Control Flow Divergence Optimization by Exploiting Tensor Cores

Weiguang Pang[1,2,3], Xu Jiang[1]*, Songran Liu[4], Lei Qiao[5], Kexue Fu[2,3], Longxiang Gao[2,3], Wang Yi[4,6]
[1] University of Electronic Science and Technology of China, China
[2] Key Laboratory of Computing Power Network and Information Security, Ministry of Education, Shandong Computer Science Center, Qilu University of Technology (Shandong Academy of Sciences), Jinan, China
[3] Shandong Provincial Key Laboratory of Computer Networks, Shandong Fundamental Research Center for Computer Science, Jinan, China
[4] School of Computer Science and Engineering, Northeastern University, China
[5] Beijing Institute of Control Engineering, Beijing, China [6] Uppsala University, Sweden

ABSTRACT

Kernels are scheduled on Graphics Processing Units (GPUs) in the granularity of GPU warp, which is a bunch of threads that must be scheduled together. When executing kernels with conditional branches, the threads within a warp may execute different branches sequentially, resulting in a considerable utilization loss and unpredictable execution time. This problem is known as the control flow divergence. In this work, we propose a novel method to predict threads' execution path before the launch of the kernel by deploying a branch prediction network on the GPU's tensor cores, which can efficiently parallel run with the kernels on CUDA cores, so that the divergence problem can be eased in a large extent with the lowest overhead. Combined with a well-designed thread data reorganization algorithm, this solution can better mitigate GPUs' control flow divergence problem.

1 INTRODUCTION

Graphics Processing Units (GPUs) have revolutionized parallel computing, with their impact spanning across data analysis [2], bioengineering [23], and notably, Artificial Intelligence / Machine Learning (AI/ML) [20], particularly in embedded autonomous systems. At the heart component of GPU architecture, as exemplified by NVIDIA's GPUs, are multiple Streaming Multiprocessors (SMs), each equipped with numerous CUDA cores. These CUDA cores execute threads using a Single Instruction Multiple Data (SIMD) execution style, which is fundamental to achieving substantial computational throughput in GPUs.

However, the SIMD execution model, while efficient, introduces a significant challenge known as warp divergence. In the context of GPUs, a warp refers to a set of threads—typically 32 in modern GPUs—that execute in parallel. When kernels with conditional branches are executed, threads within the same warp may take different execution paths. This divergence, especially pronounced in applications managing complex data structures like trees, graphs, and in solving Partial Differential Equations (PDEs) with varying boundary conditions [6], leads to threads within a warp processing different branches sequentially rather than concurrently. As a result, there is a considerable loss in resource utilization and a rise in unpredictable execution time. This problem leads to severe underutilization of resources and unpredictable execution time. This inefficiency is notably demonstrated by Baghsorkhi et al. [1] in their prefix-scan benchmark, where control flow divergences constituted a substantial fraction of the total execution time.

Addressing warp divergence, researchers have proposed various solutions. Hardware-based approaches [7–9], such as redesigning GPU micro-architectures, while innovative, are impractical for existing GPUs. Conversely, software solutions, despite their adaptability, often struggle with complex control flows or introduce significant computational overhead [4, 12, 26]. This struggle is further exacerbated by the complexity of nested branches in CUDA code and the limited visibility of branch conditions outside user kernels. Consequently, reorganizing thread data, a critical step in addressing warp divergence, becomes a laborious and unpredictable process, particularly when implementing dense prediction models [26].

In recent advancements in GPU architecture, as seen in models like NVIDIA's Orin, CUDA cores continue to grapple with warp divergence, a challenge inherent in SIMD architectures. To address the growing demands of AI/ML computing, Tensor Cores (TCs) have been strategically integrated into the SMs of these GPUs. These TCs, designed specifically for General Matrix Multiplication (GEMM) operations, feature distinct data paths and APIs, setting them apart from traditional CUDA cores. Primarily optimized for tensor operations, TCs efficiently handle neural network computations, effectively bypassing the warp divergence issues prevalent in SIMD-based CUDA cores.

This advancement presents a novel solution to the control flow divergence problem in GPU computing. In this work, we introduce an innovative method that leverages Tensor Cores to optimize control flow divergence. We developed a branch prediction network deployed on Tensor Cores in parallel with the primary processing on CUDA cores while keeping minimum overhead. This network aims to efficiently reduce the computing resource wastage caused by control flow divergence. Additionally, our approach is enhanced by a meticulously designed thread data reorganization algorithm, which significantly reduces control flow divergence in GPU environments. Extensive tests on real hardware platforms, using various benchmarks, show that the proposed method effectively tackles these challenges and significantly outperforms existing methods.

2 PRELIMINARY

2.1 GPU Architecture and Programming Model

An NVIDIA GPU typically consists of multiple SMs, which are the main computing units of the GPU. As shown in Figure 1, each SM

*Corresponding author: Xu Jiang.

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06
https://doi.org/10.1145/3649329.3658462

further organizes its computing resource as smaller units to handle the physical execution of threads dispatched to it, such as CUDA cores for normal arithmetic operations, Double Precision (DP) units for double precision computations in CUDA, along with instruction buffer, warp scheduler, and register files shared among them. Since the Volta architecture [3], NVIDIA GPUs, including those based on the Turing, Ampere, and Hopper architectures [24], have integrated TCs in their SMs. The primary serviceability of TCs is to provide tile-based matrix-matrix computation primitives on register fragments, such that applications formed by multiplication and addition operations could be significantly accelerated[16], e.g., GEMM. In particular, TCs support mixed precision computation in the form of $D_{M\times N} = A_{M\times K} \times B_{K\times N} + C_{M\times N}$, where A and B may have a precision (e.g., FP16) different from C and D (e.g., FP32). Moreover, TCs are made accessible to programmers through dedicated GEMM APIs, such as CUDAWMMA [17], and share register and memory resources with CDs.

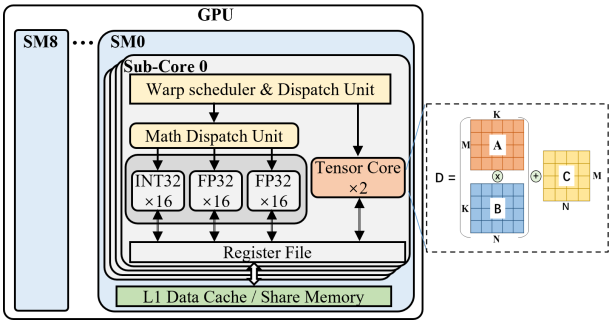


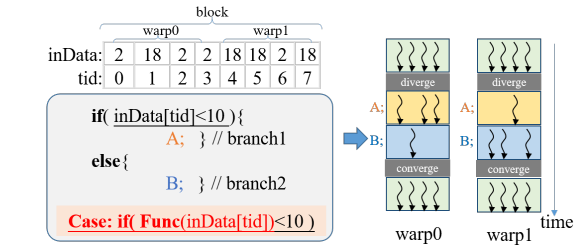
Figure 1: The architecture in the Jetson AGX Xavier GPU.

From a programmer's point of view, a typical GPU program mainly comprises three parts: a segment running on the host CPU, a segment for host/device memory copy, and the GPU kernel (called kernel for short). A kernel is composed of several threads hierarchically organized into groups that will execute on the GPU hardware. A number of threads form a block, which may be further divided into several warps. Each warp consists of a fixed number of threads (typically 32). All threads in a warp execute the same instruction in SIMD style. The number of threads within a block (i.e., block size) and the number of blocks within a kernel (i.e., grid size) must be specified when launching a GPU kernel.

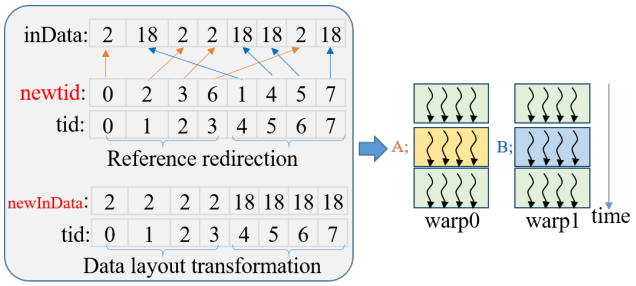
2.2 Scheduling and Control Flow Divergence

At GPU runtime, a kernel is allocated to SMs at the granularity of a block. A block can only be allocated to one SM, and each SM can concurrently process multiple blocks. The number of blocks that can be executed by an SM in parallel is determined by the block's resource requirements, the SM's computational resources, and architectural constraints. Excess blocks are queued and dynamically allocated to available SMs.

Inside these blocks, threads may not execute simultaneously despite being on the same SM. The warp, the smallest scheduling unit on an SM, is managed by warp schedulers. These schedulers allocate data to warps, select ready warps for execution, and switch among them based on resource availability and needs. Warp scheduling policies, such as Loose Round-Robin (LRR) or Greedy-Then-Oldest (GTO), dictate the instruction execution of each warp per cycle (as detailed in [10]). Blocks composed of less than 32 threads lead to the inclusion of null threads, causing resource inefficiency. Hence,



(a) Program with divergence.



(b) Input thread data regroup

Figure 2: Thread-data regroup with different branch paths via reference redirection and data layout transformation. Warp size is assumed to be 4.

a block size divisible by 32 is recommended to optimize resource utilization.

When warps are scheduled on an SM, threads execute instructions in the SIMD manner. Each cycle processes one instruction across all threads in a warp. Parallel execution occurs when all threads execute the same instruction, but warp divergence arises with different instructions, leading to thread serialization. This divergence can severely impact performance, potentially causing a slowdown by up to 32× if each thread in a warp executes a unique instruction [6].

Warp divergence is an often unavoidable aspect of GPU computing, particularly in programs with control flow involving conditional statements like if-else, while-do, and switch-case. This divergence stems from varying thread-data[1], leading to divergent execution paths in kernels. For example, the program depicted in Figure 2. (a) demonstrates how different input data can result in threads following different branches. In a scenario where this program is implemented as an eight-thread kernel block on a GPU with four threads per warp, it splits into two warps, warp0 and warp1. In warp0, processing an 'if' statement may lead to only three threads executing branch *A*, while one remains idle. Similarly, in warp1, during an 'else' statement, perhaps only one thread executes branch *B*, with the rest idle. The GPU's scheduler does not switch to a different warp unless all threads in the current warp are idle. This limitation can cause the SM to operate at half its computing capacity, leading to significant performance degradation and unpredictable kernel execution time. Additionally, when different conditional branches are governed by complex functions, predicting the execution path of threads based on input data without pre-executing the function poses a substantial challenge. This work aims to optimize control flow divergence, enhancing both resource utilization and the predictability of execution time in GPU kernels.

[1]Thread-data refers to the input data for every CUDA thread.

2.3 Related Work and Discussion

Control flow divergence in GPUs is addressed using hardware and software approaches. Hardware strategies primarily involve creating new GPU micro-architectures. For example, Fung et al. [7] developed the Dynamic Warp Formation (DWF) method, which consolidates threads executing identical paths into a single warp, and Meng et al. [15] introduced Dynamic Warp Subdivision (DWS) to enhance branch divergence handling by enabling concurrent execution of distinct control flows. Nonetheless, such techniques are geared towards new GPU designs and are incompatible with current models.

On the software side, reorganizing thread input data is a prevalent tactic. The two leading techniques are Reference Redirection (RR) and Data Layout Transformation (DLT) [26], with RR reassigning threads based on data indices and DLT modifying the data structure. Zhang et al. [26] formulated Thread Data Remapping (TDR) on the CPU, which, coupled with a CPU-GPU pipeline, reduces TDR's overhead. Notably, GPU-TDR, incorporated within the kernel, facilitates real-time data remapping. Subsequent innovations [12, 13] have refined GPU-TDR by accounting for the input thread data's influence on branch divergence. However, GPU-TDR's efficacy diminishes with dynamic branches or when in-kernel variables alter. Although enhancements, like shared memory utilization and complex branch management, exist, the overhead of GPU-TDR escalates with branch quantity and is limited by the kernel block size. Separately, Liang et al. [11][4] presented a performance model, the Basic Block Vector (BBV), to alleviate the repercussions of control-flow divergence within GPU kernels, but the model's inference overhead grows with branch count. Collectively, current methodologies either fail to manage intricate control flows or impose considerable overhead effectively.

3 DESIGN AND IMPLEMENTATION

3.1 Overview

The architecture of the proposed system is shown in Figure 3 and is divided into two phases: offline and runtime. In the offline phase, first, we train a network to predict execution paths of thread data in user kernels, addressing issues like complex nested branches and the predictability of branching conditions. Then, this network is deployed on the tensor kernel. Finally, the fusion kernel is constructed by the user kernel and the network inference kernel in a suitable ratio. The methodology of the fusion kernel is detailed in Section 3.4. The runtime phase consists of three components: the input data regrouping algorithm implemented on the CPU, the fusion kernel scheduled on the GPU, and the pipelining parallelism of the workload between the CPU and GPU.

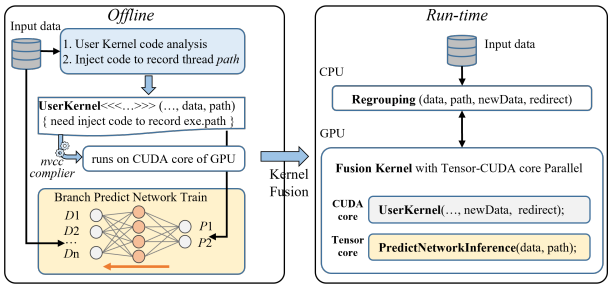


Figure 3: An overview of the proposed system structure.

3.2 Branch Prediction Network

Neural networks are known for their ability to categorize, regress and compress data[20]. These networks are well suited for predicting the execution paths of threads[5, 14], as the networks have a great advantage in learning the relationship between historical input data and the branching paths of thread execution [25]. We introduce a lightweight network to predict the branching paths executed by threads as they execute that input data. The prediction result provides thread execution paths for thread data reorganization algorithms.

Neural network structure. Our network design aims for minimal model size while ensuring prediction accuracy. A fully connected network is selected for compatibility with tensor core matrix operations. The network applies the formula $\mathbf{y} = \sigma(\sum_{l=1}^{L} \mathbf{X}^{l-1}\mathbf{W}^{l} + \mathbf{b}^{l})$ is utilized, where $\mathbf{X}^{l-1}\mathbf{W}^{l}$ represents input matrix multiplied by the weight matrix. The bias $\mathbf{b}^{l}$ and the network output $\mathbf{y}$ have $1 \times n$ dimensions, where n represents the number of CUDA thread branching paths. The activation function, $\sigma(\cdot)$, uses *LeakyReLU* and *softmax* for the middle and output layers, respectively. The number of hidden neurons should be a multiple of 8 to optimize network inference speed, aligning with tensor cores for efficient matrix operations and resource utilization.

Network input-output data encoding and training. Proper encoding of input-output data is essential to improve the accuracy of neural network classification. The choice of encoding methods depends on the type and dimensions of the thread-data input in CUDA user kernels. Standard data encoding techniques include binary, numerical, one-hot, and embedded encoding. Specifically, one-hot encoding is employed for network output data to determine the thread-data path. The encoding of network input data varies based on the user kernel's processed data format. Numeric input data is generally normalized and encoded continuously, whereas string input data in bioengineering applications typically employs embedded encoding. The training dataset includes user kernel input data and labels indicating branch paths executed by CUDA threads. Data collection involves running the kernel with diverse inputs, using cross-entropy as the loss function and *Adam* for optimization. The network, trained with extensive historical data, learns to predict conditional code branches, enhancing accuracy in branch path forecasting. Although predictions in complex branching scenarios show some variation, the general similarity in paths aids in mitigating control flow divergence impacts.

Algorithm 1 Data regrouping method

Input: *inData*, *g*, *path*[*g*];
Output: *outData*, *index*;

```
function ReGroup(inData, path, g)
    Initial the output parameters outData , index;
    Initial i and n : i ← 0, n ← len(inData) ;
    while i < n do
        end = min(i+g,n);
        p = max(path[i:end]);
        subData = inData[i:end];
        heapSortWithIndex(subData, i, end, index,p);
        i = i + g;
    end while
    return outData , index
end function
```

3.3 Data Regrouping

The data regrouping algorithm reorganizes input data based on predicted execution paths to minimize control flow divergence in GPU computing, as shown in Figure 2. The pseudocode for this Algorithm 1 works with an input data array '*inData*', a '*path*' array, and a grouping size '*g*'. It rearranges '*inData*' into subgroups of size '*g*', using a heap sort method that keeps track of the original positions of elements through an '*index*' array. This index helps maintain the original order of the data and ensures the right of user kernel execution result. The algorithm sorts each subgroup with this method and returns the reorganized data and their original indices.

For optimal efficiency, the size of the grouped arrays should be at least 64 (an integer multiple of warp size). This data grouping size is crucial as it aligns with the GPU's warp structure. Overly large groups may result in excessive disparities in data positioning, while too small groups could result in some branches having fewer elements than a warp, thereby failing to effectively eliminate warp divergence. The algorithm needs to keep the computational complexity low to avoid large time overhead. The execution time of the algorithm on the CPU should be less than the kernel execution time of the GPU. This is the case to hide the execution time of data regrouping in the CPU-GPU pipeline process.

Algorithm 2 Kernel Fusion with Persistent Threaded Blocks

Input: $fTCgrid, fTCblock, fCDgrid, fCDblock$;

1: $fTCgrid$ = min($TCgrid, CDgrid$);
2: $ptb = \lceil (TCgrid * TCblock)/(fTCblock * fTCblock) \rceil$;
3: **if** (threadIdx.x < $fCDblockDim.x$) **then**
4: tid = threadIdx.x - $fCDblockDim.x$ * 0;
5: Execute **CDkernel**(..., tid);
6: *bar.sync* inside a block;
7: **else if** (threadIdx.x < $fBlockDim.x$) **then**
8: tid = threadIdx.x - $fCDblockDim.x$ * 1;
9: **ptbTCkernel**(..., tid);
10: **end if**
11: **function** ptbTCkernel(..., ptb, tid)
12: **for** (int i=0; $i \leq ptb$; i++) **do**
13: blockPos = blockIdx.x + i * fTCblockDim.x;
14: idx = tid + blockPos * fTCblockDim.x;
15: TC code with new thread index idx to access data;
16: **end for**
17: **end function**

3.4 Tensor-CUDA core Parallel Schedule

Under the internal scheduling policies of GPUs, kernels utilizing Tensor Cores and CUDA Cores typically execute sequentially rather than concurrently. While TCs and CDs are hardware-independent, their warp scheduling is governed by a hardware mechanism beyond users' complete control or comprehension. As depicted in Figure 4(a), attempting to simultaneously execute a deep learning inference kernel (designed for Tensor Cores) alongside a graphics rendering kernel (employing CUDA Cores) may result in insufficient resource allocation for concurrent operation. This often causes one kernel to monopolize the majority of GPU resources, forcing the other to idle, as indicated by Zhao et al. [28]. Consequently, these kernels tend to operate in series rather than in parallel. Furthermore, the manner in which CUDA orchestrates streams [19] and blocks [18] can lead to conflicts in priority or an imbalanced distribution of resources. This complicates the parallel execution of kernels with different core types. Despite the inherent parallel processing capabilities of GPUs, kernels using distinct core types frequently face challenges in concurrent execution due to these scheduling constraints.

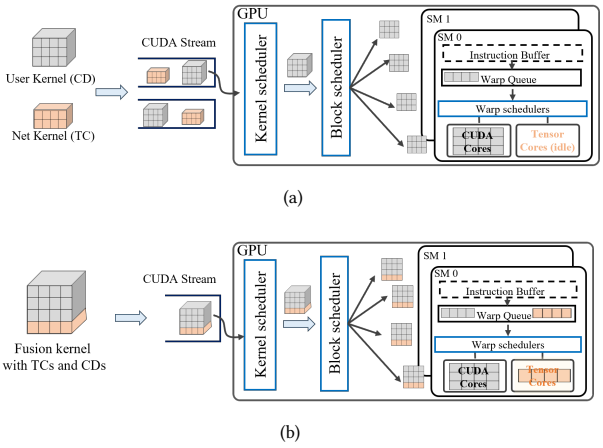


Figure 4: The scheduling process of the fusion kernel. (a) Parallelizing TCs and CDs presents significant challenges at the kernel-level scheduling. (b) An appropriate kernel fusion strategy can utilize TCs and CDs in parallel.

Kernel fusion integrates TCs and CDs within a single kernel, enhancing parallelism. This technique enables the cooperative functioning of both core types, facilitating more effective passage through CUDA streams and block schedulers, as illustrated in Figure 4(b). Warps originating from TCs and CDs are queued together for execution. Subsequently, the warp scheduler allocates each warp to the appropriate computational units. This strategy capitalizes on the strengths of both TCs and CDs, resulting in optimized resource utilization and augmented computing efficiency.

Kernel fusion focuses on optimizing GPU resource utilization and balancing the ratio of TCs and CDs to exploit their respective computational strengths. The fusion process adheres to specific constraints: minimal alteration of the user's kernel code, adjustment of the grid and block dimensions of the tensor core's network inference kernel, and compliance with GPU hardware limits. Our proposed approach, detailed in Algorithm 2, employs kernel fusion through Persistent Threaded Blocks (PTB). Initially, it computes the tensor core's dimensions, TCgrid and TCblock, in accordance with the anticipated network load. Subsequently, as indicated in line 1, it calculates a fusion Grid size, denoted by "f" to represent fusion-related dimensions, which takes into account the configuration of both the user kernel and the network. Finally, the algorithm determines the requisite number of PTBs for each block to maintain a consistent count of tensor core blocks.

During execution, threads are assigned tasks in the CDkernel or ptbTCkernel (line 11) based on their positions. This approach efficiently handles several tasks at once in one kernel, reducing context switches and memory latency. As specified in the line 6, synchronization within blocks maintains the user's kernel's accuracy. Specifically, the ptbTCkernel function carries out TC tasks for each PTB. By looping, each thread processes many data points, lessening the user kernel's block scheduling effects on the GPU. This algorithm's goal is to combine many operations in one kernel while enhancing efficiency and parallel processing.

4 EVALUATION

4.1 Experiments Setup

The effectiveness of the proposed method is empirically tested on two distinct embedded platforms with differing GPU architectures: Jetson AGX Xavier (Volta-based) and Orin (Ampere-based). This evaluation of diverse systems provides a comprehensive understanding of the method's effectiveness across GPU architectures. Importantly, this solution is not confined to these platforms and is extendable to other commercially available GPUs. The benchmarks, representing varied control flow divergence levels, are sourced diversely: LBM (Lattice-Boltzmann Method)[23] is a fluid dynamics partial differential equation solver, exemplifying moderate control flow divergence. CUDA-EC (denoted by EC for short) [22] is a bioinformatics application for error correction in high-throughput sequencing reads, encompassing a spectrum of control flow divergence. GDT (Genetic-Programmed Decision Tree)[21] is a parallel genetic algorithm for CUDA architecture, highlighting mixed non-coalesced memory access patterns and adding complexity to control flow.

Benchmark-specific prediction network models vary in structure, reflecting each CUDA thread's data size and program code branches. For CUDA-EC, the network model employs a fully connected 10×32×28 architecture with 32 neurons in the hidden layer, aligned with tensor core computation by ensuring a neuron count as a multiple of 16. Similarly, the GDT network model is fully connected with a 16×32×10 structure.

While the models are not subject to extensive tuning, they achieved over 90% accuracy in branch prediction. The execution time of these models primarily hinges on the design of a parallel scheme that integrates seamlessly with the CUDA core user kernel. Our results affirm the proposed approach's capability to effectively manage complex and nested control flow divergence scenarios.

This work uses CUDA 10.0 (for Xavier) and CUDA 11.4 (for Orin) for programming. Each benchmark is compiled with the flags "-arch sm_72" for Jetson AGX Xavier and "-arch sm_87" for Orin, enabling Tensor core API support. We compare the proposed method with two established techniques, HoT [13] and DGI [12]. HoT (Head-or-Tail) employs atomic operations on shared variable pairs in the GPU, and DGI (Data Group Indexing) is tailored for multi-path branch programs. Both methods require kernel integration in GPUs for efficient data remapping.

4.2 Performance

To evaluate performance gains, we use the kernel runtime speedup to benchmark various methods against a baseline, as shown in Figure 5. These benchmarks include evaluations on the Xavier ("-X") and Orin ("-O") platforms. Our approach, which mainly employs DLT and occasionally RR, outperforms others in most benchmarks, except HoT on Orin. The LBM used in HoT, heavily reliant on global memory access, limits the effectiveness of control flow optimization, leading to insufficient performance gains. Notably, our method achieved significant speedups of 1.7x and 1.8x in the GTD-X and EC-O benchmarks, respectively.

To assess our method's efficiency, we compare CPU and GPU execution time for branch prediction. On the Jetson Xavier platform, GPU-based prediction significantly outperforms CPU-based prediction in terms of execution time. For example, CPU-based prediction time for GDT-X, EC-X, GDT-O, and EC-O benchmarks are 75.82us+98ms, 812.07ms+3.64s, 20.76us+35ms, and 313.43ms+1.51s, respectively, with the second term indicating branch prediction

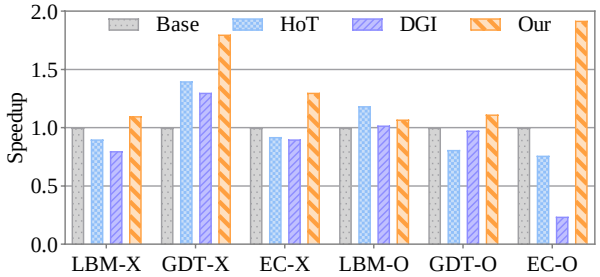


Figure 5: Average execution time speedup.

network time. In contrast, GPU execution yields a considerably shorter time: 81.63us, 828.05ms, 21.44ms, and 321.33ms.

Moreover, the branch prediction networks' processing time on the CPU greatly exceeded the benchmarks' execution time. This led to prolonged total execution time for CPU-based prediction, even considering the potential pipelining of the prediction network and benchmark execution. Such inefficiency is associated with the computational load of the branch prediction network, which is more suitably executed on tensor cores.

We evaluated the effectiveness of our method using Control Flow Efficiency (CFE), a metric measuring active threads per warp for each executed instruction: $CFE = \frac{\text{Thread Instructions Executed}}{\text{Instructions Executed} \times \text{Warp Size}}$, where 'Thread Instructions Executed' refers to the count of instructions executed by threads not affected by predicate off. We used the profiling command 'not_predicated_off_thread_inst_executed' (or 'smsp__thread_inst_executed_pred_on.sum' in CUDA 11) to measure relevant events.

A higher CFE indicates less warp divergence, reflecting better performance. Our evaluation compares four methods: 'Base', 'HoT', 'DGI', and 'Our'. 'Base' is the original benchmark's CFE, 'HoT' and 'DGI' are specific methods, and 'Our' method generally shows higher CFE, especially in benchmarks with complex branches like GDT and EC. This higher CFE across most benchmarks signifies reduced warp divergence and improved performance. The CFE metric is essential in assessing warp divergence and its performance impact. Our method demonstrates effective control flow divergence mitigation on various embedded GPU platforms.

Table 1: CFE of benchmarks under different methods.

Benchmark	Base	HoT	DGI	Our
LBM-X	78.9%	80.3%(1.02x)	63.5%(0.80x)	75.8%(0.96x)
GDT-X	63.0%	80.9%(1.28x)	68.2%(1.08x)	90.6%(1.44x)
EC-X	12.9%	30.8%(2.39x)	30.9%(2.40x)	46.9%(3.64x)
LBM-O	83.5%	83.1%(0.99x)	62.0%(0.74x)	89.6%(1.07x)
GDT-O	63.6%	72.4%(1.14x)	63.3%(1.00x)	95.8% (1.51x)
EC-O	12.7%	30.5%(2.41x)	30.5%(2.41x)	46.6%(3.68x)

4.3 Execution Time Divergence

We evaluated overall performance and analyzed execution time variability, or jitter, of each algorithm under different inputs to assess their timing predictability and stability. The execution time distributions, illustrated by box plots in Figure 6, show the standard deviations of normalized run time. Lower standard deviations indicate more predictable and consistent execution time.

Figure 6 reveals that our method consistently achieves a low standard deviation (at or below 4) across various inputs, suggesting reliable and predictable timing performance. This is crucial for real-world applications where stable execution time are imperative. In the case of GDT, a slightly higher standard deviation is observed compared to DGI, likely due to our algorithm's higher speedup ratio. Despite this, our method minimizes control flow divergence

effectively, which may result in minor variations in execution time due to increased computational efficiency.

Overall, the analysis of jitter and timing predictability confirms the effectiveness and reliability of our data regrouping algorithm in practical applications, offering stable performance across diverse inputs and platforms. This consistency is particularly valuable in scenarios requiring precise timing predictability to meet strict performance standards.

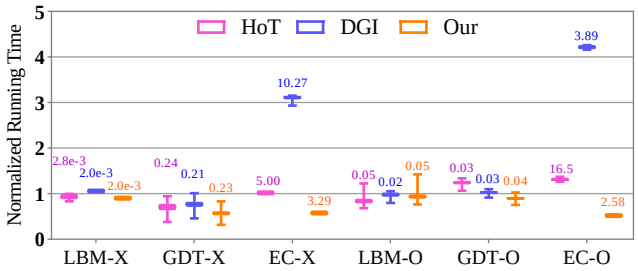


Figure 6: Box plot of normalized execution time and standard deviation.

4.4 Parallelism Analysis

In the following section, we evaluate the branch prediction overhead on TCs and its relationship with the parallelism between CDs and TCs. This overhead largely depends on the level of parallelism between CDs and TCs. We observed that the branch prediction overhead on TCs is effectively minimized when CD tasks are executed concurrently with TC tasks during multiple user kernel executions. To measure this parallelism, we use the 'Overlap Rate' metric [27], calculated as: $OverlapRate = \frac{T_{cd}+T_{tc}-T_{fuse}}{T_{cd}+T_{tc}}$, where T_{cd}, T_{tc}, and T_{fuse} represent the individual run time of the CD and TC tasks and the total concurrent duration, respectively. The Overlap Rate effectively quantifies the concurrency between CD and TC kernels.

Overlap rates range from 0 to 50%, with higher rates indicating stronger parallelism. Table 2 shows kernel details and overlap rates for fused kernels ("-F"), including GridDim, BlockDim, Register usage, and overlap rates on both Volta-based (OR-X) and Ampere-based (OR-O) platforms, to illustrated these rates.

Table 2: Kernel information and overlap rate.

Benchmark	GridDim	BlockDim	Register	OR-X	OR-O
GDT	(9,1,1)	(128,1,1)	34	-	-
GDT-F	(9,1,1)	(128,5,1)	32	49.1%	45.2%
EC	(256,1,1)	(256,1,1)	66	-	-
EC-F	(256,1,1)	(128,5,1)	89	26.9%	15.3%

Data analysis indicates that both GDT-F and EC-F benchmarks have high overlap rates, reflecting substantial parallelism between TC and CD kernels. GDT-F shows a higher overlap rate than EC-F, attributed to its TC kernel's shorter execution time. In contrast, EC-F's lower overlap rate results from the extended execution time of its CD kernel.

In summary, these findings suggest that the runtime overhead of the TC kernel is remarkably low, particularly in benchmarks with high overlap rates. Such efficient, hidden branch prediction overhead through parallelism between CDs and TCs demonstrates our method's effectiveness in handling complex control flow challenges in embedded GPUs.

5 CONCLUSION

This research introduces a Tensor core-based branch prediction network for CUDA kernels, enhancing thread path predicting and reducing warp divergence. The parallel between Tensor and CUDA cores, complemented by a data regrouping algorithm, effectively mitigates control flow divergence, notably boosting system performance.

6 ACKNOWLEDGMENTS

This research is supported by the National Science and Technology Major Project (2022ZD0116800).

REFERENCES

[1] Sara S Baghsorkhi and Matthieu Delahaye. 2010. An adaptive performance modeling tool for GPU architectures. In *ACM SIGPLAN*. 105–114.
[2] Shuai Che and Michael Boyer. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE IISWC*. Ieee, 44–54.
[3] Jack Choquette and Olivier Giroux. 2018. Volta: Performance and programmability. *Ieee Micro* 38, 2 (2018), 42–52.
[4] Zheng Cui. 2012. An accurate GPU performance model for effective control flow divergence optimization. In *2012 IEEE PDPS*. IEEE, 83–94.
[5] Colin Egan. 2003. Two-level branch prediction using neural networks. *JSA* 49, 12-15 (2003), 557–570.
[6] Rob Farber. 2011. *CUDA application design and development*. Elsevier.
[7] Wilson WL Fung. 2007. Dynamic warp formation and scheduling for efficient GPU control flow. In *IEEE/ACM MICRO 2007*. IEEE, 407–420.
[8] Wilson WL Fung. 2009. Dynamic warp formation: Efficient MIMD control flow on SIMD graphics hardware. *ACM TACO* 6, 2 (2009), 1–37.
[9] Wilson WL Fung. 2011. Thread block compaction for efficient SIMT control flow. In *2011 IEEE HPCA*. IEEE, 25–36.
[10] Mahmoud Khairy. 2020. Accel-Sim: An extensible simulation framework for validated GPU modeling. In *ACM/IEEE ISCA*. IEEE.
[11] Yun Liang. 2015. An accurate GPU performance model for effective control flow divergence optimization. *IEEE TCAD* 35, 7 (2015), 1165–1178.
[12] Huanxin Lin. [n. d.]. On-GPU thread-data remapping for nested branch divergence. *J. Parallel and Distrib. Comput.* ([n. d.]).
[13] Huanxin Lin. 2018. On-GPU thread-data remapping for branch divergence reduction. *ACM TACO* 15, 3 (2018), 1–24.
[14] Yonghua Mao. 2020. Exploring convolution neural network for branch prediction. *IEEE Access* 8 (2020), 152008–152016.
[15] Jiayuan Meng. 2010. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *Proceedings of the 37th annual international symposium on Computer architecture*. 235–246.
[16] Asit Mishra. 2021. Accelerating sparse deep neural networks. *arXiv preprint arXiv:2104.08378* (2021).
[17] NVIDIA. 2022. Warp Matrix Multiply-Accumulate (WMMA). http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
[18] Ignacio Sañudo Olmedo. 2020. Dissecting the CUDA scheduling hierarchy: a performance and predictability perspective. In *2020 RTAS*. IEEE, 213–225.
[19] Weiguang Pang. 2023. Efficient CUDA stream management for multi-DNN real-time inference on embedded GPUs. *JSA* 139 (2023), 102888.
[20] Weiguang Pang and Xu Jiang. 2021. Towards the Predictability of Dynamic Real-Time DNN Inference. *IEEE TCAD* 41, 9 (2021), 2849–2862.
[21] Petr Pospichal. 2010. Parallel genetic algorithm on the cuda architecture. In *European conference on the applications of evolutionary computation*. Springer, 442–451.
[22] Haixiang Shi. 2010. A parallel algorithm for error correction in high-throughput short-read data on CUDA-enabled graphics hardware. *Journal of Computational Biology* 17, 4 (2010), 603–615.
[23] John A Stratton. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012), 27.
[24] Wei Sun. 2022. Dissecting Tensor Cores via Microbenchmarks: Latency, Throughput and Numeric Behaviors. *IEEE Transactions on Parallel and Distributed Systems* 34, 1 (2022), 246–261.
[25] David Tarjan. [n. d.]. Merging path and gshare indexing in perceptron branch prediction. *ACM TACO* ([n. d.]).
[26] Eddy Z Zhang. 2010. Streamlining GPU applications on the fly: thread divergence elimination through runtime thread-data remapping. In *Proceedings of the 24th ACM International Conference on Supercomputing*. 115–126.
[27] Han Zhao. 2021. Exploiting intra-sm parallelism in gpus via persistent and elastic blocks. In *2021 ICCD*. IEEE, 290–298.
[28] Han Zhao. 2022. Tacker: Tensor-CUDA Core Kernel Fusion for Improving the GPU Utilization while Ensuring QoS. In *2022 IEEE HPCA*. IEEE, 800–813.