Long Time Support (LTS) 5.15.14-1-lts Linux Kernel). Benchmarks were conducted under both Windows and Linux on Platform #1, without any differences found. For Platform #1, Linux benchmarks are used in this paper.
Platform #2 is an NVIDIA Jetson AGX Xavier Development Kit device. It is equipped with a Tegra System-on-a-Chip (SoC) combining a proprietary 8-core 2.27 GHz NVIDIA ARMv8 CPU and an NVIDIA GPU with Volta microarchitecture.
The two platforms cover interesting aspects of possible implementation scenarios of a GNSS SDR receiver. Platform #1 is most similar to commonplace PCs in research institutes or companies, whereas Platform #2 provides an insight into the performance of a mobile embedded GPU-enabled GNSS SDR, commonly used for autonomous applications.

Signal Generation

Signals of differing lengths and properties are generated according to the parameters provided in Table **??**. The sampling frequency is capped at a maximum of 400 MHz. The number of antennas is switched between one and four. The four antenna case is selected to emulate DLR's GALANT receiver with a 2x2 antenna array [8]. The number of correlators is selected to include the classical early-prompt-late correlator, and additionally a correlator with seven taps. In this paper, the focus lies on the optimization of the correlation for one satellite channel. The parallelization over multiple channels can be easily extended as presented in [27]. However, this brings with it the requirement to strictly synchronize the channels for coherent integration over fixed length. This is not explored in the making of this paper.
The generated data signals assume a constant data bit stream of ones for the entire signal duration T. They are subsequently spread by multiplication with the respective CDMA code of the constellation and upconverted to f_{IF}. Differing from Equation 2, no AWGN is added to the signal, as the time complexity of the operations remains the same. The signal generation procedure for the GPU is provided below:

Table 1. : Parameters and their value variation used to execute different benchmarks presented in this paper.

	Platform #1	Platform #2
Name	Desktop PC	NVIDIA Jetson AGX Xavier Development Kit
OS	Windows 10 / Linux	Linux
CPU Name	6-core Intel Core i5-9600K	8-core NVIDIA ARMv8 SoC
CPU Clock frequency	3.70 GHz	2.27 GHz
GPU Name	NVIDIA GeForce GTX 1050 Ti	Tegra Xavier SoC
CUDA Version	v11.5	v10.2
NVIDIA Driver Version	Linux-x86_64 495.46	Linux-AArch64 32.4.4
GPU Micro-architecture	Pascal (2016)	Volta (2018)
GPU Clock frequency	1.39 GHz	1.37 GHz
GPU Number of SMs	6	8
GPU Shared memory	49152 bytes	49152 bytes
GPU Thread block size	1024	1024
GPU Grid size	(2147483647, 65535, 65535)	(2147483647, 65535, 65535)

```
code_phases = get_code_frequency(system) /
sampling_frequency .* (0:num_samples-1) .+
start_code_phase

upsampled_codes = system.codes[
    1 .+ mod.(
        floor.(Int, code_phases),
        get_code_length(system)
    ),
    prn
]

carrier_phases = CuVector{Float32}(2π *
(0:num_samples-1) * carrier_frequency /
sampling_frequency .+ start_carrier_phase)

signal.re[1:num_samples,:]
    = cos.(carrier_phases) .* upsampled_codes *
CUDA.ones(N)

signal.im[1:num_samples,:]
    = sin.(carrier_phases) .* upsampled_codes *
CUDA.ones(N)
```

Depending on the processing unit under test, the received signal is kept in the primary memory of the host, or transferred to the global memory of the GPU. The GPU signal is generated with single precision (from here on aliased as F32). The CPU signal is also initialized with single-precision floating-point values, to maximize SIMD efficiency. The received signal can be additionally broken into a Quadrature (Q) and an In-phase component (I). These are realized as a structure of arrays via StuctArrays.jl holding complex singles (from here on aliased as $\mathbb{C}_{\mathrm{F32}}$). This ensures a coalesced memory access when implementing I/Q-signal processing, which increases the efficiency of the algorithms, discussed in more detail in Section 4.

Algorithm Naming Scheme

A consistent naming scheme for the algorithms under test is used throughout this paper. The name consists of keywords relating to various properties of an algorithm. Naming starts with a number describing the overall procedural sequence presented as a block diagram in Figure 1. The number following that is connected to the parallel reduction algorithm at use is described in detail in Section 4. A string is added to the number of the reduction algorithm, indicating original implementation by the keyword "pure", a complex reduction extension by "cplx" and a "cplx_multi" keyword indicating the total fusion of all multi-antenna multi-correlator kernel invocations. Additionally, if the kernel utilizes the novel texture memory code replica generation described in Section 4 a keyword "textmem" is added. To give an example, an algorithm following the 3rd procedure from Figure 1, utilizing a fully fused 3rd reduction algorithm and texture memory, is designated the codename "1_3_cplx_multi_textmem".

Benchmarking

Measurements of execution times are inherently noisy due to OS scheduling events, clock frequency jitter, etc. Therefore, there is a need for a statistical evaluation of the collected timings, i.e. averaging over numerous executions, performing calculations of estimators.
Runtimes in this paper were collected using the BenchmarkTools.jl package. The package tunes the execution parameters, such as the number of executions, automatically. Four estimators are provided as the benchmark result. These are the *Minimum*, *Median*, *Mean*, and *Maximum* estimators. In [6] the authors of the package asses the minimum estimator, rather than median or mean, to be the most robust one, as execution times are heavily right-skewed. This renders the mean estimator heavily influenced by the outliers, and the maximum estimator being purely an outlier. Therefore the presented execution times are taken from the Minimum estimator unless otherwise noted.

Project Automation

Algorithms under test and benchmarking scripts are all compounded into a Julia module utilizing the DrWatson.jl package [10]. This provides reproducibility of the results found in this paper, as the module keeps a list of all dependencies and their version and can be instantiated via a simple command. Additionally, the algorithms used for the comparison in this paper are tested utilizing the standard Julia testing ecosystem. The repository containing the source code and scripts for the benchmarks acquired and figures found in this paper can be found on Github [29]. Raw data obtained from the benchmarks on the platforms described in this paper is also made available [20].

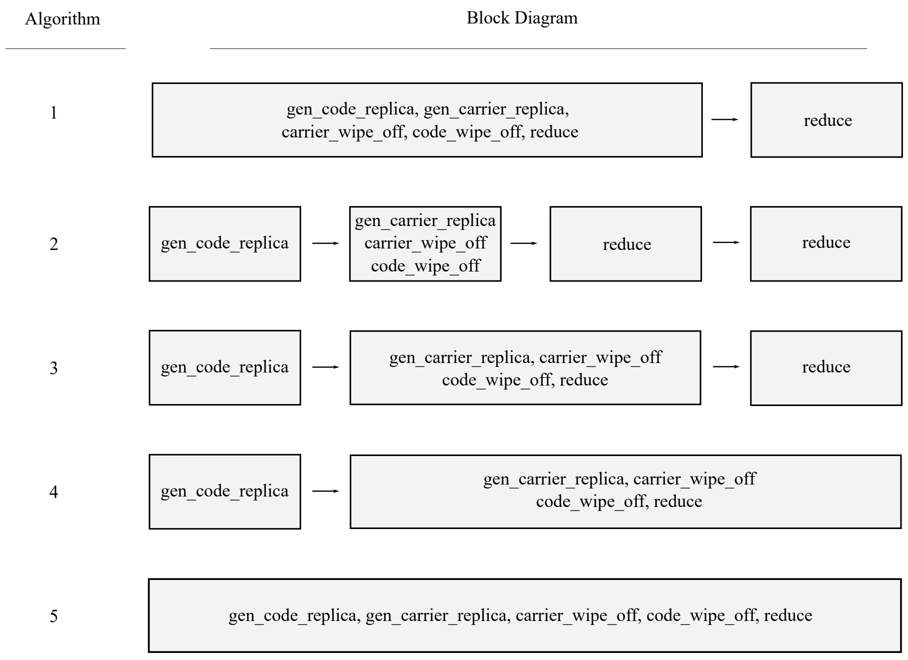
Algorithm
Block Diagram
1
gen_code_replica, gen_carrier_replica, carrier_wipe_off, code_wipe_off, reduce
reduce
2
gen_code_replica
gen_carrier_replica carrier_wipe_off code_wipe_off
reduce
reduce
3
gen_code_replica
gen_carrier_replica, carrier_wipe_off code_wipe_off, reduce
reduce
4
gen_code_replica
gen_carrier_replica, carrier_wipe_off code_wipe_off, reduce
5
gen_code_replica, gen_carrier_replica, carrier_wipe_off, code_wipe_off, reduce

Fig. 1: Block-diagram description of the algorithms being tested in this paper.

4. Algorithm Design

CUDA

CUDA is a programming framework published by NVIDIA for programming the GPU. Currently, it provides first-grade support for C/C++ and FORTRAN. CUDA is designed to ease the programming of parallel applications on the GPU, by providing an LLVM-based compiler for PTX, the NVIDIA GPU native Instruction Set Architecture (ISA). Albeit, effective CUDA programming requires a solid understanding of the low-level resources of the GPU. In the following, a short explanation of the CUDA framework is given. The information stems from NVIDIA' s "CUDA C++ Programming Guide" [19].
Functions executed on the GPU are referred to as *kernels*. Each kernel is launched on a *grid* consisting of *thread-blocks*. The smallest units, *threads*, are issued in a group of 32 to perform the same instruction. This grouping is called a *warp*. The next level of thread grouping is called a *block* or a *thread-block*. These differ in their maximum allowed size depending on the hardware. CUDA provides three-dimensional indexing along the grid, and inside an individual thread-block. The product of the dimensions inside a block is fixed to the maximum number of threads per block, as expressed below

```
max_thread_per_block = threads_x * threads_y *
threads_z
```

Thus, if one were to fix the z and y dimensions of a thread-block, the maximum allowed number of threads in the x dimension can be easily found via:

```
threads_x = max_threads_per_block / threads_y *
threads_z
```

On the grid level, no limits are tieing the x, y, and z dimensions together.
Each thread executes in parallel on the GPU, meaning one of the priorities in parallelization optimization is providing the GPU with enough resources to saturate the number of threads performing calculations in parallel. This is often measured via an *occupancy* metric. Kernels exhibiting high occupancy have the potential to better hide processing latencies, however, they are not per se the best performing as shown in [26]. In line with previous work on GNSS GPU signal processing, this paper strives for optimum occupancy at every kernel launch via a call to a launch configuration subroutine.

GPU Memory

The knowledge of various memory resources of the GPU is essential for improving the efficiency of kernels. The memory available to all threads regardless of grouping is called the *global memory* of the GPU. Inside each thread-block threads can access a *shared memory*. Shared memory is faster than global memory. Additionally, each thread has its local on-chip L2 cache memory for local variables.
Furthermore, there is another type of read-only memory available globally called *texture memory* that possesses some unique properties. Firstly, the speed of accessing the texture memory is increased by caching. Secondly, spatially close memory locations addressed in an unpredictable order get a speed-up. And most importantly for the application in this paper, rules for out-of-bounds accesses and

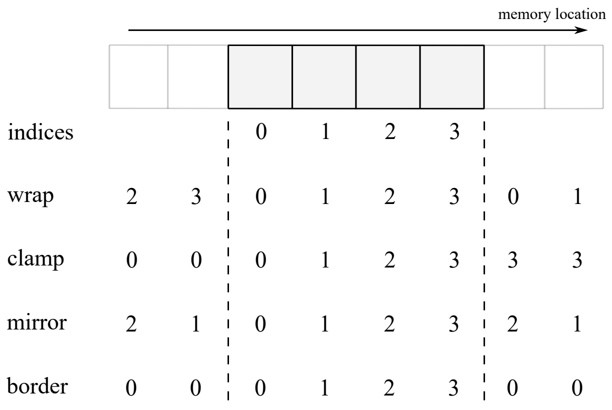

Fig. 2: Visualization of various addressing modes available for the texture memory in CUDA.

an interpolation method for non-integer indices can be defined upon allocation.
CUDA defines three texture memory addressing modes to define behavior upon an out-of-bounds access: *wrap, clamp, mirror* and *border*. Three interpolation methods are provided to deal with non-integer indices: *linear, bilinear* and *nearest neighbour* (CUDA.jl naming). Additionally, a boolean value called *normalized_coordinates* can be passed to specify if the indices are in the $[0, 1)$ range. To make use of the wrap addressing mode, the indices must be normalized. A simple example is provided in Figure 2 to illustrate the functioning of the described addressing modes.

Parallel Reduction

The cornerstone of the tracking module is the correlation process as expressed in Equation 1. This operation requires each element of the vector to be multiplied element-wise with the others, and the result of these products must be summed over N samples. While vector multiplication is trivially parallelizable, the parallelization of the sum requires some thought. Each partial sum of two elements can be computed independently of the other partial sums, however, some communication between threads is needed to facilitate a step-wise reduction of a vector of length N into a scalar value.
In his seminal work, Harris has presented ways to optimize the parallel reduction in CUDA [13]. He introduces seven kernels, each subsequent improving on the preceding. The reduction strategy involves the use of a binary tree-like summing. Harris' reduction kernel numbering is taken one-to-one in the naming of the algorithms in this paper as described in Section 3.
One of the main issues of parallel reduction is the innate need for synchronization between parallel threads. Before CUDA version 9, it was only possible to synchronize threads across a block. This meant that only a vector of length matching the maximum allowed threads per block could have been reduced in a *single-pass* (single kernel invocation). For greater input sizes, a partial sum vector of length equal to the number of blocks has to be allocated. The second kernel invocation reduces the values of partial sums across blocks. This is called *multi-pass* reduction, in contrast to *single pass*. Figure 3 illustrates the multi-pass reduction over 2048 elements by using the 3rd kernel from [13].
With the introduction of *Cooperative Groups* in CUDA version 9, it has become possible to synchronize GPUs at every level, including thread-blocks across a grid. Therefore, a kernel launched within a cooperative group can reduce across a block and subsequently across all blocks in a single-pass.
Another approach to summing a vector on a GPU is via the use of atomic operations. However, atomic operations come at a cost of not utilizing the full bandwidth of the GPU and therefore must be used sparingly. In this paper atomic reduction is used to eliminate the second reduction kernel call in algorithms 4 and 5 as illustrated in Figure 1.

Kernel Implementations

There are various ways to improve the runtime efficiency of a kernel. Firstly, a distinction has to be made if the kernel is a so-called *compute-bound* or a *memory-bound* kernel. For compute-bound kernels, the mathematical operations performed by each thread have to be optimized. For the memory-bound ones, accesses to the memory resources are to be accelerated.
Four operations in the tracking module have the potential to be parallelized. These are the code replica generation from Equation 5, carrier replica generation from Equation 4, downconversion, and correlation from Equation 1. These can be defined as separate kernels or fused in various ways as shown in Figure 1. The downconversion kernel is the only one of the four that can be described as a compute-bound one. The rest are memory-bound kernels and are to be optimized to reach the peak throughput of the GPU.
As discussed previously, the correlation operation involves two stages, firstly a vector element-wise multiplication, and subsequently a summation over all N elements. As an example, a GPS L1 C/A signal of 1 ms duration is considered. Sampled at 2.048 MHz sampling frequency this produces 2048 I/Q samples ($N = 2048$). With an assumption of early-prompt-late correlation ($L = 3$) and four antennas ($M = 4$) this results in 48 reduction kernel invocations for each 1ms chunk. This is firstly due to the maximum amount of threads in a block, which is 1024 for both platforms from Table 1. Therefore, a single-pass reduction with the algorithms from [13] is not possible. Secondly, the reduction has to be performed on both the I and Q components of the signal.
In this paper, kernel calls are reduced by implementing extended versions of the original reduction algorithm. These are the so-called "cplx" and "cplx_multi" kernels. The "cplx" kernel assigns each thread to deal with both in-phase and quadrature components of the signal and has to perform two passes for each $m \in M$ and $l \in L$. Whereas, the "cplx_multi" reduction kernel is totally parallelized and performs two passes only once per $k \in K$.
The CUDA kernel currently implemented in Tracking.jl is a variant of the first algorithm from Figure 1, albeit without the texture memory code generation. Additionally, this is the only kernel that possesses a multi-dimensional grid. The newly devised ones in this paper are all developed to be one-dimensional kernels, assigning more workload to each thread.

Code Replica Generation from Texture Memory

Generation of the code replica is computationally prohibitive due to the modding and flooring operations (see Equation 5). The proposed receiver architecture loads all CDMA codes $c^{(k)}$ of K satellites directly into the texture memory of the GPU in their original length at the beginning of the operation. At the time of writing this paper, the texture memory interface in CUDA.jl remains experimental and subject to change. An example for allocating a texture memory array is provided below. Note the specific memory adressing mode keyword argumenst passed to provide the desired functionality.

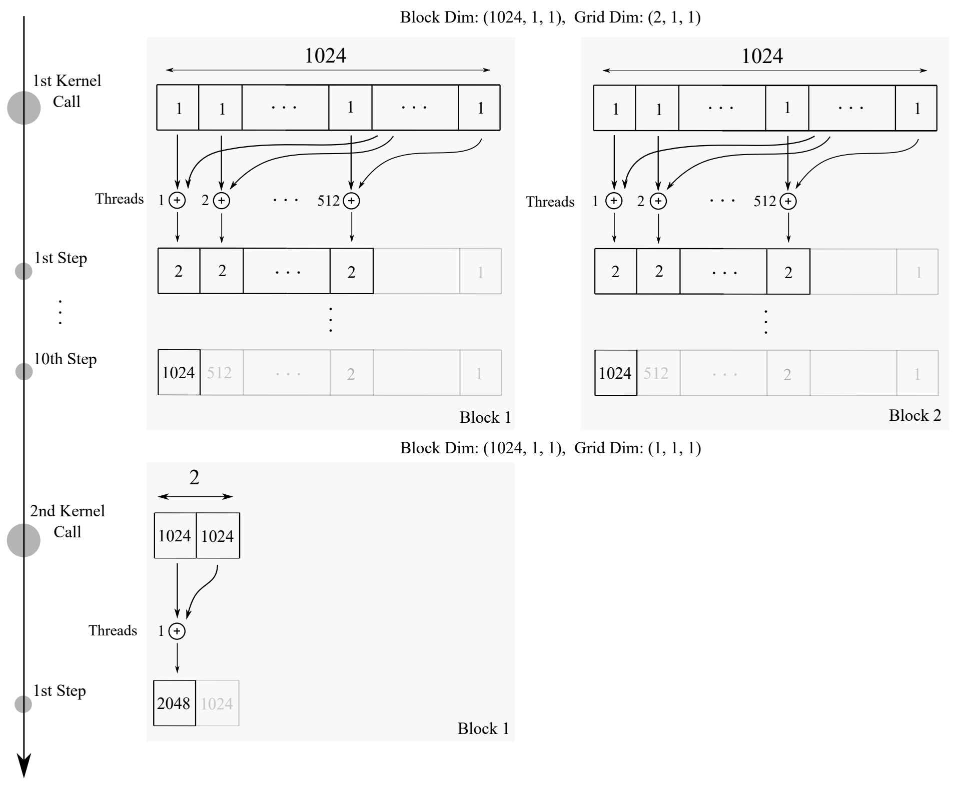
Block Dim: (1024, 1, 1), Grid Dim: (2, 1, 1)
1024
1st Kernel Call
Threads
1st Step
10th Step
Block 1
Block 2
Block Dim: (1024, 1, 1), Grid Dim: (1, 1, 1)
2
2nd Kernel Call
1st Step
2048

Fig. 3: Visualization of a tree-like reduction algorithm from [13], using a coalesced memory access. The kernel is first launched on a grid of two blocks, which produce their respective partial sums for the 2nd kernel launch to sum.

```
codes_text_mem = CuTexture(
    CuTextureArray(codes),
    address_mode = CUDA.ADDRESS_MODE_WRAP,
    interpolation = CUDA.NearestNeighbour(),
    normalized_coordinates = true
)
```

Later, a code replica can be easily generated inside a CUDA kernel as shown below:

```
code_replica[thread_idx] =
codes[(code_frequency/sampling_frequency *
(thread_idx + latest_shift) + start_code_phase) /
code_length, prn)]
```

Note that this provides the advantage of sparing the mod and floor operations, due to the selected memory addressing modes earlier.
Authors of [23] have recently introduced a texture-memory-based algorithm for BeiDou signal generation by loading codes into the texture memory. However, the specific addressing and filtering modes used are not mentioned. The speed-up is therefore only achieved via caching. Authors of [16] implement the code and carrier replica generation via tables stored in texture memory. The use of addressing modes is mentioned but the algorithm is not presented.
In this paper, the technique of code replica generation from texture memory is specifically applied to a multi-antenna GNSS receiver. Additionally, global memory and texture memory are used in cooperation, as the mutable code replica array is contained in the global memory of the GPU, initiated originally from the chip-length codes in the texture memory.

SIMD

The existing codebase of Tracking.jl uses SIMD accelerated CPU correlation. This is perfomed by utilizing the LoopVecorization.jl [1] Package. It provides an selection of user friendly macros that can be prefixed to a loop as shown below on a concrete example of the correlation operation:

```
@avx for i = start_sample:num_samples +
start_sample - 1
    for j = 1:length(a_re)
        sample_shift = correlator_sample_shifts[j] -
correlator_sample_shifts[1]
        a_re[j] += d_re[i] * code[i +
sample_shift]
        a_im[j] += d_im[i] * code[i +
sample_shift]
    end
end
```

[1]https://github.com/JuliaSIMD/LoopVectorization.jl

Algorithm 1 Complex multi reduction kernel extended from Harris #3

```
 1: procedure REDUCE_CPLX_MULTI!(output, input)
 2:     tid ← threadIdx.x
 3:     iq_offset ← blockDim.x
 4:     shmem := DynamicSharedMemory ∈ F32 $^{(2N)\times M\times L}$

 5:     # Load input into shared memory
 6:     $n$ = blockIdx.x × blockDim.x + threadIdx.x
 7:     for all $m \in M,\ l \in L$ do
 8:         if $n \leq$ length(input) then
 9:             shmem[tid, $m, l$] ← input.re[$n, m, l$]
10:             shmem[tid+iq_offset, $m, l$] ← input.im[$n, m, l$]
11:         end if
12:         sync_threads
13:         # Perform tree-like reduction in shared memory
14:         for $s$ = blockDim.x ÷ 2, $s \neq 0$, $s = s \div 2$ do
15:             if tid − 1 < $s$ then
16:                 shmem[tid, $m, l$] ← shmem[tid + $s, m, l$]
17:                 shmem[tid+iq_offset, $m, l$] ← shmem[tid+ $s$ + iq_offset, $m, l$]
18:                 sync_threads
19:             end if
20:         end for

21:         # First thread holds the result
22:         if tid == 1 then
23:             output.re[blockIdx.x, $m, l$] ← shmem[1, $m, l$]
24:             output.im[blockIdx.x, $m, l$] ← shmem[1 + iq_offset, $m, l$]
25:         end if
26:     end for
27: end procedure
```

Table 2. : Parameters for testing the reduction algorithms.

Parameter Name	Values
Number of elements, N	2048 - 32768
Number of antennas, M	1, 4
Number of correlators, L	3, 7

5. Experiment Evaluation

Reduction

An evaluation of the "cplx_multi" reduction introduced in Section 4 under Algorithm 1 is carried out. Parameters used in the experiment to assess the developed algorithm are provided in Table 2. The results are provided in Figure 4. One can observe "cplx_multi" outperforming "cplx" and "pure", due to the fusion of the kernel invocations. Therefore only "cplx_multi" kernels are presented in this paper.

Texture memory

The generation of the code replica from the texture memory of the GPU requires the use of the nearest neighbor addressing filtering and wrap addressing mode. This, however, differs in its implementation from the modding and flooring of code phases introduced in Equation 5. The discrepancy results in a slightly different

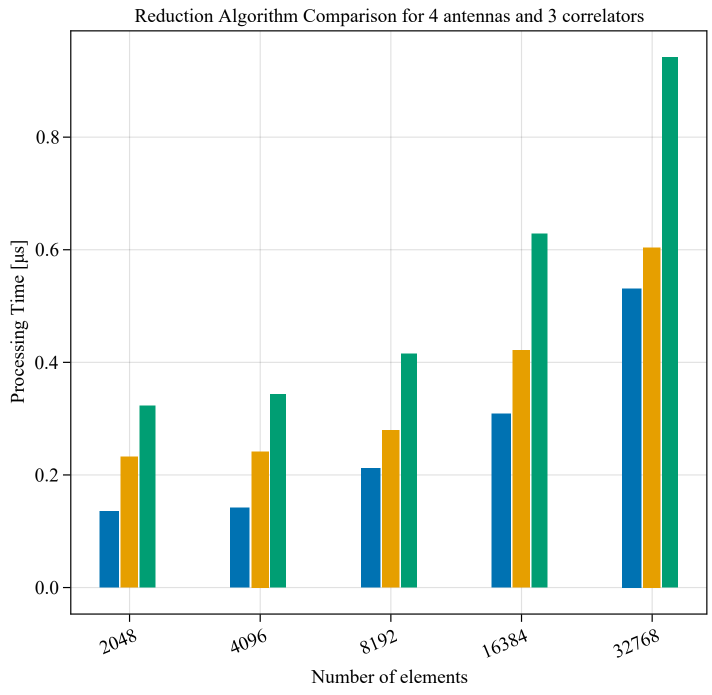

Fig. 4: Runtime analysis of the three reduction algorithms: "pure", "cplx", and "cplx_multi"

Table 3. : Statistical analysis of the relative code phase error between the global memory and texture memory code replication algorithms.

Minimum relative error	0%
Mean relative error	0.03%
Median relative error	0.02%
Maximum relative error	3.17%

code phase being calculated by the algorithms generating the code replica from global memory and texture memory. An analysis of the introduced error by this algorithm is carried out. The results can be seen in Figure 5, along with a timing experiment against the global memory counterpart. As one can observe, the discrepancy results in a negligible relative code phase error, and the speed-up is significant. Short statistical summary is found in Table 3. It can be concluded that the speed-up of using the code replication via texture memory is a well-founded trade-off.

Algorithms

Due to the multitude of various kernel combinations, only a few selected ones are presented in this paper. The experiment setup and related functions exist in the repository [29]. For the sake of brevity, only kernels with the 4th reduction algorithm are shown in this paper. They are decently performing and work effortlessly on non-power-of-two input sizes.
Upon close analysis of Figures 6 and 7, one can conclude the results suggest that differences between the architectures of the GPUs can not be ignored. The algorithms performing best on Platform #1 are the worst-performers for Platform #2, and vice-versa. The results also suggest merit in exploring parallelization on the CPU, as the CPU SIMD algorithms from [22] are found to consistently outperform the GPU on Platform #1. This is, however, not always the case on Platform #2. A possible explanation for this is

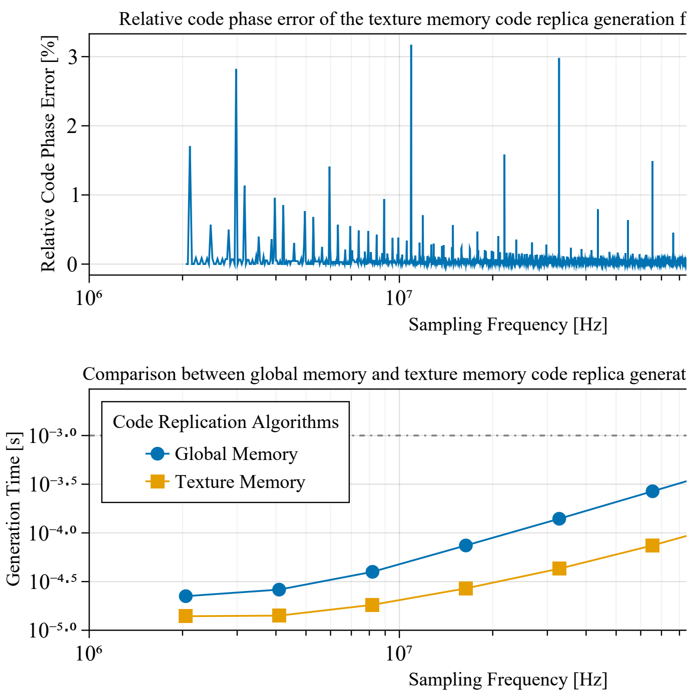

Fig. 5: Above: Code phase error due to the discrepancy between global memory and texture memory code replication algorithms. Below: runtime analysis between the global and texture memory code replication algorithms. The dashed line indicates the real-time bound.

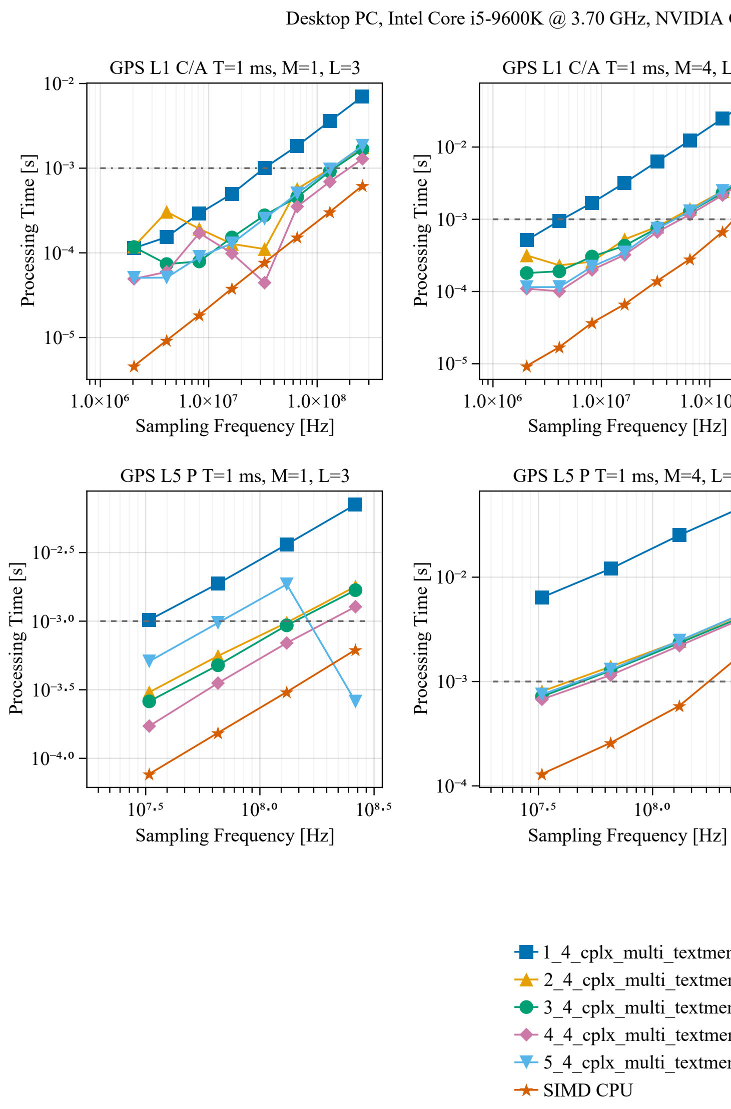

Fig. 6: Processing time of correlation of a 1ms GPS signal on Platform #1. The dashed line indicates the real-time processing bound.

the fact that Platform #2 possesses an ARM chip, with SIMD capabilities unmatched to the Desktop-grade Intel Core i5-9600K. Another factor is also the higher CPU clock frequency of Platform #1. The relative mismatch between the power of the GPU and the power of the CPU on Platform #2, leaning towards the GPU, also plays a role. The CUDA algorithm currently implemented in [22] is being outperformed by the algorithms devised in this paper. Overall it can be said that the 4th algorithm under the name "4_4_cplx_multi_textmem" is the best performing for most of the time on Platform #1. Whereas algorithm 2 under the name "2_4_cplx_multi_textmem" performs the best on Platform #2, for cases utilizing under seven correlators. This is somewhat surprising, as it suggests launch configurations of individual kernels playing a bigger role on the Jetson device, and/or that Platform #2 has shorter API overhead. Moreover, this could be attributed to the somewhat slower atomic operations on Platform #2, as these are utilized in algorithms 4 and 5.
The results suggest real-time operation capabilities of the developed algorithms, even under introduced multi-antenna and multi-correlator load. Most algorithms can be executed in real-time under 20 MHz sampling frequency both on Platform #1 and #2. Some algorithms reach around 40 MHz processing capability with 4 antennas or even 100 MHz with a single antenna and an early-prompt-late correlator. Only the case of processing GPS L5 I5 signals with 4 antennas and 7 correlators remains an unsolved challenge.

6. Conclusion

A series of benchmarks was carried out across different GPU algorithms with varying signal input sizes, ranging in the sampling frequency, number of antennas, and number of correlators from the GPS constellation at two frequency bands and respective codes: GPS L1 C/A, GPS L5 I5. The GPU algorithms differ in their implementation of the parallel reduction, use of texture memory for code replication, and the fusion of subsequent kernels. Benchmarking measurements are collected via a statistical benchmarking package, accounting for the differences in timings due to system noise.
The algorithms are implemented in Julia, an open-source, dynamic, just-in-time compiled, high-performance high-level language. GPU kernel functions utilize the CUDA.jl package, which provides an interface for Julia to wrap functionalities of CUDA C at no performance cost. The correctness of the results calculated by the algorithms is verified via the testing ecosystem of Julia. The outcomes of this paper, however, are not limited to the Julia language, as these rely purely on CUDA.

A comparison between the algorithms has shown differing results for the two platforms under test. For the Desktop PC, an implementation with the code replication kernel separate from a fused carrier wipe-off and single-pass reduction kernel shows the best performance for lower sampling frequencies of the receiver (referred to as "4_4_cplx_multi_textmem" in this paper). With the increasing sampling frequency, however, the difference between the tested algorithms diminishes. The SIMD-based CPU algorithms outperform the GPU algorithms on a Desktop PC under test (referred to as Platform #1), albeit fall behind on the NVIDIA Jetson device (referred to as Platform #2). The fully separate kernel algorithm performed the best on the NVIDIA Jetson device (referred to as "2_4_cplx_multi_textmem"). All the developed algorithms show real-time capabilities under the assumption of 4 antennas and 20 MHz sampling frequency, with some algorithms reaching the 100 MHz mark for single antenna operation.
A repository containing the experiment setup and the source code of the algorithms is made available on Github [29]. Raw data obtained from the experiments on the two platforms described in this paper is also made available under a Creative Commons license [20].
To the best of the authors' knowledge, this is the first publication describing the code replication algorithm on the GPU utilizing both texture memory and global memory. Additionally, it is the first to utilize the Julia programming language to implement CUDA GNSS signal processing algorithms.

Acknowledgements

I would like to thank Sören Schönbrod for his contributions to the Tracking.jl package [22] and for his input and guidance in preparing this paper. Additionally, I would like to thank Tim Besard, creator and co-maintainer of CUDA.jl, for his readiness to answer questions and offer advice. The data for this paper has been gathered utilizing DrWatson.jl, and visualized by using Makie.jl [10, 9]. The NVIDIA Jetson Platform has been provided by the Chair of Navigation within the project ROSANNA of the German Aerospace Center (DLR) on behalf of the German Federal Ministry for Economic Affairs and Energy under grant number: 5O NA 1901. This support is greatly acknowledged.

7. References

[1] Nikola Basta, Achim Dreher, Stefano Caizzone, Matteo Sgammini, Felix Antreich, Götz Kappen, Safwat Irteza, Ralf Stephan, Matthias A Hein, Eric Schäfer, et al. System concept of a compact multi-antenna GNSS receiver. In *2012 The 7th German Microwave Conference*, pages 1–4. IEEE, 2012.

[2] Tim Besard, Christophe Foket, and Bjorn De Sutter. Effective Extensible Programming: Unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 2018. doi:10.1109/TPDS.2018.2872064. 1712.03112.

[3] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017. doi:10.1137/141000671.

[4] Juan Blanch, Todd Walter, Per Enge, Stefan Wallner, Francisco Amarillo Fernandez, Riccardo Dellago, Rigas Ioannides, Ignacio Fernandez Hernandez, Boubeker Belabbas, Alexandru Spletter, and Markus Rippl. Critical Elements for a Multi-Constellation Advanced RAIM. *NAVIGATION*, 60(1):53–69, 2013. doi:https://doi.org/10.1002/navi.29. https://onlinelibrary.wiley.com/doi/pdf/10.1002/navi.29.

[5] Kai Borre, Dennis M Akos, Nicolaj Bertelsen, Peter Rinder, and Søren Holdt Jensen. *A software-defined GPS and Galileo receiver: a single-frequency approach*. Springer Science & Business Media, 2007.

[6] Jiahao Chen and Jarrett Revels. Robust benchmarking in noisy environments. *CoRR*, abs/1608.04295, 2016. 1608.04295.

[7] Yu-Hsuan Chen, Jyh-Ching Juang, Jiwon Seo, Sherman Lo, Dennis M. Akos, David S. De Lorenzo, and Per Enge. Design and Implementation of Real-Time Software Radio for Anti-Interference GPS/WAAS Sensors. *Sensors*, 12(10):13417–13440, 2012. doi:10.3390/s121013417.

[8] Manuel Cuntz, Lukasz Greda, Marcos Heckler, Andriy Konovaltsev, Michael Meurer, and Lothar Kurz. "GALANT - Architecture of a Real-Time Safety of Life Receiver". In *ION GNSS 2009*, September 2009.

[9] Simon Danisch and Julius Krumbiegel. Makie.jl: Flexible high-performance data visualization for Julia. *Journal of Open Source Software*, 6(65):3349, 2021. doi:10.21105/joss.03349.

[10] George Datseris, Jonas Isensee, Sebastian Pech, and Tamás Gál. DrWatson: the perfect sidekick for your scientific inquiries. *Journal of Open Source Software*, 5(54):2673, 2020. doi:10.21105/joss.02673.

[11] Carles Fernández-Prades, Javier Arribas, and Pau Closas. Accelerating GNSS software receivers. In *Proceedings of the 29th International Technical Meeting of The Satellite Division of the Institute of Navigation (ION GNSS+ 2016)*, pages 44–61, 2016.

[12] Chengjun Guo, Bingyan Xu, and Zhong Tian. Research on multi-constellation GNSS compatible acquisition strategy based on GPU high-performance operation. *EURASIP Journal on Wireless Communications and Networking*, 2018(1):1–10, 2018.

[13] Mark Harris et al. Optimizing parallel reduction in CUDA. *Nvidia developer technology*, 2(4):70, 2007.

[14] Kamran Karimi, Aleks G Pamir, and M Haris Afzal. Accelerating a cloud-based software GNSS receiver. *International Journal of Grid and High Performance Computing (IJGHPC)*, 6(3):17–33, 2014.

[15] B.M. Ledvina, M.L. Psiaki, S.P. Powell, and P.M. Kintner. Bit-wise parallel algorithms for efficient software correlation applied to a GPS software receiver. *IEEE Transactions on Wireless Communications*, 3(5):1469–1473, 2004. doi:10.1109/TWC.2004.833467.

[16] Qiushi Li, Zheng Yao, Hong Li, and Mingquan Lu. A CUDA-based real-time software GNSS IF signal simulator. In *China Satellite Navigation Conference (CSNC) 2012 Proceedings*, pages 359–369. Springer, 2012.

[17] D-J Moelker, Edwin van der Pol, and Yeheskel Bar-Ness. Adaptive antenna arrays for interference cancellation in GPS and GLONASS receivers. In *Proceedings of Position, Location and Navigation Symposium-PLANS'96*, pages 191–198. IEEE, 1996.

[18] Michael Niestroj, Marius Brachvogel, Soeren Zorn, and M Meurer. Estimation of antenna array manifolds based on sparse measurements. In *Proceedings of the 31st International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS+ 2018)*, pages 4004–4011, 2018.

[19] NVIDIA. CUDA C++ Programming Guide v11.6.0, January 2022.

[20] Can Özmaden. GPUAcceleratedTracking Raw Data, January 2022. doi:10.5281/zenodo.5933726.

[21] Kwi Woo Park, Sangwoo Lee, Min Joon Lee, Sunwoo Kim, and Chansik Park. An accelerated signal tracking module using a heterogeneous multi-GPU platform for real-time GNSS software receiver. In *2015 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pages 1412–1416. IEEE, 2015.

[22] Soeren Schoenbrod, Michael Niestroj, Erik Deinzer, Can Ozmaden, and Kristoffer Carlsson. JuliaGNSS/Tracking.jl: v0.14.10, January 2022. doi:10.5281/zenodo.5870363.

[23] Pengliang Shi, Shunxiao Wu, Tian Zeng, and Rui Xue. Satellite Navigation Simulation Signal Generation Method Based on GPU Acceleration. *Journal of Mathematics and Informatics*, 16:20, March 2021.

[24] Peter JG Teunissen and Oliver Montenbruck. *Springer handbook of global navigation satellite systems*, volume 1. Springer, 2017.

[25] R.D.J. van Nee. The Multipath Estimating Delay Lock Loop. In *IEEE Second International Symposium on Spread Spectrum Techniques and Applications*, pages 39–42, 1992. doi:10.1109/ISSSTA.1992.665623.

[26] Vasily Volkov. Better performance at lower occupancy. In *Proceedings of the GPU technology conference, GTC*, volume 10, page 16. San Jose, CA, 2010.

[27] Liangchun Xu, Nesreen I Ziedan, Xiaoji Niu, and Wenfei Guo. Correlation acceleration in GNSS software receivers using a CUDA-enabled GPU. *GPS solutions*, 21(1):225–236, 2017.

[28] Qingxi Zeng, Qing Wang, Shuguo Pan, and Chuanjun Li. A GPS L1 Software Receiver Implementation on a DSP Platform. In *2008 First International Conference on Intelligent Networks and Intelligent Systems*, pages 612–615, 2008. doi:10.1109/ICINIS.2008.99.

[29] Can Özmaden. ozmaden/GPUAcceleratedTracking: v1.0, January 2022. doi:10.5281/zenodo.5933659.

Appendix

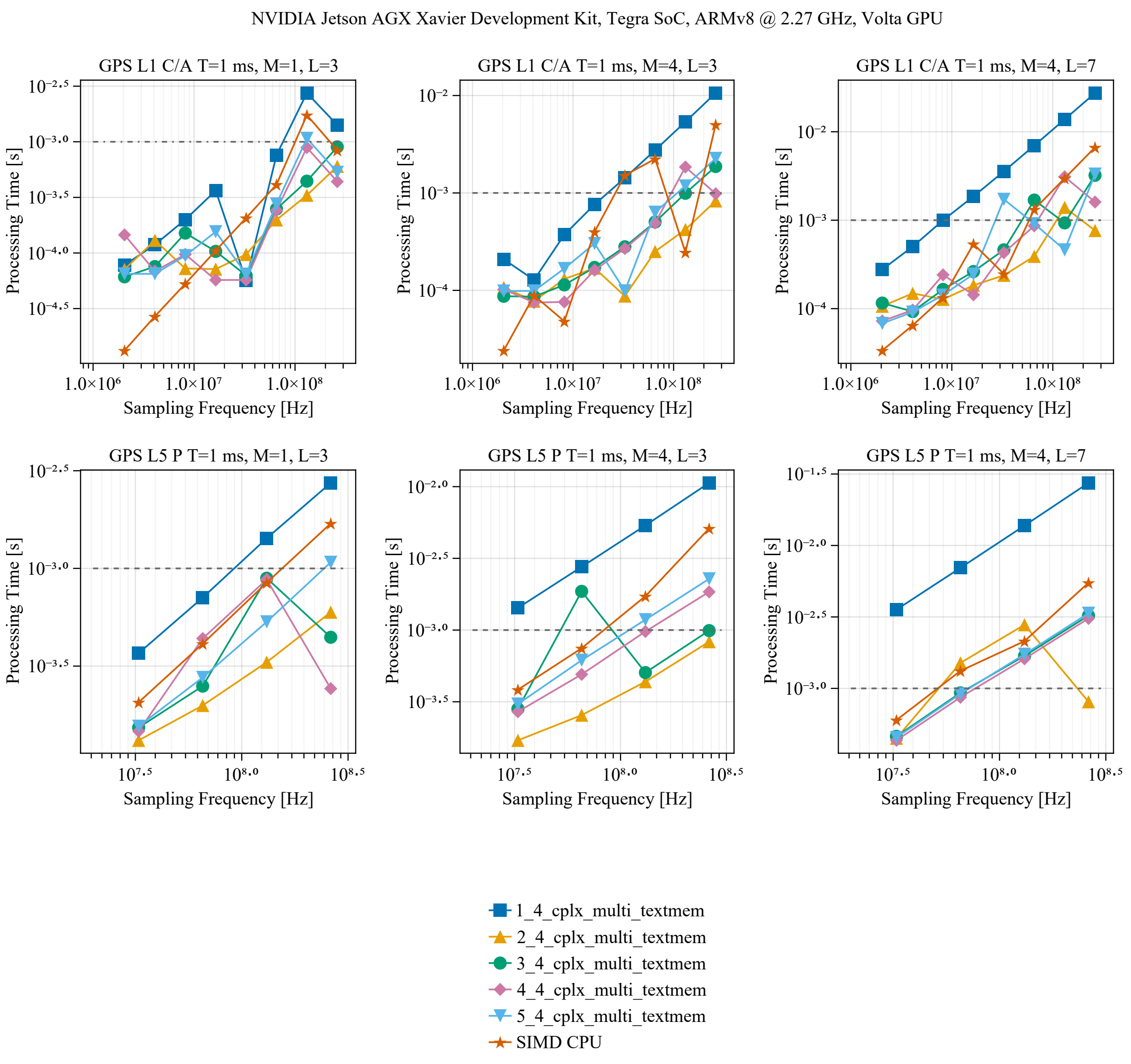

Fig. 7: Processing time of correlation of a 1ms GPS signal on Platform #2.
The dashed line indicates the real-time processing bound.