GPS Solut (2017) 21:225–236
DOI 10.1007/s10291-016-0516-2

ORIGINAL ARTICLE

Correlation acceleration in GNSS software receivers using a CUDA-enabled GPU

Liangchun Xu[1] · Nesreen I. Ziedan[2] · Xiaoji Niu[1] · Wenfei Guo[1]

Received: 23 January 2015 / Accepted: 13 January 2016 / Published online: 2 February 2016

Abstract The correlation process in a GNSS receiver tracking module can be computationally prohibitive if it is executed on a central processing unit (CPU) using single-instruction single-data algorithms. An efficient replacement for a CPU is a graphics processing unit (GPU). A GPU is composed of massive parallel processors with high floating point performance and memory bandwidth. It can be used to accelerate the burdensome correlation process in GNSS software receivers. We propose a novel GPU-based correlator architecture for GNSS software receivers, which is independent of the GPU device, the number of the processing channels, the signal type, and the correlation time. The proposed architecture is implemented and optimized using CUDA, a parallel computing platform and programming model for GPUs. We focus on the following aspects: the design and the time complexity analysis of the proposed GPU-based correlator algorithm, the tests that verify the correctness and the optimization of the implementation, and the performance evaluation of the optimized GPU-based correlator. Moreover, we introduce some new CUDA features that can be applied in a GPU-based correlator.

Keywords Global navigation satellite system (GNSS) · Software receivers · Correlation · CUDA · Real time · Graphics processing unit (GPU)

✉ Wenfei Guo
wf.guo@whu.edu.cn

[1] GNSS Research Center, Wuhan University, No. 129, Luoyu Road, Wuhan 430079, Hubei, People's Republic of China

[2] Computer and Systems Engineering Department, Faculty of Engineering, Zagazig University, Zagazig, Egypt

Introduction

The correlation process, which is executed in a GNSS tracking module and includes carrier and code wipe-off, is the most burdensome operation in a GNSS software receiver. This is due to the high sampling rate and the multiple channel architecture. The computational load of the correlation process amounts to about 96 % of the total computation during the execution time of a GNSS software receiver on a CPU (Chen et al. 2012). In order to operate a GNSS software receiver in real time, the correlation process in GNSS software receivers should be accelerated. Single-instruction multiple-data (SIMD) instructions (Heckler and Garrison 2006) and multi-threaded programming (Chen et al. 2012) running on a multi-core CPU are two conventional methods employed to accelerate the correlation process. Limited by the computational power of a CPU, however, such methods will not always meet the computational demand for future multi-constellation and multi-frequency signal processing in real time.

Compared with a CPU, a GPU is composed of massive cores with tremendous computational power and high memory bandwidth. The general-purpose computing on GPU (GPGPU) provides an approach to accelerate scientific applications, including the correlation process in GNSS software receivers. Earlier researchers working on GPGPU had to map scientific calculations onto graphics APIs (Wilt 2013). The emergence of Brook (Buck et al. 2004) and its successor CUDA have significantly simplified GPU programming and enhanced the computing power of a GPU by offering a high-level language interface to handle the low-level arithmetic and memory resources of a GPU. Implementing the correlation process in GNSS software receivers via CUDA has been carried out by many researchers (Hobiger et al. 2010; Pany et al. 2010a, b; Seo

et al. 2011; Karimi et al. 2013; Roule et al. 2013; Huang et al. 2013; Park et al. 2014).

We propose a GPU-based correlator architecture design for GNSS software receivers. Compared with previous designs, our algorithm is independent of the GPU device, the number of the tracking channels, the signal type, and the correlation time. The proposed architecture is implemented using an NVIDIA GeForce GTX 750 Ti GPU, which can maintain more than 300 tracking channels with a 19.5 MHz sampling rate (9.75 MHz for both in-phase and quadrature-phase samples) and a 4-bit resolution. Moreover, we present an optimization process on how to achieve the maximum performance of the proposed algorithm on a GPU. We also conduct theoretical and experimental comparisons between the GPU-based and the corresponding CPU-based correlation algorithms. Finally, we introduce some advanced features in the latest version of CUDA Toolkit that can be applied in the proposed algorithm.

The signal models and the principle of the GPU-based correlation are presented first. The CUDA programming model in use is then introduced. The proposed parallel algorithm for the GPU-based correlator and its time complexity analysis are illustrated in detail in the following two sections. Two subsequent sections are devoted to the verification and optimization of the code implementation of the proposed algorithm. Following that, an overall performance evaluation on the GPU-based correlator is conducted. Finally, conclusions are presented.

Signal models and correlation

The prompt correlated signal $S_{\mathrm{IQ,p}}$ can be found from

$$S_{\mathrm{IQ,p}} = \sum_{n=0}^{N-1} S_{\mathrm{IF}}(n) R^{*}_{\mathrm{carr}}(n) C_{\mathrm{p}}(n) \tag{1}$$

where $S_{\mathrm{IF}}(n)$ is the received IF signal, $R_{\mathrm{carr}}(n)$ is the local carrier replica signal, and $C_{\mathrm{p}}(n)$ is the local prompt code replica signal. The early correlated signal $S_{\mathrm{IQ,e}}$ and the late correlated signal $S_{\mathrm{IQ,l}}$ can be calculated similarly as the prompt one.

In order to achieve parallel correlation, arrays of the received complex signal and the local replica signals should be prepared before the correlation process. In case of tracking, the correlation process is repeated with every tracking update. The local carrier signal step $\Delta\theta$ and the local code signal step Δc are adjusted based on the tracking output. The number of correlation samples N, the initial carrier phase θ_0, and the initial code offset c_0 are related to the last correlation. Assuming that the correlation time is one code duration, the quantities N, θ_0, and c_0 in the next correlation can be calculated as

$$N(m) = (c_{\mathrm{len}} - c_0(m-1))\Delta c(m-1) f_{\mathrm{s}} \quad (m = 1, 2, \ldots) \tag{2}$$

$$\theta_0(m) = \theta_0(m-1) + N(m)\Delta\theta(m) \tag{3}$$

$$c_0(m) = c_0(m-1) + N(m)\Delta c(m) - c_{\mathrm{len}} \tag{4}$$

where m is the index of one correlation in time domain, f_{s} is the sampling rate, and c_{len} is the code length. In Eq. (2), $N(m)$ is floored into an integer before the actual calculation.

CUDA programming model

CUDA is a parallel platform invented by NVIDIA for C/C++/Fortran programming on the GPU (Wilt 2013). CUDA C is employed here to design and realize a GPU-based correlator. In order to utilize the computational power of the multiprocessors in GPU, kernels (functions) that are executed by each individual CUDA thread should be defined as C functions using the keyword "__global__." The thread and memory hierarchy of a GPU can be summarized as follows: Each of the 32 threads is coupled into a warp so that they can execute one common instruction with different data at a time. Branch divergence, which means executing different instructions in threads of a warp, will significantly decrease efficiency, and thus, it should be avoided. Warps are grouped into blocks, where threads of a block are executed concurrently on one multi-threaded streaming multiprocessor (SM). Blocks are further grouped into a grid, where they are executed concurrently on multiple SMs in a GPU.

CUDA provides different types of memories in the GPU to maximize the performance. Global memory, shared memory, local memory, and registers are used in the proposed GPU-based correlator architecture design. They are allocated for grid, block, and thread. Figure 1 illustrates the thread and memory hierarchy in detail.

The bandwidth of different kinds of GPU memories varies. The registers provide the fastest memory transfer with nearly 8 TB/s bandwidth. The shared memory and global memory follow the registers in terms of the memory transfer speed. The local memory is an abstract memory

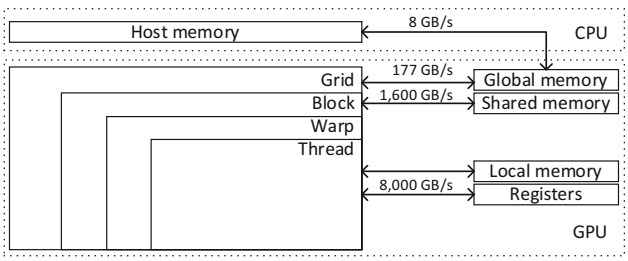


Fig. 1 Thread and memory hierarchy in a GPU

type to hold spilled registers (Farber 2011). Register spilling occurs when a block requires more registers on an SM than the available ones. GPUs with CUDA 2.0 or higher capabilities spill registers to L1 cache, which is physically integrated with the shared memory. Older GPUs spill registers directly to the global memory. Therefore, the local memory bandwidth is dependent on the device. The slowest memory transfer is between the host memory and the device memory with about 8 GB/s bandwidth. This bandwidth value is limited by the peripheral component interconnect express (PCIe) connector between the CPU memory and the GPU memory.

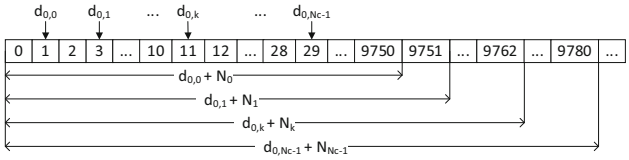


Fig. 2 Data buffer storing IF samples in the GPU

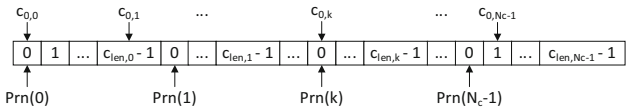


Fig. 3 Code table in the GPU

Algorithm design

The correlation process can be parallelized on both the tracking channel level and the sample level. Considering the aforementioned CUDA programming model, each tracking channel can be mapped into one block, and each signal sample can be mapped into one thread. The integration after carrier and code wipe-off can be regarded as a block reduction, which means calculating the sum of the variables in threads of a block.

The carrier and code wipe-off process for each sample can be put into one thread directly. The key strategy to design a parallel algorithm for carrier and code wipe-off lies on the data structure chosen for the input data and how the input data are mapped into each thread.

The input of the kernel correlation function in the proposed GPU structure consists the following: a buffer that holds the samples of the received IF signal, a carrier table for generating carrier replica signal, an array composed of the pseudorandom noise (PRN) codes of all tracked satellites, and an array of structures that stores the parameters of the tracking channels. A channel structure includes the initial IF sample offset in data buffer, $d_{0,k}$, the numbers of samples in one correlation, N_k, the initial carrier phase offset in carrier table, $\theta_{0,k}$, the carrier step, $\Delta\theta_k$, the initial code offset in code table, $c_{0,k}$, and the code step, Δc_k, for a tracking channel, k. The total number of tracking channels, N_c, is the final input parameter. The output results of the correlation $S_{\mathrm{IQ,e}}$, $S_{\mathrm{IQ,p}}$, $S_{\mathrm{IQ,l}}$ are also put into the channel structure to have coalesced memory access.

In a CPU-based correlation process, there is a data buffer storing the received signal, which is shared among different tracking channels. As for a GPU-based correlation process, the data buffer is transferred to the global memory of a GPU, as shown in Fig. 2, where $d_{0,k}$ is the pointer to the initial data offset in the data buffer and N_k is the number of samples in one correlation, in a channel, k. The array of PRN codes in the GPU-based correlation stores the PRN codes for each satellite sequentially with a one-dimensional size of $\sum_{k=0}^{N_c-1} c_{\mathrm{len},k}$ as shown in Fig. 3, where $c_{0,k}$ is the pointer to the initial PRN code offset in the code table and $c_{\mathrm{len},k}$ is the length of the PRN code, Prn(k), in a channel, k. The carrier table is a common resource for different tracking channels, and so, it is usually initialized and copied into the global memory of the GPU in advance.

After the data structure is chosen, the next step is to map the data into the blocks of threads in GPU. In CUDA programming, threads and blocks are marked by the thread index, threadIdx, and the block index, blockIdx, which are limited by the grid size, gridDim, and the block size, blockDim, respectively. The variables that hold the indexes and sizes are three-dimensional variables. However, in the proposed design of the GPU-based parallel correlation, only one dimension is in use. The following code snippet presents a more clear view on the data mapping:

```
for (int ich = blockIdx.x; ich < nch; ich += gridDim.x) {…
    for (int pos = sampleBase + threadIdx.x;
  pos < sampleEnd; pos += blockDim.x)
    {…}}
```

The variable ich is used to index the tracking channel, and the variable pos is used to index the IF samples in one channel. The two variables, sampleBase and sampleEnd, are used to indicate the bound $[d_{0,k}, d_{0,k} + N_k]$ of the samples that are used in one correlation in a tracking channel k.

The correlation needs six reductions in each block to generate the early, prompt, and late results for both in-phase and quadrature-phase data. Shared memory is used for the final reduction (Harris 2007). Limited by the total amount of shared memory per block in our device, which, for example, is 49,152 bytes (Table 2), and the size of float type, which is 4 bytes, then the size of one reduction array is limited to be no greater than $49{,}152/(6 \times 4) = 2048$. Under the assumption of a sampling rate of 9.75 MHz,

2048 is smaller than the size of 1 ms IF samples. Therefore, a FOR loop is added to accumulate the intermediate results into the reduction arrays. This can be described as follows:

```
for (int ich = blockIdx.x; ich < nch; ich += gridDim.x) {…
    for (int iArray = threadIdx.x; iArray < NARRAY; iArray
+= blockDim.x) {…
        for (int pos = sampleBase + iArray;
pos < sampleEnd; pos += NACCUM)
        {…}}}
```

The added variable iArray is used to index the reduction arrays, and NARRAY is the size of one reduction array. An analysis of how to choose an optimized value for this variable is presented later in the following section.

The variables in the arrays stored in the shared memory can be summed with a tree-like reduction. Harris (2007) presents the detailed algorithm design and time complexity analysis for the parallel reduction in CUDA. We adopt the third version of his reduction algorithm and make some simplifications, such as assuming that the size of the reduction array is equal to a power of two.

The number of processing channels, N_c, and the number of samples in each channel, N_k, are all variable numbers in the proposed algorithm. Compared with the extant schemes, the proposed algorithm is independent of the GPU device, the number of tracking channels, the correlation time, and the code length, which means it can be applied on different GNSS signals.

Time complexity analysis

Assume that the number of tracking channels is N_c and the number of IF samples in one tracking channel is N_s. In order to simplify the analysis, the numbers of processed IF samples in different tracking channels are assumed to be equal, and then, the total data size is

$$N = N_c N_s \tag{5}$$

The size of the reduction array is M. The number of active blocks running in parallel on the GPU is set to be P_b. The number of active threads per block running in parallel is set to be P_t. In this analysis, $P \le N$, $P_b \le N_c$, $P_t \le N_S$, and $M \le N_s$. Therefore, the total number of the threads running in parallel on the GPU is

$$P = P_b P_t \tag{6}$$

The time complexity of the proposed algorithm is

$$\begin{aligned} &O\left(\frac{N_c}{P_b}\right)\left(O\left(\frac{M}{P_t}\cdot\frac{N_s}{M}\right)+O\left(\frac{M}{P_t}+\log M\right)\right)\\ &=O\left(\frac{N_c}{P_b}\right)\left(O\left(\frac{N_s}{P_t}\right)+O\left(\frac{M}{P_t}+\log M\right)\right)\\ &=O\left(\frac{N}{P}\right)+O\left(\frac{N_c}{P_b}\right)O\left(\frac{M}{P_t}+\log M\right) \end{aligned} \tag{7}$$

The lowest complexity occurs when $M = N_s, P = N$, $P_b = N_c$, which means that the size of the reduction array is equal to the number of IF samples in one tracking channel, the number of threads per block is equal to the number of samples, and the number of blocks is equal to the number of channels. As a consequence, Eq. (7) can be manipulated as follows:

$$\begin{aligned} &O\left(\frac{N}{P}\right)+O\left(\frac{N_c}{P_b}\right)O\left(\frac{M}{P_t}+\log M\right)\\ &=O\left(\frac{N}{P}\right)+O\left(\frac{N}{P}\right)+O\left(\frac{N_c}{P_b}\right)O(\log N_s)\\ &=O\left(\frac{N}{P}\right)+O\left(\frac{N_c\log N_s}{P_b}\right)\\ &=O(\log N_s) \end{aligned} \tag{8}$$

Compared with the time complexity of a CPU-based single-instruction single-data (SISD) correlation algorithm, which is $O(N) = O(N_c N_s)$, the GPU-based correlation algorithm significantly improves the performance from a theoretical point of view.

Code implementation and verification

A GNSS software receiver written in C programming language is built to implement and verify the proposed parallel algorithm. A modified version of the open source GNSS software receiver GNSS-SDRLIB (Suzuki and Kubo 2014) is implemented and used to provide the basic functions running on the CPU. The modified version is different from the original GNSS-SDRLIB in the following aspects:

1. The receiver functions are executed in one CPU thread serially, which provides the benchmark for the GPU-based correlation in performance evaluation.
2. The GPU-based parallel algorithm is implemented and added to the modified version. GNSS-SDRLIB implements its correlation in double float precision. However, in order to maximize the performance, the GPU-based correlation and the modified CPU-based correlation are both realized in single float precision. A macro switch is defined to change the execution mode between the GPU-based correlation and the CPU-based correlation.

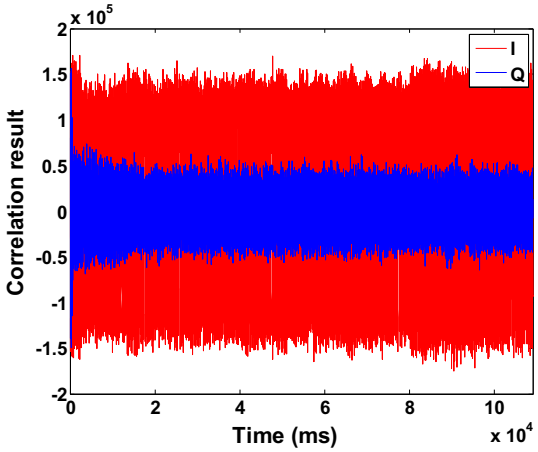


Fig. 4 Navigation bits decoded by the GPU-based tracking

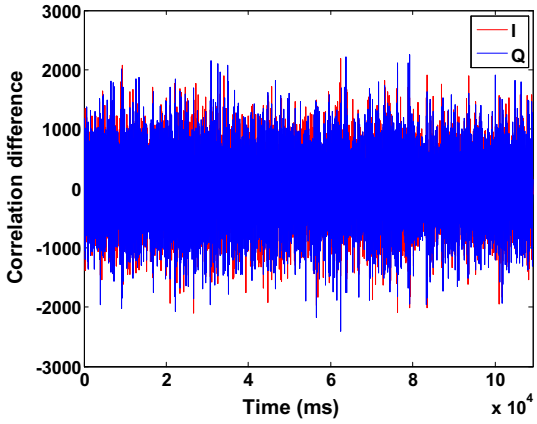


Fig. 5 Difference between correlation results in GPU and CPU

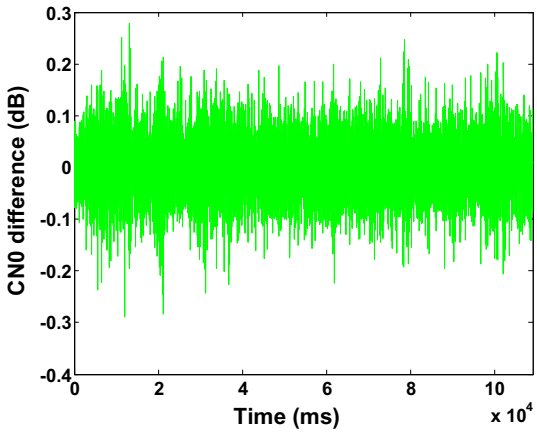


Fig. 6 Difference between CN0 results in GPU and CPU

In order to verify the correctness of the GPU-based correlation, the modified GNSS software receiver is operated to process 110 s IF samples with 9.75 MHz rate and 4-bit resolution. The tested signal is the GPS L1 C/A. The hardware used in the verification test is a NVIDIA GeForce GTX 750 Ti GPU and a quad-core 3.6 GHz Intel Core i7-4790 CPU. Both the GPU-based and the CPU-based algorithms are initialized with the same acquisition process to ensure that the initial states for the two algorithms are the same.

The GPU-based prompt correlation result, $S_{\mathrm{IQ,p}}$, and the difference between the GPU-based and the CPU-based correlation results in one channel are shown in Figs. 4 and 5. The decoded navigation bits in Fig. 4 indicates that the tracking loop can work normally with the GPU-based correlator, while there still exits some difference between the GPU-based and the CPU-based correlation results as shown in Fig. 5. Figure 6 shows the CN0 difference brought by the different correlation results in GPU and CPU. As can be seen in the figure, the CN0 difference ranges between about ± 0.3 dB.

The difference between the correlation results initially comes from the different roundoff errors of floating point numbers in the GPU and the CPU (Whitehead and Fit-Florea 2011). According to the IEEE 754 standard, the total precision of single float numbers is about seven decimal digits. The precision limit would lead the float indexes of the local replica code samples in CPU and GPU to diverse as given in Table 1. The difference can change the equality of the final correlation results only when these float indexes are converted into different integer indexes, and these integer indexes can refer to different code samples. The divergence of code samples would be expanded when they are multiplied by the product of the IF signal and the local carrier signal, which usually ranges from several hundreds to several thousands. The final difference of the correlation results would then also fall into a range from zero to several thousands as shown in Fig. 5.

Since the correlation result is used to tune the tracking loop, the difference between the current correlation results of GPU and CPU will influence the input data and the output result of the next correlation process. Once a divergence is encountered between the GPU-based and the CPU-based correlation results, it could last afterward. The correlation difference is in an order of 10^3, while the correlation results are in an order of 10^5. The relatively small difference should not invalidate the correctness of the proposed GPU-based algorithm. The CN0 difference in Fig. 6 also indicates that there is almost no sensitivity loss due to the proposed GPU-based algorithm.

Optimization

The performance evaluation and optimization of the GPU-based correlation implementation are carried out with regard to the arithmetic and memory aspects. According to

Table 1 Divergent code indexes

Channel	The first divergent correlation (ms)	Float index in CPU	Float index in GPU
0	3	649.999939	650.000000
1	10	445.000092	444.999908
2	2	762.999939	763.000000
3	4	325.999939	326.000000

Table 2 Physical limits for NVIDIA GeForce GTX 750 Ti

Compute capability	5.0
Number of SMs	5
CUDA cores/SM	128
Warp size	32
Maximum block size	1024
Maximum threads per SM	2048
Maximum shared memory per block	491,52 bytes
Shared memory per SM	65,536 bytes
Registers per SM	65,536

the time complexity analysis, the factors that influence the performance of the GPU-based correlation include the number of tracking channels, N_c, the number of IF samples in one tracking channel, N_s, the number of active blocks running in parallel on the GPU, P_b, and the number of active threads running in parallel in one block, P_t.

N_c and N_s are variable input parameters of the GPU-based correlation, while P_b and P_t reflect how effectively the hardware is kept busy. Limited by the hardware resource (Table 2), however, the maximum number of active blocks, $P_{b,m}$, and the maximum number of active threads per block, $P_{t,m}$, are values that depend on the resource usage. The resources in use include the number of threads per block, P_n, the number of registers per thread, N_r, and the amount of shared memory per block, M_s. The concept occupancy, which is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps, is employed as a metric for the utilization of the hardware resource. The CUDA toolkit provides an Excel spreadsheet called "CUDA_Occupancy_Calculator.xls" to calculate the occupancy, with inputs as the values P_n, N_r, M_s, as well as the input of the GPU compute capability. It should be noted that selecting a set of appropriate values for P_n, N_r, M_s can increase the occupancy and may even improve the execution performance.

The number of registers per thread, N_r, is highly related to the algorithm design, and it is not prone to be modified in the code implementation. The command-line option "–ptxas options = v" can be used to detail the number of registers used per thread, which is $N_r = 36$ here. However, in order to achieve a 100 % occupancy, N_r is limited to be less than the total number of registers available per block over the maximum threads per multiprocessor (Table 2): $N_r \leq 65{,}536/2048 = 32$. The simplest way to prevent the compiler from allocating too many registers, which would results in a low occupancy, is to use the "-maxrregcount = 32" command-line option to ensure that the maximum number of registers allocated per thread is smaller than 32.

Before exploring how the number of threads per block, P_n, influences the occupancy and how the occupancy influences the execution performance, the settings of the values N_c, N_s, and M_s, which should be kept as constants, are explained as follows. N_s is set to be the number of 1 ms IF samples. As for N_c, assume that a GPU-based GNSS software receivers allocate 30 channels to track a single-frequency signal for each GNSS constellation and that four constellations are considered with triple-frequency signals. Therefore, the total number of channels needed is $N_c = 4 \times 30 \times 3 = 360$. Since the assumed constellations are not yet available, simulation is used. Noise channels are duplicated from the normal tracking channels, and they are added to generate simulated signals. In the GPU-based correlation, M_s can be found from the size of the reduction array, M: $M_s = 6 \times \text{sizeof}(\text{float}) \times M = 24\,M$ bytes. Since the maximum amount of shared memory allocated per block is 49,152 bytes, then M should be smaller than $49{,}152/24 = 2048$. Moreover, because M should be equal to a power of two, there are only three candidate values for M, which are 512, 1024, and 2048. In the optimization test for choosing the appropriate value for P_n, M is set to be 1024, thus $M_s = 24{,}576$ bytes. After P_n is determined, a test on how the different values of M influence the execution performance is carried out.

The CUDA C best practices guide (NVIDIA 2014) suggests that experimentation is required to select the block size, P_n. It also indicates that P_n should be a multiple of warp size 32. An experiment is designed to search for an appropriate value for P_n from the range [128, 1024], with a step of 32. Although the number of blocks allocated, P_m, will not impact the occupancy unless it is smaller than the maximum number of active blocks, $P_{b,m}$, the experiment still measures the processing time for 1 ms IF samples in 360 channels by varying both P_m and P_n. This is because $P_{b,m}$ is determined by P_n, N_r, M_s. Since P_n is unknown, the low bound of P_m is still uncertain. A two-dimensional search would provide a detailed plot on how P_m and P_n impact the execution performance. The range for searching P_m is [10, 1024], with a step of 1. The search range lower bound is chosen as 10 because of the following. The minimal number of blocks per multiprocessor, in order to achieve 100 % occupancy, is the maximum threads per

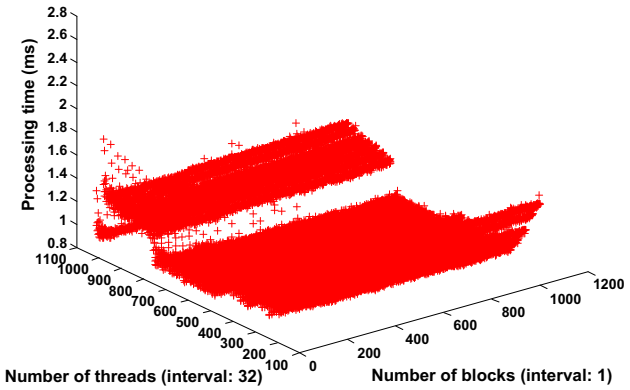


Fig. 7 Execution time measured on different grid and block sizes

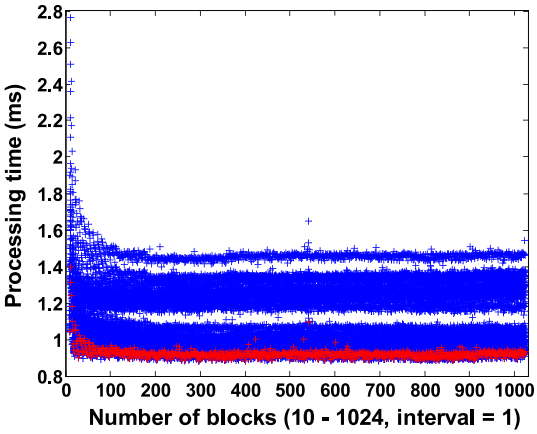


Fig. 9 Execution time measured on different block sizes

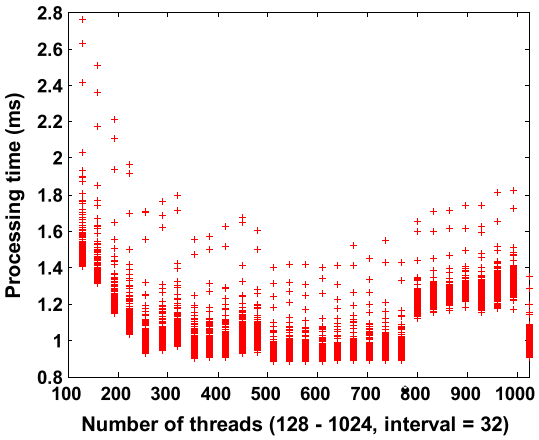


Fig. 8 Execution time measured on different grid sizes

multiprocessor over the maximum block size (Table 2), which is 2048/1024 = 2. Since there are five multiprocessors, the minimal block size should be $5 \times 2 = 10$. The experiment result is shown in Fig. 7.

Two lateral views of Fig. 7 are provided in Figs. 8 and 9 to analyze the impact of the block size and the grid size separately on the performance. From the original data, $P_m = 297$ and $P_n = 512$ generate the best performance with the least execution time, which is 0.880957 ms to process 1 ms data with 360 channels and 9.75 MHz sampling rate. However, the lateral view in Fig. 8 shows that if P_n is between 512 and 768, the performance of the GPU-based correlator keeps almost the same. When it comes to the occupancy instead of the execution performance, there is a reason to select 512 from the range [512, 768]. The reason is that the CUDA occupancy calculator indicates that 512 is the only value that can generate 100 % occupancy without any additional constraints from N_r and M_s. In fact, the execution performance at $P_n = 1024$, with only two active blocks per multiprocessor, is beyond the average because the occupancy is also 100 %. The lateral view, in turn, illustrates that lower occupancy does not always mean lower performance. Usually when 50 % occupancy is reached, additional increases in occupancy will not improve the performance.

All points with $P_n = 512$ are marked in red in the other lateral view with respect to the grid size. The red points indicate that once the blocks per grid are sufficient, the execution performance will remain steady. Choosing a value smaller than the maximum number of active blocks, $P_{b,m} = 2048/512 \times 5 = 20$, results in a low occupancy and a lack of capability to cover the latency efficiently. Since it is more flexible to choose a value for P_m, 256 is chosen for P_m, which is not a unique number, just one to guarantee sufficient blocks per gird.

There are only three options for the size of the reduction array M: 512, 1024, and 2048. The corresponding values for the amount of shared memory allocated per block M_s are 12,288, 24,576, and 49,152 bytes, respectively. Moreover, the corresponding occupancy is 100 %, 50 %, and 25 %, respectively. This is according to the CUDA occupancy calculator since P_n and N_r are already determined. In order to choose an optimized value for M_s, the GPU-based correlator is operated to process 1 ms IF samples with 9.75 MHz rate. With P_n and N_r determined, the number of the processing channels, N_c, is varying in the range of [0, 1024], with a step of 1, this time, to generate numerous test results. The measured execution time of the kernel function in Fig. 10 shows again that occupancy higher than 50 % will not bring in any extra performance enhancement in this program, while occupancy lower than 50 % would definitely degrade the efficiency. Ultimately 512 is chosen for M since it generates 100 % occupancy.

Apart from the arithmetic optimization, the data transfer between the host memory and the device memory should also be considered for optimization. One significant consideration for memory optimization is the coalescing

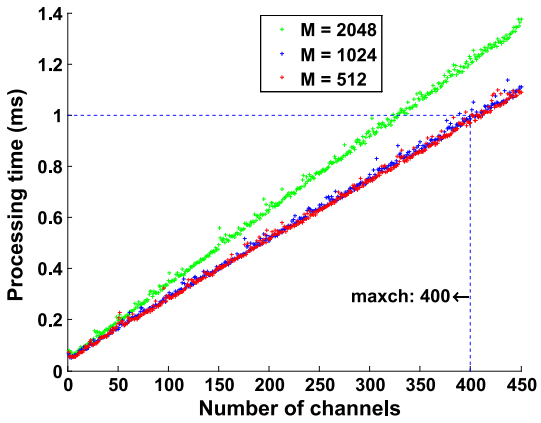


Fig. 10 Execution time of kernel with different usages of shared memory

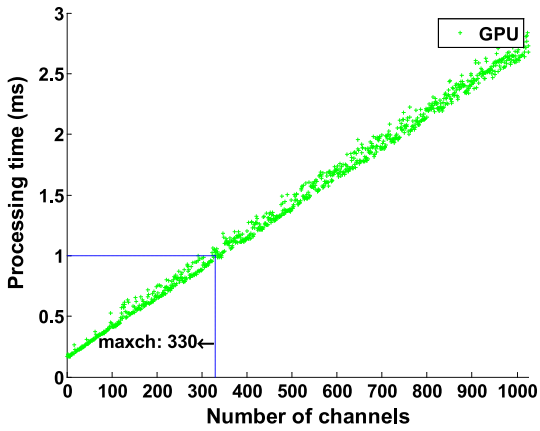


Fig. 11 Overall performance of GPU-based correlation

Table 3 Throughputs of scattering and coalescing memory transfers

Throughput	Load (MB/s)	Store (MB/s)
Scattering memory copy	189.189	95.89
Coalescing memory copy	921.053	472.973

memory transfer. In the GPU-based correlation, the carrier and code tables are generated and copied from the CPU memory to the GPU memory only once. Among the rest three input parameters, the IF samples and the number of processing channels, N_c, are copied from the CPU memory to the GPU memory for every correlation. Such unidirectional memory transmission is unavoidable and cannot be coalesced. Only the last parameter, which is an array of structures storing the parameters of the channels, whose size is proportional to the number of tracking channels N_c, can cause bidirectional memory transmission between the CPU and the GPU memory (load and store). The optimization strategy is to copy all the parameters of the channels in and out of the GPU memory, instead of one-by-one channel. A comparison is made between the different throughputs of the coalescing memory copy and scattering memory copy when $N_c = 5$, which is given in Table 3.

The results in Table 3 are gathered by NVIDIA Visual Profiler, which is a tool in the CUDA Toolkit to analyze the CUDA C program. The table shows the throughput performance of the coalescing memory copy is about five times as that of the scattering memory copy. The reason is that the load and store throughputs of both methods are far smaller than the peak memory bandwidth, which is 8 GB/s between the host memory and the device memory. Therefore, coalescing memory copy can significantly expand the transfer memory size in one load or store process, which eventually increases the throughput.

Overall performance

An overall performance evaluation of the GPU-based parallel correlation is conducted, which includes a comparison with the performance of the CPU-based serial correlation. The GPU-based correlator is operated to process 1 ms IF samples. The sampling rate is 9.75 MHz. The number of the processing channels, N_c, is varying in the range of [0, 1024], with a step of 1. Different from the test that selects the shared memory size, which only measures the execution time of the kernel function, this experimentation measures the execution time of both the kernel function and the memory transfer. The execution time of the GPU-based correlation varies with the number of tracking channels as shown in Fig. 11. A rough estimation of how many channels the GPU-based correlation can maintain in real time can be obtained. As shown in Fig. 11, when the processing time, indicated by the value of the y axis, is 1 ms, which is equal to the time required to process 1 ms IF samples in real time, the number of channels, indicated by the value of the x axis, is roughly 330.

Seo et al. (2011) made a performance comparison between several GPU-based GPS SDRs. A similar comparison is given in Table 4, where we reuse the content of Table 1 in Seo's paper and append information about algorithms developed in recent years (Seo et al. 2011; Huang et al. 2013; Park et al. 2014). The performance of our algorithm is also described in Table 4. This comparison is very coarse without considering the impact brought on by different devices used in the algorithm implementations. However, the comparison still provides an overview of the performance difference between existing GPU-based GNSS SDRs.

We also made experimental comparisons between the GPU-based and the CPU-based correlations. Massive noise channels are simulated in the CPU by running the correlation of one channel many times. The comparison of the

Table 4 Comparison of some existing GPU-based SDRs

GPU	CUDA capability	Number of SMs	Number of CUDA cores	GPU-based GNSS SDRs' development
GeForce GTX 8800	1.0	16	128	8 Channels, 40 MHz, 8-bit resolution
GeForce GTX 280	1.3	30	240	12 Channels, 8 MHz, 4-bit resolution
GeForce GTX 285	1.3	30	240	150 Channels, 5 MHz, 14-bit resolution
GeForce GTX 480	2.0	15	480	Acquisition only
GeForce GTX 480	2.0	15	480	60 Tracking channels, 40 MHz, 8-bit resolution
GeForce GTX 580	2.0	16	512	Real-time tracking of all navigation signals
GeForce GTX-Titan	3.5	14	2688	70 Tracking channels, 16.3676 MHz
GeForce GTX 750 Ti	5.0	5	640	330 Tracking channels, 19.5 MHz, 4-bit resolution

SM stream multiprocessors

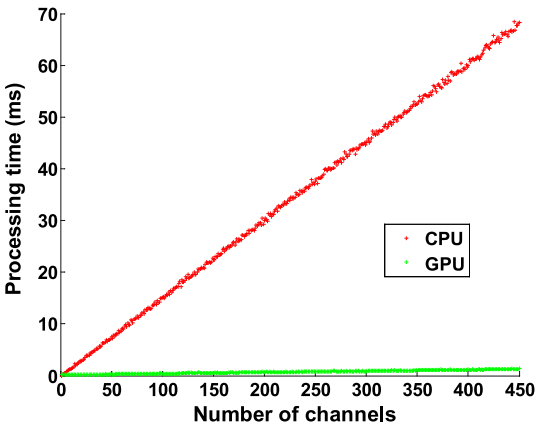


Fig. 12 Performance of GPU-based and CPU-based correlations

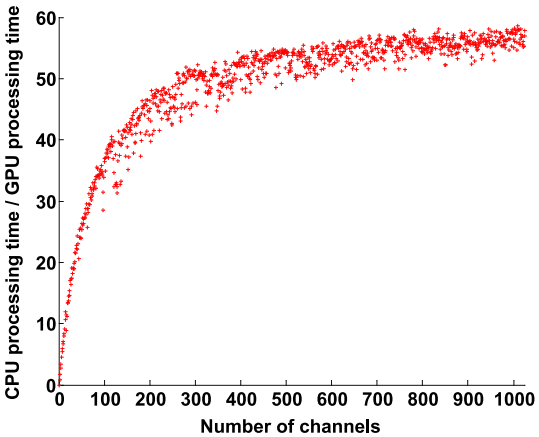


Fig. 13 Speedup brought by the GPU-based correlation

execution time between the CPU-based and the GPU-based correlations varies with the number of channels as shown in Figs. 12 and 13, where the plot shows a huge performance gap between the CPU and the GPU computations. The range of the x axis is compressed from [0, 1024] to [0, 450] so that the scatter plot of the GPU performance can be shown clearly. The other plot on acceleration rate shows that the speedup grows when the number of channels increases. However, the growth rate decreases gradually until the speedup reaches a peak value at nearly 53. This plot can also be used to check out the expected gain when the GPU-based correlator is operated to process different numbers of channels.

The performance speedup of the whole GNSS software receiver, s, caused by the speedup gained from the GPU-based correlation, s_p, can be calculated by Amdahl's law (Tanenbaum et al. 2013):

$$s = \frac{1}{1 - r_p + \frac{r_p}{s_p}} = \frac{s_p}{s_p(1 - r_p) + r_p} \tag{9}$$

where r_p is the fraction of the correlation computation load out of the whole computation of the GNSS software receiver on the CPU.

Other CUDA features

The design and implementation approach presented above can be applied on all the CUDA-enabled GPUs. While for the GPUs with CUDA capability greater than 3.0, there are some new CUDA features that can be utilized.

SM 3.0 introduced the warp shuffle functions, in which "__shfl_down" and "__shfl_xor" functions can be used to perform a reduction across 32 threads in a warp without consuming any shared memory. The shuffle functions exchange a variable between threads within a warp, which has a faster transfer speed compared with that of the shared memory. Figure 14 shows how "__shufl_down" function works.

In the GPU-based correlation, a block reduction, rather than a warp reduction, is needed. Atomic addition function in CUDA is adopted here to guarantee that the sum of variables in each warp is added correctly, excluding the cache pollution brought by writing the same GPU memory with different CUDA threads. Without storing the reduction arrays in the shared memory, the algorithm design is

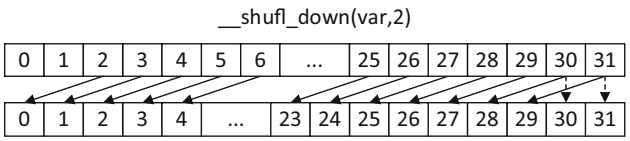


Fig. 14 Shuffle instruction

simplified into two FOR loops. Some key codes realizing the architecture of the new design are listed below:

```
for (int ich = blockIdx.x; ich < nch; ich += gridDim.x) {…
        for (int pos = sampleBase + threadIdx.x;
    pos < sampleEnd; pos +=
      blockDim.x) {…}
            for (int offset = warpSize; offset > 0;
    offset >> = 1) {
                sum_I[0] += __shfl_down(sum_I[0], offset);
    …}
            if (!(threadIdx.x & 0x1f)) {
              atomicAdd(d_ch[ich].I, sum_I[0]); …}
}
```

The usage of the atomic functions is at the cost that no other thread can access the address until the running atomic operation is complete. This cancels out the benefit brought by the usage of the shuffle instructions instead of the shared memory. Verification and performance tests are conducted for the new design. The results indicate that there is no significant gain in the performance. Nevertheless, using the shuffle instructions to implement the final block reduction can eliminate the possibility that the occupancy of the program is limited by the usage of the shared memory. Therefore, it is strongly recommended to use the shuffle instructions instead of the shared memory if a GPU card with CUDA capability >3.0 is available.

Unified memory is another new feature that can be applied in the implementation of the GPU-based correlation. It is first introduced in CUDA 6.0 and requires a GPU card with SM 3.0 or higher, and a 64-bit host application and operating system. The unified memory programming provides memory management that enables the CPU and the GPU to access the memory allocated in the managed memory without explicit data transfer between the host and the device memory. This would simplify the GPU programming and avoid the performance degradation brought by the inefficient explicit memory transfer coded by programmers. There is a real test on how the unified memory programming can affect the performance of the GPU-based correlation as given in Table 5. Even though the modified program is exempt from explicit memory copy, the implicit memory copy increases the execution time of the kernel function. The overall performance does not get improved; nevertheless, the unified memory programming is still a good way to avoid explicit memory transfers.

Table 5 Duration of kernel in unified memory

	Duration (μs)
Explicit memory copy (load)	1.216
Explicit memory copy (store)	2.368
Kernel execution (explicit memory copy)	18.783
Kernel execution (implicit memory copy)	23.935

The two aforementioned advanced features are only available for certain GPU cards. In order to scale to the future devices, they are strongly recommended to use. If the compatibility is more important, the former design and implementation should be adopted.

Conclusions

The GPGPU technology has been widely applied in numerous scientific calculations, including the correlation process in GNSS software receivers since CUDA was released in 2006. A well-designed architecture and a deeply optimized implementation are indispensable to exert the full potential of a CUDA-enabled GPU. We propose a GPU-based correlator architecture design for GNSS software receivers. The new architecture addresses the design of the data structures and the mechanism of mapping the correlation process of multiple channels into blocks of threads executed on a GPU. A time complexity analysis of the proposed parallel algorithm is conducted, which asserts the theoretical advantage of the GPU-based correlator over the CPU-based one. Following the analysis, a software prototype is established to verify the correctness and evaluate the performance of the proposed GPU-based correlator architecture. The tests verified the correctness of the implementation of the GPU-based correlator. The results also asserted the considerations on how to choose the execution parameters and the schemes to keep a high memory transfer throughput between the host memory and the device memory. An overall performance evaluation, including both the arithmetic computing and the memory transfer, is presented after the initial code implementation is fully optimized. The evaluation indicates that the GPU-based correlator can maintain the correlation process of nearly 330 tracking channels with 9.75 MHz sampling rate and 4-bit resolution in real time. Moreover, the comparison with the performance of the CPU-based correlator shows that the GPU-based correlator can achieve above 50 times performance gain at the peak. We also apply and test some

new features in recently released CUDA-enabled GPUs. The benefits and drawbacks of those new features are illustrated, and some recommendations on when to adopt those new features are provided.

Acknowledgments This work was supported by the National Natural Science Foundation of China (41174028, 61273053, 41404029), China Postdoctoral Science Foundation funded project (2013M542061, 2014T70738), and National Natural Science Foundation of Hubei province (2014CFB727).

References

Buck I, Tim F, Daniel H, Jeremy S, Kayvon F, Mike H, Pat H (2004) Brook for GPUs: stream computing on graphics hardware. ACM Trans Graph (TOG) 23:777–786

Chen YH, Juang JC, Seo J, Lo S, Akos DM, De Lorenzo DS, Enge P (2012) Design and implementation of real-time software radio for anti-interference GPS/WAAS sensors. Sensors 10:13417–13440

Farber R (2011) CUDA application design and development. Elsevier, Amsterdam

Harris M (2007) Optimizing parallel reduction in CUDA. Available on the Internet

Heckler GW, Garrison JL (2006) SIMD correlator library for GNSS software receivers. GPS Solut 10(4):269–276

Hobiger T, Tadahiro G, Jun A, Yasuhiro K, Tetsuro K (2010) A GPU based real-time GPS software receiver. GPS Solut 14(2):207–216

Huang B, Yao Z, Guo F, Deng S, Cui X, Lu M (2013) STARx—a GPU based multi-system full-band real-time GNSS software receiver. ION GNSS+ 2013, Institute of Navigation, Nashville, Tennessee, September, pp 1549–1559

Karimi K, Pamir AG, Afzal MH (2013) Accelerating a cloud-based software GNSS receiver. arXiv preprint arXiv:1309.0052

Nvidia (2014) CUDA C best practices guide. Available on the Internet

Pany T, Gohler E, Irsigler M, Winkel J (2010) On the state-of-the-art of real-time GNSS signal acquisition—a comparison of time and frequency domain methods. Indoor positioning and indoor navigation (IPIN), 2010 international conference, pp 1–8

Pany T, Riedl B, Winkel J (2010) Efficient GNSS signal acquisition with massive parallel algorithms using GPUs. In: Proceedings of ION NTM 2010, San Diego, CA, pp 1889–1895

Park KW, Yang JS, Park C, Lee MJ (2014) Implementation of GPGPU based real-time signal acquisition and tacking module for multi-constellation GNSS software receiver. In: Proceedings of ION GNSS+ 2014, Institute of Navigation, Tampa, FL, September, pp 1410–1416

Roule P, Jakubov O et al (2013) GNSS signal processing in GPU. Artif Satell 48(2):51–61

Seo J, Chen YH, De Lorenzo DS, Lo S, Enge P, Akos D, Lee J (2011) A real-time capable software-defined receiver using GPU for adaptive anti-jam GPS Sensors. Sensors 9:8966–8991

Suzuki T, Kubo N (2014) GNSS-SDRLIB: An open-source and real-time GNSS software defined radio library. In: Proceedings of ION GNSS+ 2014, Institute of Navigation, Tampa, FL, September, pp 1364–1375

Tanenbaum AS, Austin T, Chandavarkar BR (2013) Structured computer organization. Pearson, London

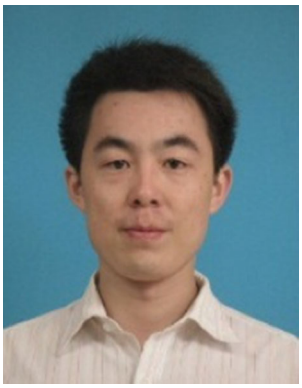
Whitehead N, Fit-Florea A (2011) Precision and performance: Floating point and IEEE 754 compliance for NVIDIA GPUs. Available on the Internet

Wilt N (2013) The CUDA handbook: a comprehensive guide to GPU programming. Pearson, London

Liangchun Xu is a Ph.D. candidate in GNSS Research Center at Wuhan University. His research interests include GNSS receiver technology.

Nesreen I. Ziedan is the Acting Department Head and an Associate Professor at the Computer and Systems Engineering Department, Faculty of Engineering, Zagazig University, Egypt. She received a Ph.D. in Electrical and Computer Engineering from Purdue University, USA. She also holds an M.S. degree in Control and Computer Engineering, a Diploma in Computer Networks, and a B.S. degree in Electronics and Communications Engineering. Prof. Ziedan has several US patents in GPS receivers design and processing. She is an Associate Fellow of the Royal Institute of Navigation (AFRIN), and she is a member of the Editorial Advisory Board of the RIN Journal of Navigation.

Xiaoji Niu is a Professor at the GNSS Research Center, Wuhan University, China. He received B.Eng. degree (with honors) in Mechanical and Electrical Engineering and the Ph.D. from Tsinghua University, Beijing, China, in 1997 and 2002, respectively. From 2003 to 2007, he was a Post-Doctoral Fellow with the Mobile Multi-Sensor Systems (MMSS) Research Group, Department of Geomatics Engineering, University of Calgary. From 2007 to 2009, he was a Senior Scientist with SiRF Technology, Inc. His research interests focus on INS and GNSS/INS integration.



Wenfei Guo received Ph.D. in Communication and Information System from Wuhan University in 2011. He now is a teacher in GNSS Research Center, Wuhan University. His research currently focuses on GNSS receivers.