Journal of Combinatorial Mathematics
and Combinatorial Computing
www.combinatorialpress.com/jcmcc

Accelerated cascade integrator comb filter with a new non-recursive GPU implementation

Yanhao Guan[1], Yi Lu[1], Guolin Shao[1,✉]

[1] *School of Software, Nanchang University, Nanchang, Jiangxi, 330031, China*

ABSTRACT

The Cascaded Integrator Comb (CIC) decimation filter is a pivotal technology extensively employed in digital signal processing (DSP). This paper delves into a comprehensive examination of the CIC algorithm within software-defined radio (SDR) systems from the perspective of parallel computing and introduces a novel Non-Recursive Implementation (NR-I) on an NVIDIA GPU using CUDA. The NR-I approach significantly reduces computational load by unfolding the recursive CIC structure with pre-derived Unfold Factors. Further optimization was achieved through data-transfer enhancements using PM Implementation (PM-I) and ODT Implementation (ODT-I). Experimental results demonstrate that NR-I achieves a speedup of over 449.48. Additionally, the data-transfer optimizations resulted in substantial performance improvements, with PM-I and ODT-I reducing execution time by 43.24% and 64.22%, respectively. The GPU implementation's speedup is significantly greater than that of OpenMP, ranging from 3.34 to 10.22 times. These results underscore the effectiveness of the proposed Non-Recursive Implementation in accelerating time-intensive and data-intensive computations.

Keywords: cascaded integrator-comb decimation filter, non-recursive CIC, CUDA, GPU implementation, open MP

1. Introduction

The CIC decimation filter is a digital multirate filter used for managing substantial sample rate changes in DSP systems [13]. Its highly symmetric structure enables efficient implementation, leading to its widespread application across various DSP domains [2]. This paper aims to examine and propose a high-performance software algorithm for implementing the CIC filter in an SDR system.

✉ Corresponding author.
E-mail address: shaoguolin@163.com (G. Shao).

Received 19 July 2024; Accepted 09 December 2024; Published Online 16 March 2025.

DOI: 10.61091/jcmcc124-04

The CIC algorithm is extensively employed in various SDR applications, including digital down converters (DDCs), digital up converters (DUCs), sample rate conversion (SRC) systems, and both transmission and reception systems, among others. Additionally, researchers have focused significantly on software implementation and optimization techniques. These approaches can be broadly categorized based on the underlying technology, including Digital Signal Processor (DSP) implementations, Field Programmable Gate Array (FPGA) implementations, Universal Software Radio Peripheral (USRP) implementations, and multi-core CPU parallel implementations.

DSP represents a specialized microprocessor, tailored explicitly for executing digital signal processing duties. In the context of CAI SS's research, a DDC framework leveraging DSP technology was developed and deployed, resulting in substantial enhancements to both the performance and adaptability of communication systems.

FPGA, on the other hand, constitutes an integrated circuit architecture that allows for in-field customization by programming professionals. Numerous prior endeavors have delved into this domain, including the work of Datta et al. [7], who concentrated on a configurable CIC decimator endowed with pruning features aimed at minimizing hardware resource consumption. Furthermore, by employing partitioning strategies within the decimator factor, a notable reduction in computational latency can be achieved.

The Universal Software Radio Peripheral (USRP) finds ubiquitous application in the realm of Software-Defined Radio (SDR) [8, 1], serving as a versatile and cost-effective transceiver that transforms standard hosts into potent wireless prototyping platforms [15]. Kodali R K et al. [12] delved into the intricacies of CIC and various other filter types employed in Direct Digital Conversion (DDC) and Direct Up Conversion (DUC) processes, offering insights into their utilization across diverse USRP platforms.

Multi-core Central Processing Units (CPUs) have gained widespread adoption across myriad application domains, SDR being no exception. These CPUs, collectively, possess the capability to concurrently execute multiple instructions, enhancing overall efficiency. OpenMP, a prevalent shared memory parallel programming interface, provides high-level constructs that have been extensively employed in SDR endeavors [3], facilitating seamless parallel execution.

In recent times, General-Purpose computing on Graphics Processing Units (GPGPU) has emerged as a potent means of tackling DSP algorithms [14, 4, 19]. Given their exceptional data and thread parallelism, immense computational prowess, and relatively low cost, GPGPUs have garnered significant attention for executing time- and data-intensive algorithms. NVIDIA's introduction of the CUDA development environment in 2007 further propelled GPU computing by establishing a general-purpose parallel computing architecture that simplifies and accelerates programming efforts [5]. As a viable acceleration method, GPUs offer the dual advantage of FPGA's high-performance capabilities and the flexibility of general-purpose programming languages, making them suitable for supporting real-time SDR systems. Prior research has explored the feasibility of GPU-accelerated DSP algorithms [16, 11], yet, to the best of our knowledge, no study has specifically targeted the CIC algorithm within an SDR context and addressed its implementation on GPUs.

In this work, we conducted an exhaustive analysis of the CIC algorithm with a focus on its GPU-based implementation. Specifically, we introduced the R-I approach for executing the CIC decimation filter on the GPU, followed by the Novel NR-I method aimed at enhancing performance. A comparative assessment between R-I and NR-I revealed that the latter exhibited significantly superior performance. To delve deeper into optimization, we elaborated on practical strategies for optimizing data transfers. Additionally, we presented the sequential CPU version and OpenMP-based

implementation as benchmark tests to quantify the overall speedup achieved by our CIC algorithm implementation.

2. Overview of Cic algorithm and cuda architecture

2.1. Cascaded integrator-comb (CIC) decimation filter

Over two decades ago, Eugene B. Hogenauer first introduced this concept [10], which has since been employed for both interpolation and decimation purposes [9]. Notably, it boasts a multiplier-free architecture, requiring minimal memory and hardware complexity, comprising solely of adders and delay elements. The CIC filter's low power consumption is a significant advantage, making it a popular choice in digital down converters (DDC) and digital up converters (DUC) [6]. Essentially, the CIC filter's primary functions involve adjusting sampling rates, either by decreasing them (decimation) or increasing them (interpolation). Typically, the CIC structure is comprised of multiple ideal integrator and comb filter stages [17].

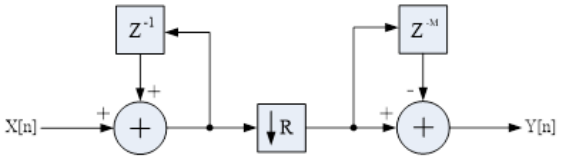

Fig. 1. The Core Composition of a Single-Stage CIC Architecture

Figure 1 displays the primary architecture of a single-stage CIC decimation filter, comprising a pair of integrator and comb filters, with a down-sampling unit interposed between them. Considering f_s as the sampling frequency within the integrator section, the down-sampling unit processes the signal to produce a lower sampling rate of f_s/R, which then serves as the input sequence for the comb section. For a given data sequence S, the integrator section aims to determine the output sequence S' in accordance with the prescribed equation.

$$S'[n] \quad = \quad S'[n-1] + S[n]. \tag{1}$$

The equation for the comb-section can be formulated as given in Eq. (2), where the output sequence O is solely determined by three integer parameters: R, D and N. These parameters correspond to the decimation ratio, the differential delay (commonly 1, but occasionally 2), and the number of stages in the filter, respectively.

$$O[n] \quad = \quad S'[n] - S'[n-RD]. \tag{2}$$

By integrating Eq. (1) and Eq. (2), the combined integrator-comb section can be mathematically represented as Eq. (3).

$$O[n] = \sum_{k=0}^{RD-1} S[n-k] = O[n-1] + S[n] - S[n-RD]. \tag{3}$$

Typically, a CIC decimation filter incorporates multiple stages, and the realization of an N-stage CIC filter follows a recursive approach. Each stage's objective is encapsulated in Eq.(3). For a given sequence of length L, the CIC filter's task involves deriving a sequence O of length L/RD. The recursive process of an N-stage CIC filter is outlined in Algorithm 1, encompassing two distinct

phases: the cascaded integrator-comb phase and the decimation phase. During the first phase, the process adheres to Eq. (3), where the output sequence from one stage serves as the input for the subsequent stage in an N-stage CIC configuration. Subsequently, the decimation process selects one input sample for every RD output samples, resulting in an output sequence O with a length of L/RD.

Algorithm 1 General CIC Algorithm Implementation

Require: S, R, D, N, L
for $n = 0$ to $N - 1$ **do**
  for $k = 0$ to $L - 1$ **do**
    $i = k$, $O[k] = 0$
    while $i \geq 0$ and $i > k - RD$ **do**
      $O[k] = O[k] + S[i]$
      $i = i - 1$
    end while
  end for
end for
$k = 0$, $i = 0$
while $k \leq n - 1$ **do**
  $O[i] = S[k]$
  $k = k + RD$
  $i = i + 1$
end while
return O

2.2. CUDA architecture and programming overview

NVIDIA unveiled the CUDA development environment in 2007, establishing a universal parallel computing platform and programming interface paradigm that significantly enhances the efficiency and accessibility of GPU computing. As depicted in Figure 2, the CUDA architecture and programming model encompass two distinct code types: host-side code, executed on the CPU, and device-side code, executed on the GPU. This parallel execution on the GPU intertwines with serial execution on the CPU. On the device side, threads are structured in a three-tiered hierarchy: grids, blocks, and threads. Memory, meanwhile, is categorized into six types: Registers, Local Memory, Shared Memory, Global Memory, Constant Memory, and Texture Memory.

In the CUDA programming model, the fundamental execution unit on the device side is the thread. These threads can be grouped into thread blocks, which form a cohesive unit capable of efficient data sharing and intercommunication via shared memory. However, cooperation between threads belonging to different blocks is hindered by the need for complex synchronization mechanisms. Similarly, thread blocks can be assembled into grids, where each grid executes the same kernel. Nevertheless, threads in separate blocks within the same grid cannot directly communicate with each other.

On the host side, the basic execution unit in the CUDA programming model is referred to as a kernel. When a CUDA program initiates a kernel on the host, the thread blocks within the grid are enumerated and dispatched to multiprocessors with available computational resources. This allows for the parallel execution of all threads within the grid [20].

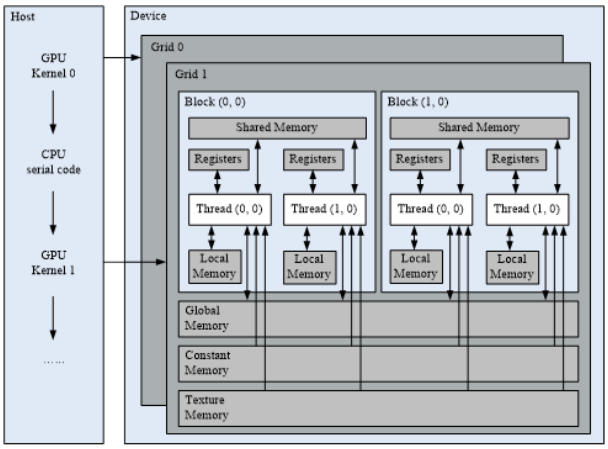

Fig. 2. The CUDA architecture

3. The proposed Gpu implementation and optimization

In this section, we initially discuss the block-based computational method for managing extensive signal data over prolonged durations. Subsequently, we individually introduce the core concepts and implementation details of the Recursive Implementation (R-I) and the Non-Recursive Implementation (NR-I). To conclude, we briefly touch upon the subject of optimizing data transfer processes.

3.1. Study on data dependence of CIC algorithm

Tackling the challenge of processing lengthy signals is a common endeavor in signal processing, often addressed by transforming extensive data-sequence computations into manageable block-wise calculations. This involves partitioning the input data-sequence into segments, each of which can be individually processed. Assuming that the extensive signal sequence is segmented into length-N data blocks, the k-th segment, denoted as S_k, is represented by Eq.(4).

$$S_k = \{s\,[k*N]\,,\ s\,[k*N+1]\,,\ \cdots,\ s\,[(k+1)*N-2]\,,\ s\,[(k+1)*N-1]\}\,. \tag{4}$$

As depicted in Eq. (1) through (3), during the initial CIC stage, the computation of the element $s\,[k*N]$ necessitates the utilization of RD-1 additional data items from the preceding segment to determine the corresponding O value. For multi-stage CIC implementations, the quantity of required data items from the prior segment surpasses RD-1. Consequently, it is imperative for the algorithm's execution to account for data dependencies to avoid incorrect output results from block-based computations. To address this challenge, we have employed an overlap-save mechanism in this paper. To facilitate understanding, we have outlined the notations utilized in the GPU-based implementation of CIC in Table 1.

Addressing the task of filtering a lengthy signal data-sequence S within an overlap-save framework, it is necessary to segment S into smaller data blocks, each with a length denoted as L_B. As previously noted, additional input samples from the preceding segment (k-1) are required to process segment k. Assuming the overlap length in the previous segment is L_O, it follows that L_I data items are necessary to process a segment of length L_B . In simpler terms, the input sequence length L_I equates to the sum of L_B and L_O. Regarding the N-stage CIC decimation filter, the initial CIC stage outcome $O\,[k]$ can be calculated using Eq. (5), which incorporates $S\,[k]$ along with RD-1 additional data items.

Table 1. Symbol explanation

Notation	Explanation
S	Input data-sequence
O	Output data-sequence
L_I	Input data length
L_B	Data segment size for single block processing
L_O	Overlap prefix length in CIC kernel execution
R	Decimation ratio
D	Differential delay
N	Number of stages
UF	Unfold Factors in NR-I
NT	Thread count per block in CUDA configuration
NB	Block count in CUDA programming model
NB_{IC}	The number of blocks in CUDA for the CIC Integrator-Comb pair kernel
NB_{DE}	The number of blocks in CUDA for the CIC decimation kernel

Specifically, the most preceding data item required is $S[k - RD + 1]$.

$$O[k] = O[k] + O[k-1] + ... + O[k - RD + 1]. \tag{5}$$

Building upon the aforementioned foundation, during the second CIC stage, $O[k]$ can likewise be computed using Eq. (5), albeit with a crucial distinction: the $S[k - RD + 1]$ mentioned here corresponds to $O[k - RD + 1]$ from the preceding CIC stage. Analogously, $O[k - RD + 1]$ necessitates RD-1 preceding data items prior to the index $k - RD + 1$. Consequently, the first CIC stage necessitates RD-1 additional data items, while the second stage requires a total of $2 * (RD - 1)$ data items. This trend can be extrapolated, with each subsequent stage demanding an additional RD-1 previous data items. In conclusion, for an N-stage CIC decimation filter characterized by parameters R and D, the prefix data length for overlap across the CIC stages can be formulated as follows.

$$L_O = (RD - 1) * N. \tag{6}$$

3.2. Design of a GPU-optimized recursive CIC algorithm

To facilitate the execution of CIC on a GPU, the data subject to processing must be transferred to the GPU's global memory and subsequently dispersed among multiple thread blocks for computation. Nevertheless, a primary challenge lies in the absence of CUDA mechanisms that permit inter-block thread synchronization, rendering recursive CIC application within a single kernel launch impractical. To overcome this limitation, the aforementioned two-phase CIC process must be implemented discretely, with the cascaded integrator-comb phase and the decimation phase handled and invoked independently.

To embed the CIC process within the GPU architecture, the R-I approach is introduced. This implementation comprises two distinct kernels, tailored to execute the functionalities of the integrator-comb phase and the decimation phase, respectively. Specifically, for the cascaded integrator-comb phase, a CUDA kernel titled CIC_IC_Kernel is devised to encapsulate the single CIC integrator-comb operation. When dealing with an N-stage CIC process, the desired effect is achieved by sequentially invoking the CIC_IC_Kernel N times. As shown in Algorithm 2.

Algorithm 2 The CIC_IC_Kernel (the CIC integrator-comb pair kernel)

Require: X, R, M, L_o, Y
1: $idx = blockIdx.x * blockDim.x + threadIdx.x$
2: **if** $idx < L_o$ **then**
3: $tp = X[idx]$
4: $idx_{min} = idx - R * D$
5: $i = idx - 1$
6: **while** $i > idx_{min}$ and $i \geq 0$ **do**
7: $tp = tp + S[i]$
8: $i = i - 1$
9: **end while**
10: $O[idx] = tp$
11: **end if**
12: **return** O

During the downsampling or decimation phase, the corresponding GPU-oriented implementation is outlined in Algorithm 3. The primary objective of this decimation process is to extract a single output sample for every RD input samples, thereby reducing the length of the output array O to L/RD subsequent to the decimation operation.

Algorithm 3 The CIC_DE_Kernel

Require: S, R, D, L, L_o, O
1: $idx = blockIdx.x * blockDim.x + threadIdx.x$
2: $k = idx * RD + L_o$
3: **if** $k < L$ **then**
4: $O[idx] = S[k]$
5: **end if**
6: **return** O

The execution strategy for launching the CIC kernels is detailed in Algorithm 4. When tackling an N-stage CIC process, the cascaded integrator-comb phase is implemented by iteratively launching the CIC Integrator-Comb kernel N times. Following each kernel launch, the output generated from the preceding stage serves as the input sequence for the subsequent stage. Subsequently, the decimation phase is enacted by invoking the CIC decimation kernel.

3.3. Design of a GPU-optimized non-recursive CIC algorithm

As outlined in the preceding section, to achieve the effect of N-stage CIC process, the recursive CIC algorithm was proposed. However, launching CUDA kernel frequently will have a certain impact on the performance of the algorithm. For more optimization, we studied the multi-stage structure of CIC algorithm in depth, and proposed a Non-recursive CIC algorithm on GPU. Algorithm 1 reveals that the multiple stages CIC implementation in C language is the structure of two-layer *for-loop*. For simplicity, we explain the 2-stage CIC process with parameters R=2 and D=1, assume S is the input data-sequence with items $S\,[1]$, $S\,[2]$, ..., $S\,[n]$, then define O_1 and O_2 as the output sequence in the first and second CIC stage. According to Algorithm 1, $O_1\,[n]$ is equal to $S\,[n]$+$S\,[n-1]$; likewise, $O_2\,[n]$ is equal to $O_1\,[n]$+$O_1\,[n-1]$, and it can be further expressed as $S\,[n]$+$2S\,[n-1]$+$S\,[n-2]$.

Algorithm 4 The CIC kernels launch algorithm

Require: S, R, D, N, L, L_o

$N_B = L/NT + (L \mod NT == 0 ? 0 : 1)$
$N_{DE} = L_o/RD$
$src = S$, $dst = O$
for $n = 1$ to N **do**
    CIC_IC_kernel $\ll< N_B, NT \gg> (src, R, D, L, dst)$
    $tp\_ptr = src$
    $src = dst$
    $dst = tp\_ptr$
end for
CIC_DE_kernel $\ll< N_{DE}, NT \gg> (src, R, D, L, L_o, dst)$
return O

From the demonstration procedure described above, it is not hard to draw the conclusion that, in the 2-stage CIC structure with parameters R=2 and D=1, for any item $S[k]$, the relative CIC result $O_2[k]$ can be expressed as $S[k]+2S[k-1]+S[k-2]$, a set of weighting coefficients [1, 2, 1] can be found out to present the non-recursive CIC structure. According to its principle of generating, UF can be pre-derived as described in Algorithm 5.

Algorithm 6 introduces a non-iterative approach for implementation. As previously mentioned, in the context of an N-stage CIC decimation filter, the computation of $O[k]$ in the N-stage CIC necessitates the preceding L_O data elements. Consequently, the non-iterative formulation can be formulated as follows. Here, we designate UF as the Unfolding Factors pertaining to the CIC.

$$O[k] = S[k] * UF[0] + S[k-1] * UF[1] + ... + S[k - L_O + 1] * UF[L_O]. \tag{7}$$

Assuming a CUDA configuration with NT threads per block, given that only L_B/RD data elements need to be processed, the number of thread blocks NB can be calculated as $NB = (L_B/RD)/NT$. Furthermore, since the length of the input sequence exceeds the number of threads by a factor of RD, the mapping between the data index and the thread index is specified in Eq. (8). It is important to note that after the CIC stage, the length of the output is reduced to $1/RD$ of the original input length.

$$I_R = idx * RD + L_O. \tag{8}$$

3.4. Data-transfer optimization

Within the realm of GPU parallel computing acceleration frameworks, efficiently transferring data between the system RAM and GPU's VRAM is paramount, as it significantly impacts overall performance [18]. Analysis of execution times reveals that data transfer accounts for 77.21% to 87.76% of the total time in the non-recursive implementation (NR-I), underscoring its status as a major bottleneck. To mitigate this, two strategies were employed to minimize data transfer delays between the host and device: employing pinned memory instead of pageable memory (PM-I), and overlapping kernel execution with data transfer (ODT-I).

When allocating CPU memory destined for GPU processing, a choice exists between pageable and pinned memory. In scenarios where data is transferred from pageable host memory to device memory, pinned memory typically serves as an intermediary, adding unnecessary overhead and hindering

Algorithm 5 The unfold factors generating algorithm

Require: $S, S\_BAK, R, D, N, L_o, UF$

$S\_RD = R * D$
$M\_RD = S\_RD$
for $k = 0$ to L_o **do**
  if $k < S\_RD$ **then**
    $S[k] = 1$
  else
    $S[k] = 0$
    $S\_BAK[k] = 0$
  end if
end for
for $n = 1$ to $N - 1$ **do**
  for $k = 0$ to $S\_RD - 1$ **do**
    for $i = 0$ to $M\_RD - 1$ **do**
      $S\_BAK[i + k] = S\_BAK[i + k] + S[i]$
    end for
  end for
  $p\_ptr = S$
  $S = S\_BAK$
  $S\_BAK = p\_ptr$
  for $k = 0$ to L_o **do**
    $S\_BAK[k] = 0$
  end for
  $M\_RD = M\_RD + S\_RD - 1$
end for
$UF = S$
return UF

performance. PM-I circumvents this by directly allocating host memory as pinned, eliminating the need for an intermediate transfer step. Analysis shows that PM-I enhances performance by a factor of 1.76 compared to NR-I.

The conventional GPU workflow involves sequential steps: transferring data from host to device, executing kernels, and transferring results back to host. This serial approach hinders concurrency. To address this, we leveraged CUDA streams to organize multiple CIC processes into workflows, enabling concurrent execution of data transfer and kernel operations. By structuring operations into streams, independent kernels can execute in a pipelined fashion, as illustrated in Figure 3. Our analysis indicates that ODT-I further reduces execution time by 35% to 60% when applied in conjunction with PM-I.

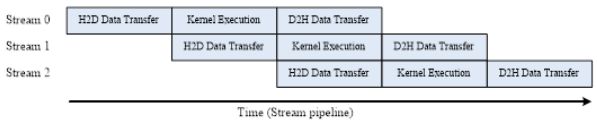

Fig. 3. Overlap data-transfer with kernel execution

Algorithm 6 The Non-recursive CIC kernel

Require: $S, R, D, N, L, L_o, UF, O$

```
__shared__ uf[NT]
if threadIdx.x < L_o then
    uf[threadIdx.x] = UF[threadIdx.x]
end if
__syncthreads()
idx = blockIdx.x * blockDim.x + threadIdx.x
I_x = idx * RD + L_o
if I_x < L then
    tp = S[I_x]
    idx_min = I_x - L_o
    k = I_x - 1, i = 1
    while k ≥ idx_min do
        tp = tp + S[k] * uf[i]
        k = k - 1, i = i + 1
    end while
    O[idx] = tp
end if
return O
```

4. Experimental evaluation

In this segment, we commence by outlining the experimental configuration in a concise manner. Subsequently, we delve into a comparative analysis of the speed enhancements achieved by the Recursive Implementation (R-I) and the Non-Recursive Implementation (NR-I). Following this, we present the experiments conducted to optimize data transfer processes. Lastly, we conclude with an overview of the comprehensive speedup experiments that were undertaken.

4.1. Experimental setup

To thoroughly assess the efficacy of our proposed GPU-accelerated implementation, a comprehensive experimental framework is established. In addition to the GPU-based implementation of the CIC algorithm, we also develop its sequential CPU counterpart and an OpenMP-parallelized version utilizing 32 threads, allowing for a side-by-side performance comparison and validation of the achieved acceleration.

In terms of hardware, the experiments are executed on a high-performance computing (HPC) server, equipped with a potent Intel Xeon E5-2650 v2 CPU featuring dual processors, each comprising 16 cores capable of executing 32 threads concurrently, operating at a base frequency of 2.6 GHz with a core speed of 1.2 GHz. Complementing this setup, a NVIDIA Tesla k20c GPU is utilized, boasting 2496 CUDA cores organized across 13 Streaming Multi-Processors (SMs), 2 GB of dedicated RAM, a processor clock speed of 1059 MHz, a memory clock speed of 900 MHz, and an impressive memory bandwidth of 28.8 GB/s.

On the software side, the experimental setup is anchored on the Windows Server 2008 R2 Enterprise x64 operating system. The CUDA Toolkit version utilized is 7.0, while the OpenMP environ-

ment is configured with version 4.5, ensuring compatibility and optimal performance for the parallel computations involved.

4.2. Speed up experiments of recursive and Non-recursive implementation

To assess and contrast the efficacy of the introduced R-I and NR-I methods, we designed and executed three experimental groups. In the control group, we executed the sequential CPU implementation across a range of problem sizes, meticulously documenting the execution times. Subsequently, we evaluated both the recursive and non-recursive CIC kernels on the GPU, meticulously tracking both the memory management time (inclusive of allocation, transfer, and deallocation durations) and the GPU execution time. Our analysis encompasses both these individual times and the overall execution time, which comprehensively accounts for both memory management and GPU computation phases.

Table 2 presents the execution times and acceleration factors of R-I and NR-I, with millisecond serving as the time unit. Let CPU execution time be denoted as CPU-EXE, GPU kernel execution time as GE, memory management time as MM, and overall execution time as OE (OE = GE + MM). The GPU implementation speedups (SP_G) signify the ratios of CPU-EXE to GE, while the overall process speedups (SP_O) represent the ratios of CPU-EXE to OE. It is important to note that the CIC filter is configured with a stage of 5 and a decimation ratio of 8, and 12 distinct problem sizes are examined. Figure 4 visually depicts the algorithm and overall speedups achieved by R-I and NR-I. As evident from Figure 4, the speedups of R-I and NR-I across varying problem sizes (ranging from 8K to 16M) lie between 18.34 and 174.74, and 35.55 to 449.48 respectively, clearly indicating that NR-I outperforms R-I, with its algorithm speedup being approximately 1.9 to 2.7 times higher.

Table 2. Experimental results on Recursive and Non-recursive CIC implementations

Data size	CPU-EXE	Recursive				Non-recursive			
		MM	GE	SP_G	SP_O	MM	GE	SP_G	SP_O
8K	2.4093	0.2967	0.1313	18.34	5.63	0.4787	0.0678	35.55	4.41
16K	4.8183	0.3092	0.1401	34.39	10.72	0.5032	0.0743	64.84	8.34
32K	9.7390	0.3315	0.1601	60.85	19.81	0.5205	0.0726	134.15	16.42
64K	19.377	0.3860	0.2057	94.20	32.75	0.5720	0.0965	200.86	28.99
128K	39.0872	0.6550	0.2974	131.43	41.04	0.6710	0.1229	318.05	49.24
256K	77.6370	0.8672	0.4905	158.29	57.18	0.8756	0.1897	409.29	72.88
512K	149.9924	1.3108	0.8584	174.74	69.15	1.3578	0.3337	449.48	88.67
1M	269.6571	2.1801	1.5686	171.91	71.93	2.2428	0.6196	435.22	94.21
2M	465.6863	3.8564	2.9829	156.12	68.09	3.9411	1.1273	413.09	91.88
4M	791.5175	7.3286	5.8286	135.80	60.16	7.4728	2.1719	364.44	82.07
8M	1429.0720	14.7735	11.4980	124.29	54.40	14.782	4.2463	336.55	75.10
16M	2749.3795	28.3771	22.8279	120.44	53.69	28.3835	8.3797	328.10	74.79

However, it is noteworthy that the overall speedups are substantially lower than the algorithm speedups. This disparity stems primarily from the substantial contribution of memory management time to the total execution time. As reflected in Table 2, the data transfer times for R-I and NR-I constitute a significant portion of the total execution time, accounting for 55.4% to 69.3% and 77.21% to 87.76% respectively.

Additionally, we have delved into the reasons behind the decrease in performance graphs beyond

512K. In our experiments, the problem sizes were doubled consecutively. If the execution time scaled linearly with the problem size, the execution time for the next size would theoretically be twice that of the previous. We define TR as the ratio of execution times between consecutive problem sizes, with an ideal value of 2. Our findings indicate that the TRs for CPU implementations hovered around 2 but experienced a slight downturn post-512K. Conversely, the TRs for GPU implementations gradually increased from 1 towards 2 and eventually stabilized at approximately 2, signifying enhanced GPU performance with increasing problem sizes. Ideally, if CPU TRs remained constant at 2, the graphs would progressively rise and stabilize. However, the slight decline in CPU TRs beyond 512K (implying improved CPU performance post-512K, contrary to the ideal constant) is precisely what accounts for the decline observed in the graphs presented in Figure 4.

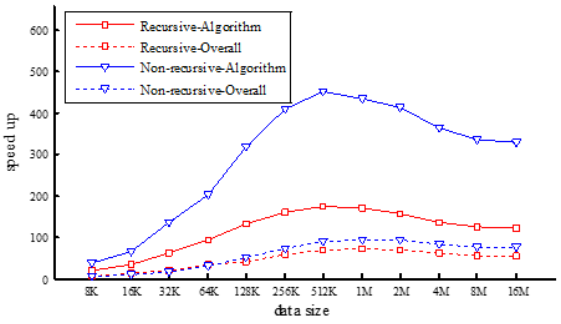

Fig. 4. The speedups of Recursive and Non-recursive CIC implementations

4.3. Experiments of the data-transfer optimization

As aforementioned, data transfer emerges as the primary performance hindrance. To mitigate this, PM-I and ODT-I strategies were implemented. To assess and contrast their impact on accelerating data transfer, three experimental sets were undertaken, with outcomes summarized in Table 3.

Table 3. Experimental results of the data-transfer optimization

Data Size	NR-I	PM-I			ODT-I		
	Time	Time	Reduction (%)	SP	Time	Reduction (%)	SP
8K	0.5376	0.5521	-2.70	0.97	0.2260	57.97	2.38
16K	0.5620	0.5647	-0.47	1.00	0.2251	59.96	2.50
32K	0.6178	0.5565	9.93	1.11	0.2366	61.70	2.61
64K	0.6937	0.5787	16.57	1.20	0.2971	57.16	2.33
128K	0.8299	0.7120	14.20	1.17	0.4473	46.10	1.86
256K	1.2639	0.8137	35.62	1.55	0.5284	58.20	2.39
512K	1.6713	1.2207	26.96	1.37	0.7415	55.63	2.25
1M	2.8349	1.9891	29.84	1.43	1.1737	58.60	2.42
2M	5.0118	3.2258	35.64	1.55	2.0608	58.88	2.43
4M	9.5911	5.7642	39.90	1.66	3.6616	61.82	2.62
8M	19.0699	10.8709	42.99	1.75	6.8241	64.22	2.79
16M	36.8110	20.8924	43.24	1.76	13.1794	64.20	2.79

This table outlines the comprehensive execution times (encompassing memory management and

GPU processing) for NR-I, PM-I, and ODT-I across varying problem sizes. It also presents the percentage reductions in execution time and speedups achieved by the optimized data transfer implementations (PM-I and ODT-I). The data in Table III underscores that for larger problem sizes (exceeding 128K data points), GPU-based implementations yield substantial acceleration gains. Compared to NR-I, the optimized methods, PM-I and ODT-I, achieve noteworthy improvements, with PM-I reducing execution time by over 43.24% and ODT-I by over 64.22%. Specifically, PM-I achieves a 1.76x speedup, while ODT-I attains a 2.79x speedup. These findings affirm that the adopted data transfer optimization techniques effectively bolster practical performance.

4.4. Comprehensive analysis of overall acceleration

To assess the comprehensive speedup achieved by CIC, we executed three sets of experiments across varying problem sizes. The total execution time encompasses kernel execution, data transfer, and additional processing durations. To benchmark the overall speedup of CIC, we initially presented a sequential CPU implementation as a comparative baseline. Nevertheless, acknowledging that a GPU implementation inherently surpasses a sequential CPU counterpart, it is imperative to broaden the comparison scope beyond just a sequential algorithm. Thus, we also implemented an OpenMP version of CIC, enhancing the optimization analysis's significance. Essentially, both GPU and OpenMP leverage the SIMD (Single Instruction, Multiple Data) paradigm. While a GPU can spawn thousands of threads, each concurrently updating a vector element, OpenMP's thread allocation is constrained by the physical processor's capabilities. Given the Intel Xeon E5-2650 v2 CPU's 16 cores and 32 threads, we maximized the parallel region's thread count to 32, ensuring optimal CPU utilization.

Table 4. The overall execution times and speedups

Data Size	CPU-Time	OpenMP#32		ODT-I		
		Time	SP-CPU	Time	SP-CPU	SP-OMP
8K	2.4093	0.7543	3.19	0.2260	10.66	3.34
16K	4.8183	0.8573	5.62	0.2251	21.41	3.81
32K	9.7390	1.1014	8.84	0.2366	41.16	4.66
64K	19.377	1.4552	13.32	0.2971	65.22	4.90
128K	39.0872	2.0521	19.05	0.4473	87.38	4.59
256K	77.6370	3.2156	24.14	0.5284	146.93	6.09
512K	149.9924	5.2231	28.72	0.7415	202.28	7.04
1M	269.6571	9.5148	28.34	1.1737	229.75	8.11
2M	465.6863	18.0432	25.81	2.0608	225.97	8.76
4M	791.5175	34.8946	22.68	3.6616	216.17	9.53
8M	1429.0720	68.412	20.89	6.8241	209.42	10.03
16M	2749.3795	134.7377	20.41	13.1794	208.61	10.22

Table 4 details the comprehensive execution times and acceleration factors across varying problem sizes. Each row showcases the experimental problem size, alongside the execution times for the CPU (CPU-time), OpenMP (with 32 threads) implementation and its respective speedup vis-à-vis the CPU, as well as the GPU execution time and its speedups compared to both the CPU and OpenMP (32 threads) implementations. Figure 5 visually depicts these speedups, highlighting a discernible trend: the overall speedup generally escalates with an increase in problem size, peaking at over

229.75.

Concurrently, the OpenMP implementation, when compared to the sequential CPU version, demonstrates a notable acceleration effect, achieving maximum speedups exceeding 28.72. Nevertheless, the GPU implementation, when juxtaposed against the OpenMP (32 threads) variant, emerges as the more potent accelerator, exhibiting speedups that far surpass those of OpenMP, ranging approximately from 3.34 to 10.22 times. These experimental findings underscore the substantial acceleration gains afforded by the GPU implementation.

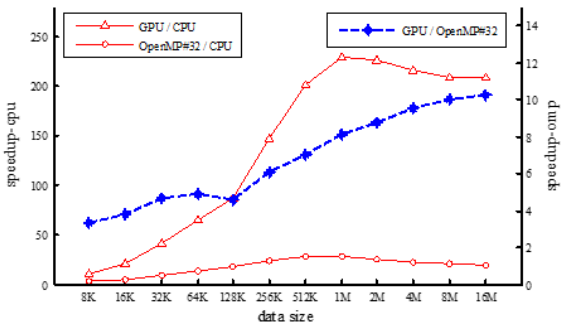

Fig. 5. The overall speedups compared with CPU and OpenMP

5. Conclusions

The CIC decimation filter is an important component which is extensively used in DSP, to enable the software CIC decimation filter to real-time processing the massive digital signal data, two kinds of GPU based implementations are subsequently proposed: R-I and NR-I. Benefiting from the tremendous computational power of GPU, the GPU based CIC implementations gain huge performance improvements, the algorithm speedups of the R-I and NR-I achieve more than 174.74 and 449.48. The speedup of NR-I is about 1.9-2.7 times than the corresponding recursive one, it validates that the Non-recursive structure based CIC which proposed in this paper can reduce the computational burden in much degree and achieve more performance. As data-transfer has become the overall performance bottleneck, the PM-I and ODT-I were applied for further optimization, the results confirmed that the data-transfer optimization approaches effectively enhance the performance in practice. To assess the overall speedup of the GPU-based CIC, both the sequential CPU implementation and the OpenMP implementation were used for comparison. The GPU implementation achieves a speedup of over 229.75 times compared to the CPU version. When compared to the OpenMP implementation, which uses 32 threads, the GPU version demonstrates even greater acceleration, with speedups ranging from approximately 3.34 to 10.22 times. These findings lead to the conclusion that the GPU-based implementation and optimization of software CIC offer substantial performance improvements over the CPU-based approach, particularly in scenarios that are both time- and data-intensive.

Acknowledgements

This work was supported in part by National Natural Science Foundation of China (Grant No. 62362049, 62402205, 62262023), Natural Science Foundation of Jiangxi Province of China (Grant No. 20232BAB212009), the Key Laboratory of Data Protection and Intelligent Management, Ministry of Education, Sichuan University (Grant No. SCUSAKFKT202307Y), the Jiangxi Provincial

Key Laboratory of Data Security Technology(Grant No. 2024SSY03181), the Finance Science and Technology Special "Contract System" Project of Jiangxi Province (Grant No. ZBG20230418014).

References

[1] N. B. Abid and C. Souani. Sdr-based transmitter of digital communication system using usrp and gnu radio. In *Proceedings of the 8th International Conference on Sciences of Electronics, Technologies of Information and Telecommunications (SETIT'18), Vol. 2*, pages 373–381. Springer, 2020. https://doi.org/10.1007/978-3-030-21009-0_36.

[2] V. Awasthi and K. Raj. Study on application of hardware efficient cic compensation filter in narrow band filtering. *Novel Perspectives of Engineering Research*, 2(9):16–28, 2021. https://doi.org/10.9734/bpi/nper/v2/1986C.

[3] H. Blume, J. von Livonius, L. Rotenberg, T. G. Noll, H. Bothe, and J. Brakensiek. Openmp-based parallelization on an mpcore multiprocessor platform–a performance and power analysis. *Journal of Systems Architecture*, 54(11):1019–1029, 2008. https://doi.org/10.1016/j.sysarc.2008.04.001.

[4] Y. S. Choi, H. S. Seo, and Y. B. Kim. Evaluation of gpu computing capacity for all-in-view gnss sdr implementation. *Journal of Positioning, Navigation, and Timing*, 12(1):75–81, 2023. https://doi.org/10.11003/JPNT.2023.12.1.75.

[5] S. Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Newnes, 2012.

[6] D. Datta and H. S. Dutta. High efficient polyphase digital down converter on fpga. *Circuits, Systems, and Signal Processing*, 40(11):5787–5798, 2021. https://doi.org/10.1007/s00034-021-01749-y.

[7] D. Datta and H. S. Dutta. Cic decimation filter implementation on fpga. *Journal of The Institution of Engineers (India): Series B*, 104(1):85–90, 2023. https://doi.org/10.1007/s40031-022-00840-5.

[8] C. Diouf, G. J. Janssen, H. Dun, T. Kazaz, and C. C. Tiberius. A usrp-based testbed for wideband ranging and positioning signal acquisition. *IEEE Transactions on Instrumentation and Measurement*, 70:1–15, 2021. https://doi.org/10.1109/TIM.2021.3065449.

[9] C. Gan and X. Li. Improved cic decimation filter on software defined radio. In *Proceedings of the 2021 9th International Conference on Communications and Broadband Networking*, pages 232–238, 2021. https://doi.org/10.1145/3456415.3456453.

[10] E. Hogenauer. An economical class of digital filters for decimation and interpolation. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 29(2):155–162, 1981. https://doi.org/10.1109/TASSP.1981.1163535.

[11] J. Kim, S. Hyeon, and S. Choi. Implementation of an sdr system using graphics processing unit. *IEEE Communications Magazine*, 48(3):156–162, 2010. https://doi.org/10.1109/MCOM.2010.5434388.

[12] R. K. Kodali, L. Bopppana, and S. R. Kondapalli. Ddc and duc filters in sdr platforms. In *2013 15th International Conference on Advanced Computing Technologies (ICACT)*, pages 1–6. IEEE, 2013. https://doi.org/10.1109/ICACT.2013.6710526.

[13] R. Latha, C. Venkatesan, A. Suhas, and T. Thamaraimanalan. Fpga implementation of polyphase cic based multistage filter for digital receivers. In *2022 8th International Conference on Advanced Computing and Communication Systems (ICACCS)*, volume 1, pages 1987–1991. IEEE, 2022. https://doi.org/10.1109/ICACCS54159.2022.9785005.

[14] W. Li, C. Tang, S. Vishwakarma, K. Woodbridge, and K. Chetty. Design of high-speed software defined radar with gpu accelerator. *IET Radar, Sonar & Navigation*, 16(7):1083–1094, 2022. https://doi.org/10.1049/rsn2.12244.

[15] S. Mhaske, D. Uliana, H. Kee, T. Ly, A. Aziz, and P. Spasojevic. A 2.48 gb/s qc-ldpc decoder implementation on the ni usrp-2953r. *arXiv preprint arXiv:1505.04339*, 2015. https://doi.org/10.48550/arXiv.1505.04339.

[16] W. Plishker, G. F. Zaki, S. S. Bhattacharyya, C. Clancy, and J. Kuykendall. Applying graphics processor acceleration in a software defined radio prototyping environment. In *2011 22nd IEEE International Symposium on Rapid System Prototyping*, pages 67–73. IEEE, 2011. https://doi.org/10.1109/RSP.2011.5929977.

[17] D. E. T. Romero. Simplifying zero rotations in cascaded integrator-comb decimators [tips & tricks]. *IEEE Signal Processing Magazine*, 40(3):50–58, 2023. https://doi.org/10.1109/MSP.2023.3236772.

[18] A. H. N. Sabet, Z. Zhao, and R. Gupta. Subway: minimizing data transfer during out-of-gpu-memory graph processing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020. https://doi.org/10.1145/3342195.3387537.

[19] C. Wang, X. Wang, X. Cui, G. Liu, and M. Lu. Efficient chip-shape correlator implementation on a gpu-based real-time gnss sdr receiver. *GPS Solutions*, 26(4):143, 2022. https://doi.org/10.1007/s10291-022-01332-1.

[20] N. Wilt. *The Cuda Handbook: A Comprehensive Guide to GPU Programming*. Pearson Education, 2013.