International Journal for Housing Science and Its Applications
Publish August 6, 2025. Volume 46, Issue 3 Pages 7587-7596

https,//doi.org/10.70517/ijhsa463651

Novel non-recursive accelerated cascade integrator filter optimization design based on GPU parallel computing

Yanhao Guan[1], Yi Lu[1] and Guolin Shao[1,*]
[1] School of Software, Nanchang University, Nanchang, Jiangxi, 330031, China
Corresponding authors: (e-mail: shaoguolin@163.com).

Abstract With the development of computer technology, the traditional single-threaded CPU computing has been difficult to meet the needs of large-scale data processing, and GPU has become the key technology for optimizing filtering algorithms by virtue of its powerful parallel computing capability. In this paper, we propose a new non-recursive accelerated cascade integrator filter design method based on multi-GPU parallel computing optimization, which adopts the CUDA programming model and synchronous batch normalization technique to make full use of the GPU parallel computing architecture to improve the filter performance. The method designs a four-stage optimization process: firstly, the reference frame image is stored into the GPU global memory; secondly, the adaptive correlation function matrix is calculated and stored into the shared memory; then the filter coefficients are solved by the three-step method of LU decomposition, forward substitution and backward substitution; and finally, the reference frame interpolation calculation is completed in the GPU. The performance test shows that the algorithm in this paper is accelerated up to 119.36 times compared with the traditional CPU method and SIRP+CU method in planar filter computation; In X-ray dynamic micro-CT reconstruction, the algorithm in this paper achieves a speedup ratio of 107.58 over the conventional CPU method when processing 1500 frames of projection data; In the radar clutter simulation, the acceleration ratio reaches 183.14 when processing 1 × 107 data volume, and the average computation time is only 73.15 ms for different data volumes. Experiments demonstrate that the non-recursively accelerated cascaded integrator filter based on GPU parallel computing significantly improves the processing efficiency while guaranteeing the computational accuracy, providing an efficient solution for large-scale computation.

Index Terms GPU parallel computing, non-recursively accelerated cascaded integrator filter, simultaneous batch normalization, CUDA programming, computational efficiency, acceleration ratio

I. Introduction

Real-time communication has become a fundamental technology in modern society, which plays a pivotal role in ensuring the efficiency and stability of information transmission [1]. Especially in the context of the rapid development of 5G, Internet of Things and remote control and other high-tech, the requirements of the community for real-time signal processing are becoming more and more diversified [2]. The new demands are not only reflected in the processing speed, but also in the processing capacity and energy consumption situation [3], [4]. The GPU-based unified computing device architecture (CUDA) parallel computing architecture has a broad application prospect in the field of real-time signal processing [5]. Its powerful parallel processing capability and high-performance computing advantages are widely used in scientific computing, industrial simulation and data analysis [6], [7]. Applying CUDA architecture and programming model to real-time communication signal processing helps to improve the real-time processing, reduce energy consumption, and enhance adaptability [8], [9].

Digital filtering is a frequently used basic computing module in communication systems, and common digital low-pass filter design methods include infinite impulse response (IIR) filters, finite impulse response (FIR) filters, and integral algorithm filters [10]-[12]. Compared to filter design methods that require a large number of convolutions and operations, integral filtering can be used to complete the computation with few sampling points when realizing a low-pass filter [13], [14]. Among them, the cascade integral filter has a simple and standardized structure, and is a very simple and effective anti-alias filtering unit in high-speed extractors is very suitable for implementation on field-programmable gate arrays (FPGAs) with strong real-time and parallel processing capabilities [15]-[17]. However, the mutual suppression of its stopband attenuation and passband ripple limits its filtering performance [18]. Meanwhile, known improved integral filters either reduce the passband attenuation or increase the stopband attenuation, or simultaneously reduce the passband attenuation and increase the stopband attenuation, but occupy more hardware logic resources [19]-[21]. Therefore, it is important to study the optimal design of new cascaded integrator filters based on CUDA architecture.

Computer technology plays a crucial role in modern science and engineering, in which high-performance computing has become a key means to solve large-scale complex problems. The traditional CPU serial computing model shows obvious performance bottlenecks in the face of massive data processing, making it difficult to meet the demands of real-time or near real-time processing. Especially in image processing, medical imaging, radar signal processing and other application scenarios, the computationally intensive tasks put forward extremely high requirements on hardware processing capabilities. In recent years, GPUs have become an important tool in the field of high-performance computing by virtue of their highly parallelized processing architecture. Compared with CPUs, GPUs have thousands of computational cores and are capable of executing a large number of threads with the same instructions at the same time, making them ideal for data-parallel computing tasks. As a fundamental component in the field of signal processing, filters are widely used in noise suppression, signal extraction and enhancement. Among them, non-recursively accelerated cascade integrator filters are widely used in multiple fields due to their excellent frequency response characteristics and phase linearity. However, the computational complexity of such filter algorithms is high, and the filter coefficients need to be computed independently for each pixel point in the traditional implementation, resulting in a heavy computational burden. With the continuous expansion of data size, how to improve the computational efficiency of non-recursively accelerated cascade integrator filters has become a hot research topic. Distinguished from the traditional CPU optimization method, GPU parallel computing provides a new idea for filter performance improvement. By reasonably dividing the task granularity and utilizing the thread-level parallelism of GPU, the computation time can be significantly reduced. However, the GPU programming model is complex, and the memory access pattern has a significant impact on the performance, so how to fully utilize the potential of GPU computing still faces many challenges. In addition, in a multi-GPU environment, the problem of data synchronization and load balancing among devices also needs to be properly solved.

Based on the above analysis, this study proposes a novel non-recursive accelerated cascade integrator filter optimization design method based on GPU parallel computing. The study first analyzes the parallel computing model in multi-GPU environment and explores the characteristics and applicable scenarios of different architectures of shared and distributed systems. Secondly, the synchronized batch normalization technique is introduced to solve the problem of synchronizing the statistical information of data among multi-GPUs to ensure the accuracy of computation. Then, the CUDA programming model is studied in depth to analyze the thread organization structure and various types of memory characteristics to lay the foundation for the optimization algorithm. On this basis, the four-step non-recursive accelerated cascade integrator filter parallel algorithm is designed to maximize the advantages of GPU parallel computing by reasonably arranging the computational tasks and data access modes. Finally, the performance of the algorithm is verified by three typical applications, namely, planar filter computation, X-ray dynamic micro-CT reconstruction and radar clutter simulation, and compared with the traditional CPU method and the existing GPU acceleration scheme to evaluate the acceleration effect and computational accuracy.

II. Parallel computing in a multi-GPU environment

II. A.Application of multiple GPUs in different systems

A multi-GPU shared system is one that integrates multiple GPUs within a single system, these GPUs connect and communicate with each other through the memory (RAM) in the system, but in general it is not possible to connect multiple GPUs together directly, this requires the use of some techniques to accomplish the connection between GPUs, such as SLI.

Distributed system multi-GPU means that each system is assembled with its own independent GPU, these GPUs are connected together through the network, the advantage of doing so is that there is no need to share the system resources between each GPU, in this way, each GPU can work individually, which can greatly improve the efficiency of GPU use.

II. B.Multi-GPU Parallel Computing Models

The strategy of parallel computing [22], [23] is to increase the speed and processing power and reduce the problem solving time by problem decomposition and allocation to multiple processors for simultaneous computation. Multi-GPU architecture using shared systems can effectively reduce the computational overhead, while multi-GPUs in distributed systems are equivalent to GPU clusters, which require a lot of overhead on the network, as well as high requirements on network transmission latency.

The traditional GPU parallel computing model can be represented as follows:

$$t_{total} = t_{CPU} + t_{CPU-COMM} + t_{GPU} + t_{GPU-COMM} \tag{1}$$

The t_{total} table represents the total time required to complete the computation, which includes t_{CPU}, which is used to represent the time consumed by the CPU execution, and $t_{CPU-COMM}$, which is the time required for the data exchange transfer within the CPU. In a shared system environment, the time spent when multiple GPUs need to transfer data to random access memory (RAM) is labeled as t_{memcpy}. Comparatively, in a distributed system, the time consumed when data is transferred over a network is denoted as $t_{network}$. Together, these components affect the speed of computation. As in Eq:

$$t_{CPU-COMM} = \begin{cases} t_{memcpy} \\ t_{network} \end{cases} \tag{2}$$

where t_{GPU} denotes the GPU execution time and $t_{GPU-COMM}$ denotes the time required to transfer data between CPU and GPU. Parallel computing in a multi-GPU environment can be divided into several key steps:

(1) Data loading: first, the dataset to be computed is read from the hard disk into the memory.

(2) Data transfer: Next, the dataset is efficiently transferred from memory (RAM) to multiple GPUs via the PCI-E bus.

(3) Independent computation: After receiving the data, each GPU will independently perform its assigned computation task.

(4) Result Integration: After completing the computation, each GPU returns its respective results to the CPU and further writes them to the hard disk for subsequent processing or analysis.

II. C.Simultaneous batch normalization

Synchronized Batch Normalization (SyncBN) [24] is a normalization technique for deep learning models that, unlike traditional batch normalization, ensures that the same mean and variance are used to normalize the data on each device by synchronizing the statistical information on all GPUs. In standard batch normalization, each device calculates its own mean and variance in each batch and then uses these statistics to normalize the data.

Fig. 1 shows a diagram of the computational process of BN synchronized across devices. Assuming batch_size=2, the mean and variance computed by each GPU are specific to these two samples.A remarkable property of BN is that as the batch size increases, the computed mean and variance become closer and closer to those of the whole dataset, which helps to improve the model. The input for this study is the small batch $B = \{x_1, x_2, \ldots, x_m\}$, where μ is calculated from $\frac{1}{m}\sum_{i=1}^{m} x_i$, which denotes the mean of the small batch B. The σ^2 is calculated from $\frac{1}{m}\sum_{i=1}^{m}(x_i-\mu)^2$ and represents the small batch variance. Meanwhile, the input x_i is normalized in this study, i.e., $\frac{x_i-\mu}{\sqrt{\sigma^2+\grave{o}}}$. The output $y_i = \gamma\hat{x}_t + \beta = BN_{\gamma,\beta}(x_i)$, where μ, σ^2 are obtained statistically during forward propagation and γ, β are obtained by training during the back propagation process. When multiple GPUs are used for training, the mean and variance of each BN layer's inputs on all devices are calculated. If both GPU1 and GPU2 get two feature layers each, then the two GPUs calculate the mean and variance of four feature layers in total, which can be viewed as batch_size = 4. If SyncBN is not used, and instead, each device calculates the mean variance of its own batch of data, the effect is consistent with that of a single GPU, and merely improves the training speed. If SyncBN is used, the effect is somewhat improved, but some of the parallel speed is lost.

III. CUDA-based filter optimization design

In this paper, a well-supported CUDA parallel platform is used as the software development environment. On this basis, the thread organization and memory model of CUDA are introduced, and a novel non-recursive accelerated cascaded integrator filter optimization algorithm for GPU-BN parallel computing is designed.

III. A. CUDA Thread Organization and Memory Model

III. A. 1) CUDA core functions and thread hierarchy

In CUDA programming [25], since there is both host-side and device-side executed code, it is necessary to distinguish between them.CUDA recognizes this process by adding function type qualifiers. The three main function type qualifiers are __global__, __device__, and __host__.

The qualifier __global__ indicates that the core function is executed on the device side and called from the host side.

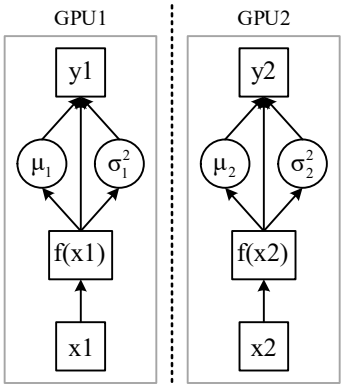

Figure 1: Calculation process of BN synchronization among different devices

The return type must be void, variable parameters are not supported, and it cannot be a class member function. The core function defined by __global__ is asynchronous, that is to say, the Host side will not wait for the end of the core function execution will automatically execute the next step, so it is also necessary to add synchronization functions, generally use the __synchronize() function to achieve process synchronization. The qualifier __device__ means that the core function is executed on the Device side and can only be called from the Device side. Therefore, this qualifier cannot be used in conjunction with __global__.

The qualifier __host__ means that the kernel function is executed on the Host side and can only be called from the Host side, and can usually be omitted. __host__ cannot be used with __global__, but can be used with __device__.

From a threading perspective, a core function is a function that is executed by each thread of CUDA. When a core function is executed on the Device side, it actually starts many threads. All the threads started by a core function are called a thread grid, which belongs to the first level of CUDA thread organization, and threads on the same thread grid are allowed to share the global memory space. Thread grids can be divided into thread blocks, and a thread block can contain multiple threads, which correspond to the second and third levels of CUDA thread organization, respectively.

III. A. 2) CUDA Memory Modeling

The rationality of memory allocation is very important to accelerate the performance of CUDA programs, so it is necessary to understand the common characteristics of each type of CUDA memory. A schematic of the CUDA memory model is given in Figure 2 to facilitate the analysis of the hierarchical progression between individual memories.

In the figure, the Host-side memory on the leftmost side realizes high-speed data exchange with the Device-side memory through the PCIe bus.Register and Local Memory are owned by the threads, and their lifecycles are equal to the bound threads.Shared Memory can be used for the communication between all threads in a Block, and its lifecycle is also equal to the lifecycle of the bound threads. Shared Memory can be used for communication between all threads in a Block, and its lifespan is also equal to that of the bound threads. All threads in the Thread Grid support access to Global Memory, Constant Memory, and Texture Memory, all of which have the same lifecycle as the application.

The following briefly describes their respective memory characteristics.

(1) The largest and most commonly used graphics memory in the GPU is Global Memory, which in a sense is quite similar to the CPU's system memory. Optimizing access to global memory in programming is critical to improving the data throughput of global memory.

(2) Registers are the top-performing memory on GPUs, and this type of memory is typically used to store private variables in threads. In addition, only the bundle of threads that are running in the stream multiprocessor (SM) is eligible for register access, but it also limits the number of registers that each thread can use to a certain number.

(3) For each thread, local memory is privatized, and private memory is typically used to store register overflow data. The local memory is located in the off-chip memory of the GPU and has a more general performance, characterized by high latency and low bandwidth.

(4) Shared memory requires the introduction of the identifier __shared__ to complete the declaration, and is characterized by low latency and high bandwidth. Its performance is much higher than that of global memory and private memory, and therefore it is often used to store more frequently used data. During the execution of a core function, the data needed in global memory is usually copied to shared memory, and then the result is copied back to global memory after the program is executed.

(5) Constant memory requires the introduction of the identifier __constant__ to complete the declaration. It is slower than shared memory and can only be read-only, and thus is rarely used.

(6) Texture memory is mainly used for 2D images, and like constant memory, it supports only read-only.

It is worth noting that CUDA is not a visual programming model, and developers must complete a proper analysis of the code based on GPU features and functionality in order to strike a balance between data transfer and memory access performance improvements.

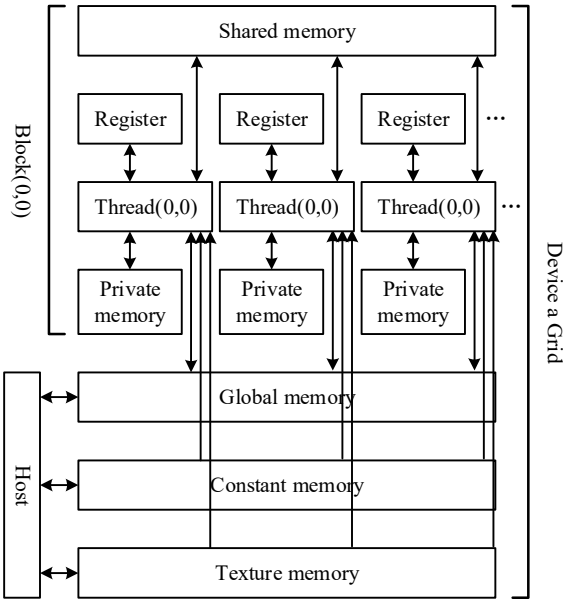

Figure 2: Schematic diagram of CUDA Memory Model

III. B. Parallel optimization of non-recursively accelerated cascade integrator filters

Since the non-recursive accelerated cascade integrator filter is based on the content of the image elements, there is a different filter for different subpixel points, which means that the prediction and solution of the filter coefficients need to be performed for each point.

Since each sub-pixel point performs the filter computation independently and there is no data exchange between them, this section utilizes the fusion of the multi-GPU parallel computing model described above with the simultaneous normalization method for the parallel optimization of the filters. The specific implementation is as follows:

(1) Firstly, the decoded reference frame image is deposited into the GPU's Global Memory at one time for subsequent filter coefficient computation. Since the transfer of values between the GPU's graphics memory and the host memory consumes a considerable amount of time, the time spent on transferring the data during the various storage periods within the GPU's graphics memory is much smaller and can be ignored. So the whole frame image is passed into the graphics memory and read from Global Memory each time a calculation is performed.

(2) Read the pixel information from Global Memory to calculate the required Adaptive Correlation Function Matrix $N \times N\_Matrix$ and Intercorrelation Function Matrix N_Vector for each subpixel point. Store the results in the Shared Memory of each Block.

(3) For each threaded Block, solving the coefficients is actually solving the equation:

$$N \times N\_Matrix * FilterCoef = N\_Vector \tag{3}$$

The corresponding filter coefficients can be computed in three steps. First the LU decomposition of $N \times N\_Matrix$ with $N \times N_{Matrix} = LU$, the above equation becomes:

$$LU * FilterCoef = N\_Vector \tag{4}$$

Then order:

$$U * FilterCoef = Y \tag{5}$$

Solving Equation $LY = N\_Vector$ using forward substitution; then solving the equation using backward substitution yields the filter coefficients.

(4) After the filter coefficients are obtained it is necessary to interpolate the reference frame. Both FilterCoef and reference frame data are in the GPU and the sampling processes are independent of each other and thus can also be computed in the GPU.

The interpolation of each sub-pixel point is a matrix multiplication process, because the product of the width of the filter matrix and the width of the reference frame samples is less than 512, so it can be done in a single threadBlock, each time a row of A and a column of B are loaded (A is the reference frame samples matrix, and B is the filter coefficients lifted), and the corresponding elements of A and B are done once for multiplication and addition. Finally, the result is passed to memory via the CUDAAPI function.

IV. Numerical examples and performance analysis

In this section, several common problems in real-world applications of non-recursively accelerated cascaded integrator filters are computed to verify the acceleration effect of multi-GPU-BN parallel algorithms. The computing platform in the case is the GPU node of the Wave Computing Cluster, which has an Intel Xeon Gold 6248R CPU@2.0GHz CPU, 1 TB of RAM, and two Tesla V100 NVLink GPU cards with Volta architecture as GPUs. The bit width is 4096 bits, the video memory is 32GB, and the number of CUDA cores is 5120.

IV. A. Planar Filter Resource Consumption and Acceleration Rate Measurement

The case in this section is a planar filter with an operating band of 2.5 GHz-5 GHz, which is fed by two air coaxial lines with inner and outer diameters of 0.75 mm and 1.50 mm, respectively. The excitation source is a coaxial wave port and the excitation signal is a modulated Gaussian pulse from 2.5 GHz to 3.5 GHz. The outer wall and inner cavity of the filter are made of PEC, the cavity filling medium is air, and the truncated boundary of the entire computational domain is the PEC boundary.

For dielectric filters, relatively more iteration time steps are often required to achieve convergence in order to obtain accurate numerical simulation results, so the model uses sequential excitation with 1,000,000 time steps per port iteration.To verify the acceleration effect of the algorithms in this paper, the computation time of the different acceleration schemes is compared, firstly for the conventional CPU method, and then for the fine-grained and coarse-grained optimization of the SIRP-CU heterogeneous acceleration method is used to calculate the S-parameters of this model at 2.5GHz-3.5GHz.

Fig. 3 shows the results obtained for the calculation of the S-parameters of the planar filter, and Table 1 gives the resource consumption and the speedup multiplier of the different methods used to calculate this model. The graphical results show that the filter model has a good filtering effect in the 2.8GHz-3.1GHz band, and the GPU-BN algorithm in this paper can be used to obtain results that match the traditional CPU algorithm in the calculation. In addition, the SIRP+CU algorithm achieves an acceleration factor of 55.55 compared to the traditional CPU algorithm, and the algorithm in this paper achieves an acceleration factor of 119.36 compared to the traditional CPU algorithm, and the numerical results show that the algorithm in this paper has a higher acceleration effect compared to GPU-accelerated algorithms in the related literature.

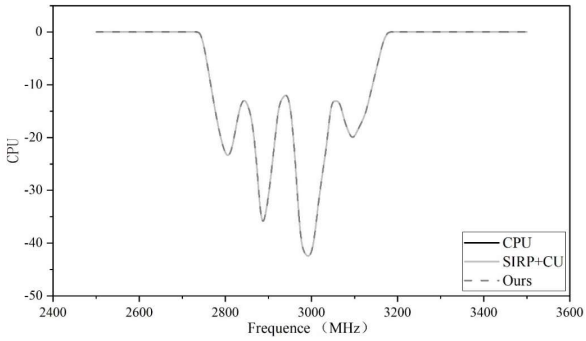

Figure 3: The S parameter of the plane filter is calculated

Table 1: Resource consumption and acceleration multiple of different methods

Method	Time (min)	Consuming memory (GB)	Consumption visualization (GB)	Accelerating speed
CPU	29160.6	5.28	1.43	1
SIRP+CU	524.9	4.86	2.66	55.55
Ours	244.3	3.17	3.91	119.36

IV. B. Rapid Reconstruction Comparison Based on X-ray Dynamic Micro-CT

In this section, the algorithm is applied to fast reconstruction of X-ray dynamic micro-CT to test the performance of the GPU-based algorithm. The experimental data required for the test comes from a CT data acquisition system with 20 Hz temporal resolution and a data pixel size of 1024 × 1024.

The projection data with a pixel number of 1024 × 1024 are reconstructed using the conventional CPU method, SIRP+CU, and the algorithm based on multi-GPU+BN parallelism, respectively, and the reconstruction times of the several schemes are compared with CT data of different sizes.

Fig. 4 shows the trend of reconstruction time with CT data size for the above three schemes. From the figure, it can be seen that the CT reconstruction algorithm based on the multi-GPU parallel computing in this paper shortens the running time of CT reconstruction. When the projection data is 1500 frames, the running speed of this paper's algorithm is nearly 1.8 times higher than the traditional CPU reconstruction algorithm.

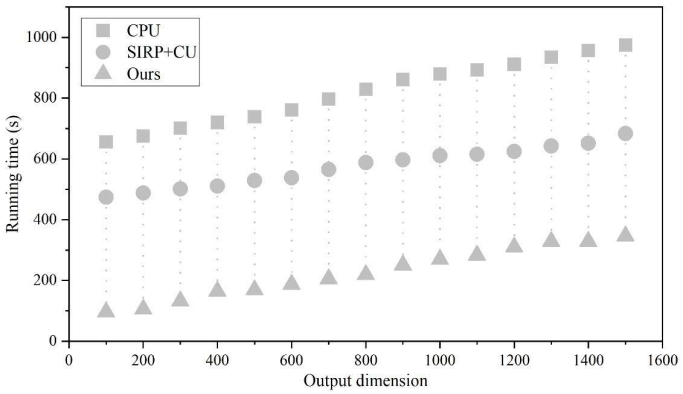

Figure 4: The reconstruction time of the three schemes is changed with CT data

With the increasing data size, the CPU reading hard disk data and local memory and GPU graphics memory data transfer time increases. For different sizes of projection data, the trend of the reconstruction time acceleration ratio of SIRP+CU and the intersecting traditional CUP algorithm of this paper's method with CT data size is shown in Fig. 5.

The reconstruction time acceleration ratio of this paper's algorithm compared to the traditional CPU method is always higher than that of the SIRP+CU algorithm and the traditional CPU method under different sizes of data, and the reconstruction time acceleration ratio of the two algorithms is 43.89 and 107.58 respectively when the data size is 1500.In addition, when the projection data volume is increased to a certain size, the reconstruction time acceleration ratio of the image reconstruction will decrease with the increase of data volume. The main reason for this is that the CT reconstruction algorithm based on GPU parallel computing plays an accelerating effect in the inverse projection computation part, while it does not get accelerated in the projection reading and slice writing parts.

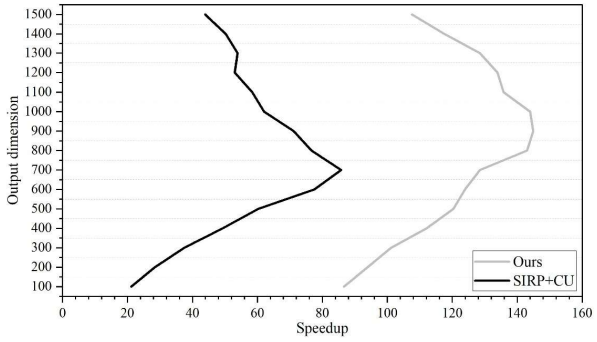

Figure 5: The acceleration ratio is compared to the change curve of the CT data size

IV. C. Numerical results of radar clutter simulation with different methods

In this paper, we use multi-GPU-BN parallel algorithm to simulate radar clutter acceleration experiments, clutter simulation commonly used thread optimization allocation + shared memory method, and will be compared with the fine-grained and coarse-grained optimization of the SIRP-CU method as well as the CPU method of the computation time used, three methods used in the computation time of the comparison results are shown in Table 2. It can be seen that when the data volume is 1.0×10^2, the computation time of this method and the SIRP-CU method are close to 6.40 and 7.01 ms, respectively, but as the data volume increases, the difference between the computation time of the three methods gradually increases, and the average computation time of this method under different data volumes is 73.15 ms, which is less than that of the other two methods.

Table 2: The comparison of the calculation time used in three methods

Data volume	Computation time (ms)		
	CPU method	SIRP+CU	Ours
1×10^2	9.26	7.01	6.40
1×10^3	77.88	20.80	15.90
1×10^4	423.04	35.42	27.83
1×10^5	1905.51	54.52	38.94
1×10^6	3566.50	88.36	59.06
1×10^7	33895.93	533.05	290.76
Average	6646.35	123.19	73.15

Figure 6 shows the acceleration ratio of GPU and CPU convolutional computation obtained from this paper's method and the SIRP-CU method. It can be seen that when the data volume is 1×10^2, the acceleration ratios of both methods are small. When the data volume increases to, 1×10^5, the acceleration ratios of both methods increase, and the computational efficiency is improved. When the data volume reaches 1×10^7, the optimized acceleration ratio of this paper's method is 183.14, and the acceleration ratio of the SIRP-CU method is only 91.46, which is a significant increase in the acceleration ratio compared to the SIRP-CU method, and the computational efficiency is higher. From the above, it can be seen that when CPU is used to generate the clutter data, the computation is more time-consuming under a large amount of data. The computational efficiency improvement is not obvious when using GPU for smaller data volume computation, but GPU parallel computation is more efficient than CPU computation for large amount of data computation.

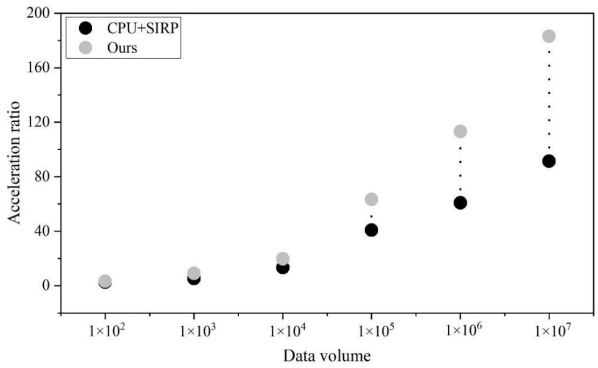

Figure 6: The acceleration ratio of GPU and CPU convolution in the two methods

V. Conclusion

Multi-GPU-BN parallel computing technique brings significant performance improvement for non-recursively accelerated cascaded integrator filter optimization. The numerical example results of planar filters show that the algorithm in this paper obtains 119.36 times speedup compared to the traditional CPU method and SIRP+CU method, and the memory consumption is only 3.17 GB, which is significantly lower than the 5.28 GB of the CPU method. The X-ray dynamic micro-CT reconstruction test proves that when processing 1500 frames of data, the acceleration ratio of this paper's method reaches 107.58, which is better than that of the SIRP+CU method (43.89).The radar clutter simulation experiment further verifies the excellent performance of the algorithm in large-scale data processing scenarios, and the computation time for the data of the magnitude of 1 × 107 is only 290.76ms, with an acceleration ratio of up to 183.14, far exceeding the 91.46 of the SIRP+CU method. Three types of

application tests confirm that GPU parallel acceleration significantly improves processing efficiency on the basis of ensuring computational accuracy. It is worth noting that the acceleration effect is relatively limited when the data size is less than 1×103, but the advantage is obvious when the data volume increases. Future work can further explore memory access optimization and dynamic load balancing strategies to further improve the adaptability and computational efficiency of the algorithm in heterogeneous computing environments.

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] Aguilar, A. H., Bonilla-Robles, J. C., Díaz, J. C. Z., & Ochoa, A. (2019). Real-time video image processing through GPUs and CUDA and its future implementation in real problems in a Smart City. International Journal of Combinatorial Optimization Problems and Informatics, 10(3), 33.

[2] Huang, Y., Li, S., Chen, Y., Hou, Y. T., Lou, W., Delfeld, J., & Ditya, V. (2020). GPU: A new enabling platform for real-time optimization in wireless networks. IEEE Network, 34(6), 77-83.

[3] Nejedly, P., Plesinger, F., Halamek, J., & Jurak, P. (2018). Cuda F ilters: AS ignal P lant library for GPU-accelerated FFT and FIR filtering. Software: Practice and Experience, 48(1), 3-9.

[4] Al-Mouhamed, M. A., Khan, A. H., & Mohammad, N. (2020). A review of CUDA optimization techniques and tools for structured grid computing. Computing, 102(4), 977-1003.

[5] Dehal, R. S., Munjal, C., Ansari, A. A., & Kushwaha, A. S. (2018, October). Gpu computing revolution: Cuda. In 2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN) (pp. 197-201). IEEE.

[6] Georgis, G., Lentaris, G., & Reisis, D. (2019). Acceleration techniques and evaluation on multi-core CPU, GPU and FPGA for image processing and super-resolution. Journal of real-time image processing, 16(4), 1207-1234.

[7] Khujayorov, I., & Ochilov, M. (2019, November). Parallel signal processing based-on graphics processing units. In 2019 International Conference on Information Science and Communications Technologies (ICISCT) (pp. 1-4). IEEE.

[8] Liu, G., Yang, W., Li, P., Qin, G., Cai, J., Wang, Y., ... & Huang, D. (2022). MIMO radar parallel simulation system based on CPU/GPU architecture. Sensors, 22(1), 396.

[9] Kim, Y., & Park, Y. (2019). CPU-GPU architecture for active noise control. Applied Acoustics, 153, 1-13.

[10] Agrawal, N., Kumar, A., Bajaj, V., & Singh, G. K. (2018). Design of bandpass and bandstop infinite impulse response filters using fractional derivative. IEEE transactions on industrial electronics, 66(2), 1285-1295.

[11] Zhao, S., Shmaliy, Y. S., Ahn, C. K., & Liu, F. (2020). Self-tuning unbiased finite impulse response filtering algorithm for processes with unknown measurement noise covariance. IEEE Transactions on Control Systems Technology, 29(3), 1372-1379.

[12] Bamigbade, A., & Khadkikar, V. (2022). A cascaded integral action based filter for distorted conditions. IEEE Transactions on Industrial Electronics, 69(10), 10752-10760.

[13] Gao, M., Yu, Z., Wan, S., & Shao, J. (2022, June). Design of low-power CIC decimation filter based on nibble serial arithmetic. In Journal of Physics: Conference Series (Vol. 2294, No. 1, p. 012008). IOP Publishing.

[14] Jovanovic Dolecek, G., & Fernandez-Vazquez, A. (2024). Improving Passband Characteristics in Chebyshev Sharpened Comb Decimation Filters. Applied Sciences, 14(23), 11421.

[15] Gear, K. W., Sánchez-Macián, A., & Maestro, J. A. (2021). Reduced length redundancy adaptive protection for the cascaded integrator-comb interpolation filter on FPGA. Microelectronics Reliability, 118, 114043.

[16] Zhi, C. H. E. N., & Bin, W. U. (2020). Chebyshev comb compensation filter based on FPGA. Chinese Journal of Quantum Electronics, 37(1), 15.

[17] Bontempi, F., Andriolli, N., Scotti, F., Chiesa, M., & Contestabile, G. (2019). Comb line multiplication in an InP integrated photonic circuit based on cascaded modulators. IEEE Journal of Selected Topics in Quantum Electronics, 25(6), 1-7.

[18] Dudarin, A., Molnar, G., & Vucic, M. (2022). Optimum multiplierless sharpened cascaded-integrator-comb filters. Digital signal processing, 127, 103564.

[19] Geng, Z., Xie, Y., Zhuang, L., Burla, M., Hoekman, M., Roeloffzen, C. G., & Lowery, A. J. (2017). Photonic integrated circuit implementation of a sub-GHz-selectivity frequency comb filter for optical clock multiplication. Optics Express, 25(22), 27635-27645.

[20] Shanthi, G., Kumar, A. S., Phanidra, P., Raj, G. S., Niharika, N., & Kalyani, K. (2022, September). An efficient FPGA implementation of cascade integrator comb filter. In 2022 International Conference on Intelligent Innovations in Engineering and Technology (ICIIET) (pp. 151-156). IEEE.

[21] Cheng, K. C., Chang, S. J., Chen, C. C., & Hung, S. H. (2025). A 94.3 dB SNDR 184dB FoMs 4th-order noise-shaping SAR ADC with dynamic-amplifier-assisted cascaded integrator. IEEE Solid-State Circuits Letters.

[22] Buyu Liu,Wei Song,Mingyi Zheng,Chong Fu,Junxin Chen & Xingwei Wang. (2025). Semantically enhanced selective image encryption scheme with parallel computing. Expert Systems With Applications,279,127404-127404.

[23] Jianqiang Kang,Zhichao Gong,Jing V. Wang,Weihua Chen,Qian Wang,Yaxiang Fan & Bin Huang. (2025). A novel method of parameter identification for electrochemical models of LiFePO4 batteries using parallel computing and multi-objective optimization. Journal of Energy Storage,122,116662-116662.

[24] TuSimple, TuSimple, TuSimple, Computer Science Department,Tsinghua University & TuSimple. (2019). Training Deep Nets with Progressive Batch Normalization on Multi-GPUs. International Journal of Parallel Programming,47(3),373-387.
[25] Helder J. F.Luz,Paulo S. L.Souza & Simone R. S.Souza. (2024). Structural testing for CUDA programming model. Concurrency and Computation: Practice and Experience,36(14).