# GoPTX: Fine-grained GPU Kernel Fusion by PTX-level Instruction Flow Weaving

Kan Wu, Zejia Lin, Mengyue Xi, Zhongchun Zheng, Wenxuan Pan, Xianwei Zhang[#], Yutong Lu[#]
*Sun Yat-sen University, Guangzhou, China*
{wukan3, linzj39, ximy, zhengzhch3, panwx5}@mail2.sysu.edu.cn, {zhangxw79, luyutong}@mail.sysu.edu.cn

***Abstract*—GPUs have been heavily utilized in diverse applications, and numerous approaches, including kernel fusion, have been proposed to boost GPU efficiency through concurrent kernel execution. However, these approaches generally overlook the opportunities to mitigate warp stalls and improve instruction level parallelism (ILP) in inter-kernel resource sharing. To address this issue, we introduce `GoPTX`, a novel design for kernel fusion that improves ILP through deliberate weaving instructions at the PTX level. `GoPTX` establishes a merged control flow graph (CFG) from original kernels, enabling to interleaving of instructions that were sequentially executed by default and minimizing pipeline stalls on data hazards. We further propose a latency-aware instruction weaving algorithm for more efficient instruction scheduling and an adaptive code slicing method to enlarge the scheduling space. Experimental evaluation demonstrates that `GoPTX` achieves an average speedup of 11.2% over the baseline concurrent execution, with a maximum improvement of 23%. The hardware resource utilization statistics show significant enhancements in eligible warps per cycle and resource use.**

***Index Terms*—GPU, Kernel Fusion, ILP, Warp Stall, Data Hazard**

## I. Introduction

As GPUs incorporate an increasing amount of computing resources, it becomes difficult for a single GPU kernel to fully utilize the vast resources. One solution is to share resources through concurrent kernel execution, where GPU vendors provide hardware parallel task queues and management, such as cuStream, MPS [1], and MIG [2] to execute multiple tasks. Essentially, these solutions rely on the GPU's thread-block (TB) scheduler to allocate resources among the co-running kernels. Nonetheless, the leftover policy [3], [4] employed by the TB scheduler results in imbalanced resource allocation, greatly limiting the effectiveness of intra-SM resource sharing.

To overcome the hardware limitations, software-based approaches have been proposed for improving inter-kernel resourcing utilization. One such representative method is kernel fusion, which has been adopted by mainstream deep learning frameworks [5], [6], tools [7], [8] and systems [9], [10]. The fused kernel enforces the co-execution of instructions from two kernels onto the same SM which issues warps that utilize complementary resources, thereby improving the overall SM utilization. However, such warp-level concurrency incurs frequent thread context switching and stalls because of resource contention [3], which can negatively impact the instruction pipeline and further ILP. To mitigate such overhead, carefully instruction scheduling is needed and thus a fine-grained technique to weave instruction flows from concurrent kernels into a unified flow is urgently demanded. By weaving independent instructions, the data dependency length can be increased, allowing instructions from another kernel to proceed without stalling or warp switching, filling pipeline bubbles, hiding the execution latency, as shown in Figure 1.

However, several unique challenges are associated with instruction weaving. *First*, kernels may have complex control flow graphs (CFGs) that include synchronization barriers for groups of threads. Weaving the instructions of these kernels requires a careful operation to merge the diverging CFGs into a unified structure while ensuring correctness and avoiding potential deadlocks. *Second*, achieving perfect instruction reordering is an NP-hard problem [11], [12], especially for GPUs with thousands of parallel threads. Identifying a near-optimal weaving output within a limited time is crucial for a feasible solution. To the best of our knowledge, no existing software-based solutions automatically weave instructions from distinct kernels to share resources and optimize ILP. Previous attempts [13], [14] have been limited to fusion at the source code level, focusing primarily on exploiting TLP.

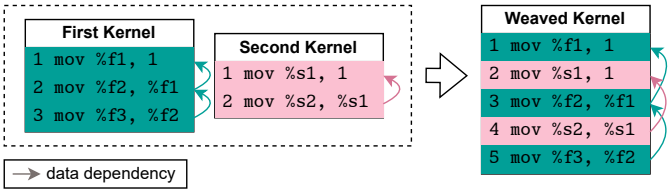



Fig. 1: Increase dependency length by instruction weaving.

In this paper, we present `GoPTX`, a novel design for kernel fusion that operates at the PTX level. `GoPTX` introduces a new CFG merging algorithm to unify CFGs from multiple kernels, which involves critical techniques to preserve the execution semantics and resolve deadlocks caused by synchronization. Moreover, we develop a latency model to guide the interleaving of independent instructions within the basic blocks of newly generated CFGs, which effectively helps fill pipeline bubbles to optimize execution performance. Additionally, the latency model is used to estimate the elapsed cycles of basic blocks (BBs), enabling the adaptive slicing of long codes into balanced segments before CFG merging, creating more opportunities for instruction weaving.

The contributions of this paper are summarized as follows:

- Our analysis reveals that GPU warp stalls primarily originate from the scoreboard, hinders the achievable ILP.
- We propose `GoPTX`, a design for kernel fusion that weaves instructions from distinct kernels and fills the pipeline bubbles. `GoPTX` is further augmented with code slicing to expose more ILP opportunities. `GoPTX` particular handles to avoid deadlocks.
- Evaluations show that `GoPTX` effectively improves execution performance with optimized hardware utilization and reduced stalls, outperforming the prior arts of kernel fusion.

## II. Background and Motivation

### A. Background

*a) GPU Hardware Architecture:* A GPU[1] consists of multiple streaming multiprocessors (SMs), each containing hundreds of processing cores and on-chip components like register files and

[#]Corresponding author.

[1]We use the terms from NVIDIA GPU design in this paper.

L1 caches. Logical threads execute on these cores in a single-instruction multiple-thread (SIMT) fashion. A group of 32 threads (a warp) operates in lock-step within the execution pipeline. If a warp stalls, the scheduler switches to another eligible warp, continuously issuing instructions (Figure 2). This thread-level parallelism (TLP) hides instruction pipeline latency, making the GPU a high-throughput computational device.

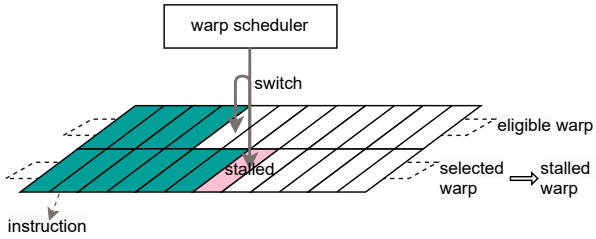



Fig. 2: The GPU warp scheduler frequently encounters stalls, causing a switch to another thread and resulting in "bubbles" (pink colored) on the execution path.

*b) CUDA Compilation Workflow:* Programmers define parallel portions of their applications as kernels within the vendor's dialect (e.g., CUDA C). As shown in Figure 3, the CUDA frontend (e.g., `nvcc` or `clang` [15]) parses the source code and generates an intermediate representation (IR) named PTX. Subsequently, the PTX code is compiled by the backend compiler `ptxas` into an executable binary file (*cubin*) that is tailored for the target GPU architecture. PTX is defined by NVIDIA and resembles LLVM IR [16]. A kernel in PTX consists of basic blocks (BBs) forming a control flow graph (CFG). Each BB contains a bunch of diverse instructions and ends with a terminator instruction (e.g., a branch or function return). All the instructions are in single-assignment form with limitless registers available and may include hardware-specific primitives like barriers and tensor core operations. PTX offers a direct abstraction of the underlying GPU hardware, enabling effective backend compiler optimizations.

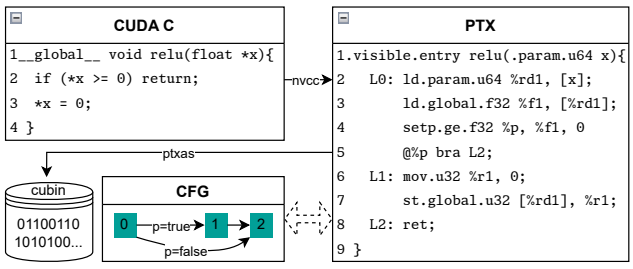



Fig. 3: CUDA compilation workflow.

*c) Kernel Fusion:* Targeting improved utilization and performance, kernel fusion combines two concurrent kernels at the software level to share on-chip resources. Unlike hardware methods, kernel fusion incurs no hardware overhead or limitations, making it flexible and efficient. Figure 4 shows two primary existing fusion techniques and our desire. Vertical fusion (`VFuse`) [13] sequentially combines the instructions from input kernels, horizontal fusion (`HFuse`) [14] distributes the instructions to different warps, while our desired fine-grained weaves two kernels together.

### B. Opportunity to Mitigate Warp Stalls

We analyze utilization statistics from representative application kernels (Section IV-B) to explore the potential of ILP. Figure 5a compares the average number of eligible warps per cycle (EWPC) of the kernels. We observed that the EWPC of other kernels is significantly lower compared to the computationally intensive MICND kernel. Figure 5b shows the stall reasons. The most prevalent stall cause for kernels other than MICND is the scoreboard, which handles data dependencies and enables out-of-order instruction execution. This suggests that data hazards and resource contention cause pipeline stalls and limit the achievable EWPC and ILP.

To alleviate the scoreboard stalls , fine-grained instruction scheduling is required. However, current intra-kernel instruction reordering techniques reach a performance ceiling because of data dependents. Therefore, we turn to "weave" instructions between concurrent kernels for further scheduling space. Figure 1 shows an example, where each instruction depends on the previous one so all the data dependency length is 1, and can not be reordered or pipelined to improve ILP. After weaving, all the dependency length increases to 2, improving the pipeline's ability to process dependent instructions.

## III. Design

In this section, we present `GoPTX`, a novel design for kernel fusion that improves ILP by weaving instructions at PTX level, enabling efficient resource sharing between kernels while mitigating warp stalls. Figure 6 depicts the overall workflow of `GoPTX`, generating a weaved PTX code from two input PTX codes. The workflow comprises three phases:

*a) CFG Merging:* A new control flow graph (CFG) is constructed, unifying the execution paths of both kernels. Branching conditions and dummy nodes are inserted to maintain equivalents and avoid deadlocks.

*b) Instruction Weaving:* A low-complexity weaving strategy constructs a locally optimal instruction sequence, minimizing pipeline stalls. Instructions are scheduled based on latencies measured via micro-benchmarking.

*c) Code Slicing (Optional):* Long BBs are split into multiple short segments before CFG merging, creating opportunities for cross-BB instruction weaving.

### A. CFG Merging

Algorithm 1 describes CFG merging (CFM) as a traversal procedure working on BB-level. CFM merges the input kernels, denoted as $F$ and $S$ into a merged CFG $M$, ensuring that the generated CFG preserves the execution semantics of both original kernels. Initially, the output CFG $M$ is set to empty. The traversal begins at the initial state $\langle 0, 0\rangle$ of $M$, corresponding to the starting BBs $F_0$ and $S_0$ of both CFGs (lines 3–6). For each traversed state $\langle f, s\rangle$ (line 8), the followings are performed. Firstly, CFM fuses BBs $F_f$ and $S_s$ into a single BB $M[\langle f, s\rangle]$, providing the foundation for *instruction weaving*. Then, the successors of the merged BB are retrieved using the function $get\_next\_list$ (line 10). The traversal continues until all possible state $M[\langle f, s\rangle]$ are consumed (lines 11–15).

The function $get\_next\_list$ is tailored to handle branching in CFG merging. Figure 7 illustrates the outputs of this function, depending on the characteristics of the origin basic blocks (BB) $F[f], S[s]$. For trivial BBs without branches (Figure 7a), the function returns the succeeding BB if $M[\langle f, s\rangle]$ is not an end node, otherwise an empty node. When the BB has conditional branches (Figure 7b), the function exhaustively combines each conditional branch from the original kernels. To handle the branching logic correctly, two dummy nodes are inserted, allowing the complete conditions to be determined and facilitating correct execution across the merged paths.

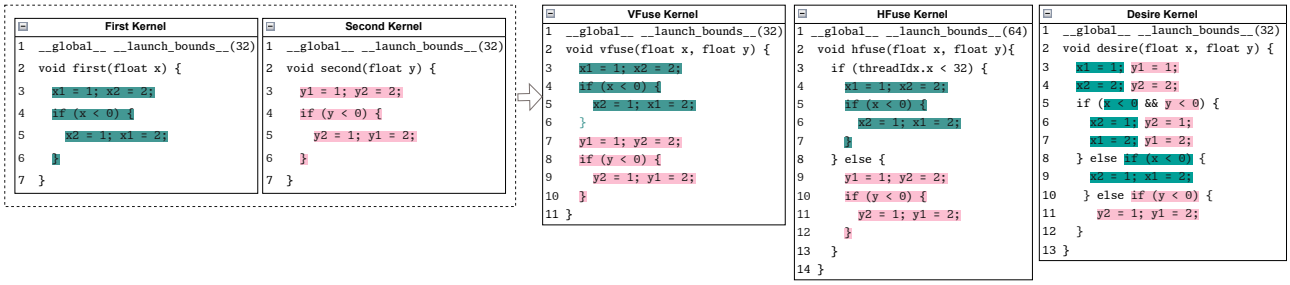
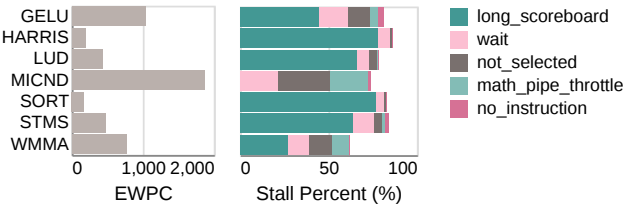



Fig. 4: Illustrative examples of `first` and `second` input kernel with coarse-grained (`VFuse`, `HFuse`) and fine-grained (`Desired`) fusion.




(a) Significant disparity of EWPC.

(b) Stall reasons breakdown, of which scoreboard contributes the most.

Fig. 5: EWPC and stall reasons distribution.

**Algorithm 1** Control Flow Graph Merging

**input**: $F \neq \emptyset, S \neq \emptyset$ {the first and the second input CFG}
**output**: $M$ {the merged CFG}
$M \leftarrow \emptyset$
$M.add\_node(< 0, 0 >)$
$Stack \leftarrow \{< 0, 0 >\}$
$Pushed \leftarrow \{< 0, 0 >\}$
**while** $Stack \neq \emptyset$ **do**
  $< f, s > \leftarrow Stack.pop()$
  $M[< f, s >].block \leftarrow F[f].block + S[s].block$
  **for** $< u, v > \in get\_next\_list(< f, s >, F, S)$ **do**
    **if** $< u, v > \notin Pushed$ **then**
      $M.add\_node(< u, v >)$
      $Stack.push(< u, v >)$
      $Pushed.insert(< u, v >)$
    **end if**
    $M.add\_edge(< f, s >, < u, v >)$
  **end for**
**end while**
**return** $M$

### B. Latency-Aware Instruction Weaving

Having fused the BBs from two kernels in Section III-A, we now focus on scheduling instructions from both input BBs within a limited time frame. However, finding the best instruction scheduling for a given basic block is a well-known NP-hard problem [11], [12] when the instruction dependencies form a directed acyclic graph (DAG) instead of a tree. To generate an optimized heuristic input for the `ptxas` backend compiler, we simplify the problem and propose a latency-aware instruction weaving method. We propose a latency-aware weaving algorithm based on the metrics measured in Paragraph III-D0b, which can be viewed as a greedy algorithm that takes instructions lists of the fusing BBs as input. As shown in Figure 8, in each iteration, `GoPTX` calculates the sum of instruction latencies for instructions already woven from each list, and selects a new instruction from the list with the lower sum. If the sums are equal, the instruction with the higher individual latency is chosen. The process continues until all instructions are woven.

### C. Adaptive Code Slicing

The basic CFM approach can overlook weaving opportunities when block sizes differ significantly as the smaller block may not provide sufficient instructions to hide the latency of the longer instruction flow. To address this, we introduce code slicing, dynamically adjusting block sizes to balance latencies and maximize weaving potential. Slicing involves dividing blocks into smaller segments, ensuring each segment's execution time remains below a threshold, which is determined by the average number of execution cycles of all basic blocks, as shown in Equation 1. This strategy balances block sizes, maximizes weaving opportunities, and improves latency hiding.

$$threshold = \left\lceil \frac{\sum_{i=1}^{N_{blocks}} \sum_{j=1}^{M_{block_i}} latency(block_i, j)}{N_{blocks}} \right\rceil \quad (1)$$

### D. Implementation

*a) PTX Code Transformation:* Due to the lack of official PTX processing tools from NVIDIA, we developed a custom PTX parser based on ANTLR4 [17]. This parser enables in-depth control flow and identifier analysis and allows us to rename identifiers from two kernels before CFG merging to prevent namespace conflicts. Compared to source code level work [13], [14], PTX-level transformation provides a closer abstraction to the underlying GPU hardware while maintaining a simple intermediate representation, benefiting instruction-level manipulation in `GoPTX`. Our design also avoids modifying existing CUDA C source code and executables, ensuring easy adaptability to existing code for current and future devices.

*b) Instruction Latency Model:* We measure instruction latency via microbenchmark [18] which records the average execution cycles of a single thread repeatedly executing the target PTX instruction. For shared memory and global memory accesses, we employ a pointer-chasing technique to ensure accurate latency measurements. This technique serializes memory accesses, preventing simultaneous multiple memory accesses that could lead to inaccurate measurements. The resulting latency profile enables precise performance predictions and optimizations specific to the GPU architecture, providing a versatile solution for instruction weaving.

*c) Race Condition Avoiding:* GPU thread synchronization instructions, such as barriers and tensorcore operations, can lead to deadlocks when weaving instructions from two kernels. As shown in Figure 9a, both the input CFGs contain blocks with synchronization instructions. However, one of the CFGs exhibits different threads performing synchronization operations in different branches, disrupting the synchronization order in the merged CFG, and leading to potential deadlocks. To address this issue (Figure 9b), during

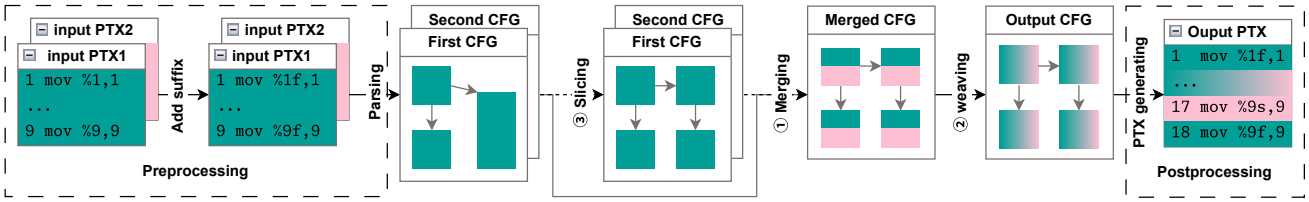
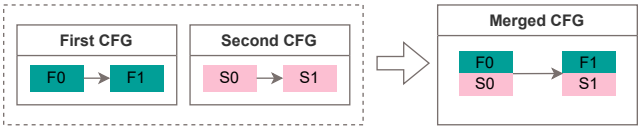
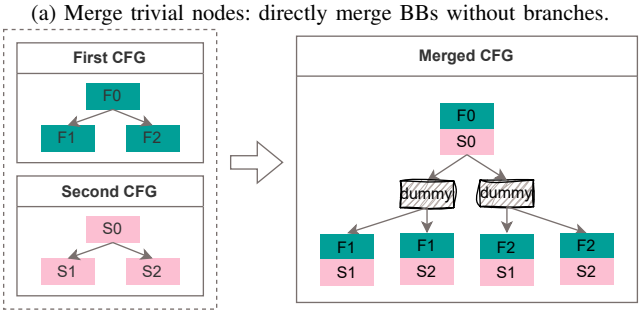



Fig. 6: Overall workflow of GoPTX, with three key phases of *Slicing*, *Weaving* and *Merging*, and additional phases of *Preprocessing* and *Postprocessing*.




(a) Merge trivial nodes: directly merge BBs without branches.




(b) Merge conditional branches: insert dummy nodes and branching for divergent control flow.

Fig. 7: Merging the control flow from two kernels.

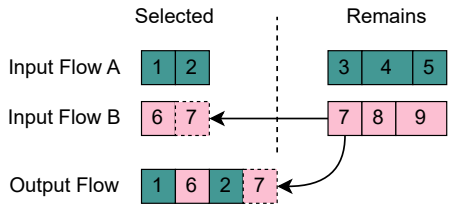



Fig. 8: The weaving process. Instruction 7 is selected because instruction 6's latency is lower than the sum of instruction 1 and instruction 2.

CFG merging, when encountering a block with synchronization instructions from the second kernel, we insert dummy blocks before it until the preceding block and its successors from the first kernel have no synchronization instructions. This ensures that threads do not stall indefinitely and preserves the original control flow, resulting in deadlock-free instruction weaving.

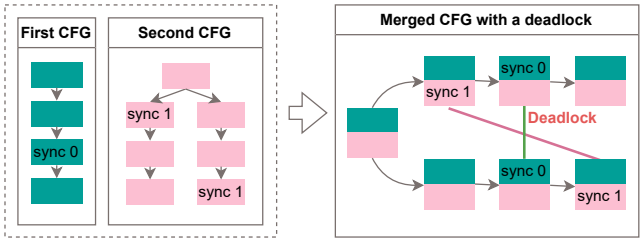
*d) Register Tuning:* CFG merging can increase register pressure, impacting occupancy and performance. Based on profiling, we measure the optimal `-maxrregcount` parameter in (32, 40, 48, 64, 80, 128, 256) to balance ILP and TLP.

*e) Multiple Kernel Fusing:* N-kernel fusion can be decomposed into sub-problems, e.g., (A+B)+C for 3-kernel fusion. However, previous studies [14] showed that merging more kernels diminishes returns due to the long tail effect.

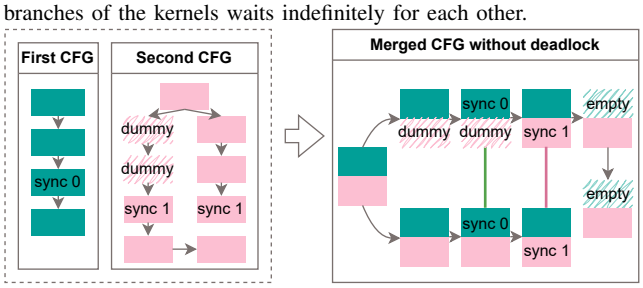
## IV. Evaluation Methodology

### A. Hardware and Software Platform

We conduct experiments on a server featuring AMD EPYC 7742 CPU, 256GB DRAM, and NVIDIA A100-PCIE-40GB GPU. We lock




(a) Deadlock occurs when the synchronization barrier in different branches of the kernels waits indefinitely for each other.




(b) Deadlock solved by inserting dummy blocks before synchronizations, delaying the barriers to prevent conflicts.

Fig. 9: Addressing race condition in the weaved kernel.

the GPU on max frequency 1410 MHz. The operating system is Ubuntu 24.04LTS, and CUDA driver version is 555.42.06. We use `nvcc` and `ptxas` shipped with CUDA 12.5.1, and `clang` 17.0.6 as the host compiler with compile option `-O3 -arch=sm_80`.

### B. Workloads

We evaluate GoPTX using multiple representative kernels drawn from realistic applications like the Rodinia Benchmark Suite [19], ONNX-runtime [20], CUDA Samples [21], and HIPACC Samples [22]. Table I characterizes the kernels in terms of register count (Reg), arithmetic operations (INT, FP), synchronization barrier (Sync), shared memory (Smem) and Tensor Core use (TC). For evaluation, we pair each kernel with all others, including itself, to create concurrent execution cases per kernel. This showcases GoPTX's ability to transcend source code limitations and integrate seamlessly into existing scheduling systems [23], [24]. To maintain consistent experimental conditions, we adjust the kernels to employ a standardized launch configuration without compromising correctness or performance, as supported by previous studies [13], [14]. In detail, we set the *blockDim* to (256,1,1) and the *gridDim* to (1048576,1,1). For `HFuse`, the *blockDim* is specifically set to (512,1,1) since `HFuse` directly combines two kernels at the block level. This launch configuration fulfills A100's stream multiprocessors to isolate the impact of instruction weaving and other factors on overall performance.

TABLE I: Kernels and their resource requirements.

| kernel | Time | Reg | INT | FP | Sync | Smem | TC |
|---|---|---|---|---|---|---|---|
| GELU [20] | 1.96ms | 19 | | ✓ | | | |
| HARRIS [22] | 1.49ms | 14 | ✓ | | | | |
| LUD [19] | 3.68ms | 30 | | ✓ | ✓ | ✓ | |
| MICND [21] | 1.25ms | 15 | | ✓ | | | |
| SORT [19] | 3.19ms | 17 | | | ✓ | ✓ | |
| STMS [22] | 1.63ms | 12 | ✓ | ✓ | | | |
| WMMA [21] | 2.81ms | 44 | | ✓ | ✓ | | ✓ |

### C. Metrics

To evaluate the effectiveness of our approach, we collect statistics with the CUDA Profiling Tools Interface (CUPTI) [25], with metrics being listed in Table II. *EWPC* is the indicator we focus on, representing the average number of warps ready per clock cycle. The higher the value, the more instructions can be executed at the same time, and the higher the ILP. *AOC* is the number of active warps per cycle divided by the GPU's maximum supported, representing TLP. Other indicators measure the GPU hardware utilization.

TABLE II: Hardware metrics to reflect resource usages.

| Metric | Note | Type |
|---|---|---|
| EWPC | eligible warps per cycle | ILP indicator |
| AOC | achieved occupancy | TLP indicator |
| DU | dram utilization | off-chip resource |
| L2U | l2 cache utilization | |
| L2HR | l2 cache hit rate | |
| ISU | issue slot utilization | on-chip resource |
| LSU | load-store unit utilization | |
| SPU | single precision unit utilization | |
| TSU | tex/l1 unified cache utilization | |
| THR | tex/l1 cache hit rate | |

### D. Comparing Schemes

We compare GoPTX with the following schemes shown in Figure 4 in our experiments: ❶ Baseline, launching two kernels in separate cuStreams to enable concurrent execution; ❷ VFuse [13], concatenating two kernels sequentially for fusion; ❸ HFuse [14], scheduling two kernels into different warps; ❹ GoPTX, our proposed design, merging control flow and weaving instructions.

## V. Results and Analysis

### A. Performance Improvement

Figure 10 shows the overall speedup achieved by GoPTX, which delivers an average of 11.2% speedup over the baseline, ranging from 23% (*STMS+LUD*) to -2% (*WMMA+GELU*). In contrast, VFuse and HFuse achieve moderate geometric mean speedups of 6.4% and 1%, respectively. While VFuse can achieve significant performance gains in certain compute-intensive benchmarks (*LUD*, *MICND*, *WMMA*), its benefits are less pronounced in others. HFuse, on the other hand, does not always lead to performance improvements as it requires extensive profiling to determine the optimal kernel combine ratio [14]. GoPTX significantly outperforms both HFuse and VFuse in the vast majority of benchmarks, except for *WMMA*. The *WMMA* kernel employs a large number of warp synchronization instructions to utilize tensor cores. To avoid deadlocks, GoPTX adopts a conservative merge strategy, which introduces overhead and leads to a slight performance decrease (0.05%) compared to VFuse.

Figure 11 showcases the impact of our approach on instruction-level parallelism (ILP), as measured by the number of eligible warps per cycle (EWPC). The data reveals that HFuse reduces EWPC by 14.4% in exchange for TLP improvement, while VFuse exhibits minimal changes in EWPC due to the sequential concatenation of kernels. In contrast, our approach demonstrably increases EWPC by 5.5% and max 50% (*STMS*, *LUD*) and min -35% (*WMMA*, *GELU*). The *SORT* benchmark presents unique cases. *SORT* is memory-bound, with a high volume of read operations and insufficient computation to effectively hide latency with other kernels. Consequently, despite performance gains achieved through inter-kernel resource sharing, ILP remains impacted by -12%. The distribution of EWPC acceleration across the benchmarks highlights the consistent effectiveness of GoPTX in enhancing ILP over existing techniques, and thus translates to performance gains, as demonstrated by the overall speedup presented in Figure 10.

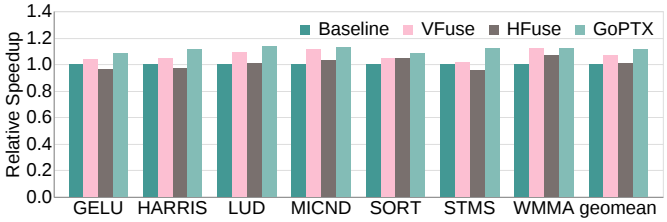



Fig. 10: Performance improvement.

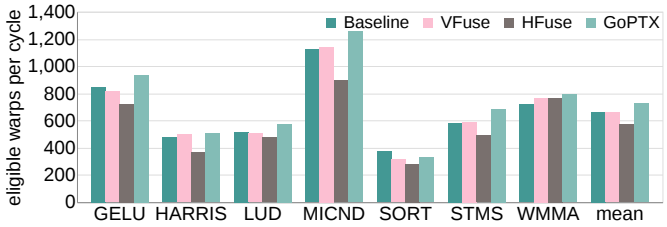



Fig. 11: Eligible warps per cycle comparison.

### B. Resource Utilization

Figure 12 shows that VFuse and GoPTX improve occupancy (AOC) by 4% compared to the baseline, while HFuse suffers a 5.7% decrease because it launches more warps in a block waiting for others to finish. This suggests that register pressure did not significantly impact performance, as modern GPU compilers use advanced register allocation strategies to optimize even with high register usage. For example, *WMMA* uses 40 registers per block (75% theoretical occupancy), while *HARRIS* uses 14 registers (100% occupancy). Our weaved kernel uses 40 registers (75% occupancy). VFuse and GoPTX improve off-chip resource utilization, e.g DU by 3.7% and 6.5%, respectively, while HFuse shows no change. GoPTX also achieves the highest instruction throughput (ISU) increase (4.0%). For on-chip resources, VFuse and HFuse show increments but suffer from inconsistent performance gains due to warp competition. Overall, GoPTX's ability to improve ILP and hide latency leads to the best resource utilization and ISU among all evaluated techniques.

### C. Case Study of Stalls

We analyze the impact of instruction weaving on warp stalls using *WMMA+GELU* and *WMMA+HARRIS*. These workloads share a common kernel (WMMA) but exhibit significantly different performance improvements. GoPTX achieves a 20% performance improvement and a 38% increase in EWPC for *WMMA+HARRIS*,

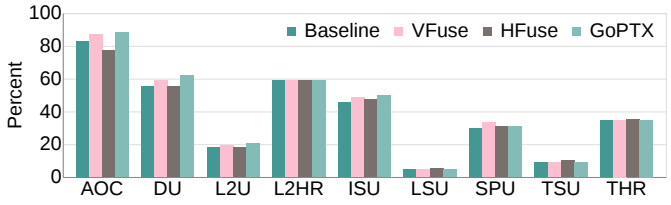



Fig. 12: Hardware resource utilization comparison.

while experiencing a -2% performance degradation and a -35% decrease in EWPC for *WMMA+GELU*. As Figure 13 illustrates, after `GoPTX` processing, *WMMA+HARRIS* experiences a 20.5% reduction in scoreboard stalls, while *WMMA+GELU* incurs a 30.4% increase (still better than `VFuse` and `HFuse`). For total stall cycles, *WMMA+HARRIS* sees a 34.5% reduction, while *WMMA+GELU* remains unchanged. This disparity can be attributed to the differing hardware resource contention between the kernels involved, as Table I shows. Both WMMA and GELU involve a significant amount of half-precision computation, leading to contention for ALU resources. In contrast, WMMA and HARRIS (primarily integer computations) exhibit complementary computation types. This result highlights the importance of resource complementarity for ILP enhancement.

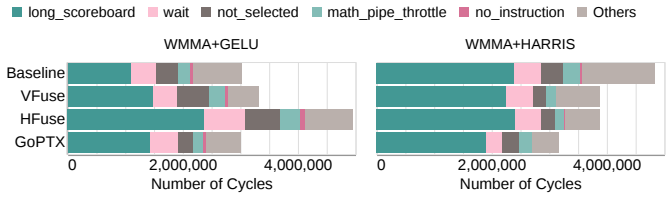



Fig. 13: Warp stalled cycles and reasons.

### D. Performance Breakdown

*a) Contributions of Merging, Weaving and Slicing:* To isolate the effects of each technique, we conducted a series of experiments where we selectively enabled weaving and slicing. Even without both, control flow optimization alone achieves a performance improvement of 9.7%. This enhancement is attributed to the effective sharing of resources within the SM and the instruction reordering capabilities of the `ptxas` backend. Slicing incurs a slight decrease of 0.6% while creating opportunities for weaving to achieve a significant performance improvement of 2.1%. Without slicing, weaving only improves 0.3%. These findings underscore the necessitity of both weaving and slicing in facilitating instruction parallelism.

*b) Threshold for Code Slicing:* We compared our adaptive slicing algorithm with fixed-threshold approaches, where the slicing threshold ranged from 256 to 1024. As evident from Figure 14, the effect of the threshold varies significantly and no single value is universally optimal. Our adaptive algorithm outperforms the fixed-threshold approach in most cases. However, the WMMA benchmark exhibits better performance with the fixed-threshold approach due to the presence of tensor core instructions, which occur at fixed intervals. This highlights the sensitivity of the fixed-threshold approach to workload characteristics. In contrast, our adaptive approach dynamically adjusts the slicing threshold, making it more robust and adaptable to a wider range of workloads.

## VI. Related Work

*a) Concurrent Kernel Execution:* Methods based on cuStream, MPS, or MIG [23], [26], [27] focus on scheduling and optimizing

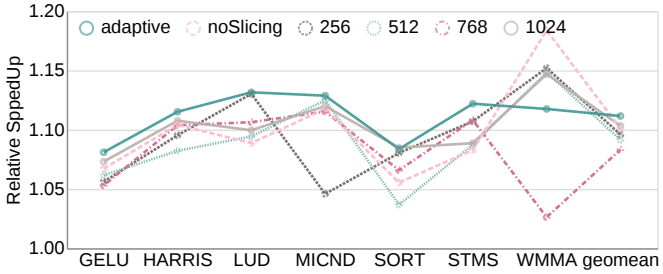



Fig. 14: Speedup of the threshold for code slicing, compared to our adaptive algorithm.

multiple kernels but don't fully address resource under-utilization [3], [28], [29]. Xu *et al.* propose Warped-Slicer [28], which dynamically and statically schedules thread blocks from different kernels to a single SM. Wang *et al.* introduce SMK [29], which uses preemption for block-level scheduling by efficiently dividing resources. Although these approaches can enhance utilization, Dai *et al.* [3] have shown that intra-SM sharing schemes may reduce overall performance due to substantial interference between kernels.

*b) Code Combination Techniques:* Guevara *et al.* [30] and Gregg *et al.* [31] combine kernels at the thread block level, which can incur significant overhead. Wang *et al.* [13] suggest three thread-level fusion strategies, but their method cannot handle synchronization [4], [13]. Li *et al.* [14] present automatic horizontal fusion, allowing GPUs to distribute instructions to different warps. Even with the above designs, severe kernel interference remains unresolved, and our evaluation demonstrates that `GoPTX` outperforms those prior designs on parallelism enhancement.

*c) GPU ILP Optimization:* Shobaki *et al.* [11] propose a compile-time Branch-and-Bound algorithm to balance ILP and occupancy, while their later work [12] parallelizes instruction scheduling, further enhancing occupancy and reducing schedule length. WASP [32] introduces warp specialization to enable parallel tasks within a block, boosting utilization. These approaches generally overlook resource sharing and latency hiding, thereby limiting ILP and utilization across kernels constrained by hardware units, while our fine-grained `GoPTX` method addresses this by enabling resource sharing to enhance ILP.

## VII. Summary

Targeting at raising instruction-level parallelism of GPU executions, this paper presents `GoPTX` to weave instructions from two different kernels, serving as a fine-grained kernel fusion at PTX level. The design overall encompasses CFG merging, instruction weaving and code slicing to generate highly efficient codes for better utilization of the underlying resources. We implement the prototype of `GoPTX` based on NVIDIA off-the-shelf CUDA software stack, with particular handling of race conditions and deadlock. Experimental results on representative kernels demonstrate that `GoPTX` effectively improves performance with higher ILP and utilization.

## Acknowledgment

We thank the anonymous reviewers for their constructive comments and suggestions. This research was supported by the National Natural Science Foundation of China-#62472462/#62461146204, and sponsored by CCF-Tencent Rhino-Bird Open Research Fund (CCF-Tencent RAGR20240102).

## REFERENCES


[1] "NVIDIA Multi-Process Service (MPS)." http://docs.nvidia.com/deploy/mps/index.html.

[2] "NVIDIA Multi-Instance GPU (MIG)." https://www.nvidia.com/en-sg/technologies/multi-instance-gpu/.

[3] H. Dai, Z. Lin, C. Li, C. Zhao, F. Wang, N. Zheng, and H. Zhou, "Accelerate gpu concurrent kernel execution by mitigating memory pipeline stalls," in 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 208–220, 2018.

[4] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving gpgpu concurrency with elastic kernels," in Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, (New York, NY, USA), p. 407–418, Association for Computing Machinery, 2013.

[5] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski, G. Chauhan, A. Chourdia, W. Constable, A. Desmaison, Z. DeVito, E. Ellison, W. Feng, J. Gong, M. Gschwind, B. Hirsh, S. Huang, K. Kalambarkar, L. Kirsch, M. Lazos, M. Lezcano, Y. Liang, J. Liang, Y. Lu, C. K. Luk, B. Maher, Y. Pan, C. Puhrsch, M. Reso, M. Saroufim, M. Y. Siraichi, H. Suk, S. Zhang, M. Suo, P. Tillet, X. Zhao, E. Wang, K. Zhou, R. Zou, X. Wang, A. Mathews, W. Wen, G. Chanan, P. Wu, and S. Chintala, "Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation," in Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS '24, (New York, NY, USA), p. 929–947, Association for Computing Machinery, 2024.

[6] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: a system for large-scale machine learning," in Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'16, (USA), p. 265–283, USENIX Association, 2016.

[7] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: an automated end-to-end optimizing compiler for deep learning," in Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation, OSDI'18, (USA), p. 579–594, USENIX Association, 2018.

[8] Z. Ye, R. Lai, J. Shao, T. Chen, and L. Ceze, "Sparsetir: Composable abstractions for sparse compilation in deep learning," in Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, (New York, NY, USA), p. 660–678, Association for Computing Machinery, 2023.

[9] Y. Gui, Y. Wu, H. Yang, T. Jin, B. Li, Q. Zhou, J. Cheng, and F. Yu, "Hgl: accelerating heterogeneous gnn training with holistic representation and optimization," in SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–15, IEEE, 2022.

[10] H. Zhao, W. Cui, Q. Chen, Y. Zhang, Y. Lu, C. Li, J. Leng, and M. Guo, "Tacker: Tensor-cuda core kernel fusion for improving the gpu utilization while ensuring qos," in 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA), pp. 800–813, 2022.

[11] G. Shobaki, A. Kerbow, and S. Mekhanoshin, "Optimizing occupancy and ilp on the gpu using a combinatorial approach," in Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization, CGO 2020, (New York, NY, USA), p. 133–144, Association for Computing Machinery, 2020.

[12] G. Shobaki, P. Muyan-Özçelik, J. Hutton, B. Linck, V. Malyshenko, A. Kerbow, R. Ramirez-Ortega, and V. Gordon, "Instruction scheduling for the gpu on the gpu," pp. 435–447, 03 2024.

[13] G. Wang, Y. Lin, and W. Yi, "Kernel fusion: An effective method for better power efficiency on multithreaded gpu," in 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing, pp. 344–350, 2010.

[14] A. Li, B. Zheng, G. Pekhimenko, and F. Long, "Automatic horizontal fusion for gpu kernels," in 2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 14–27, 2022.

[15] J. Wu, A. Belevich, E. Bendersky, M. Heffernan, C. Leary, J. Pienaar, B. Roune, R. Springer, X. Weng, and R. Hundt, "gpucc: an open-source gpgpu compiler," in Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO '16, (New York, NY, USA), p. 105–116, Association for Computing Machinery, 2016.

[16] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, CGO '04, (USA), p. 75, IEEE Computer Society, 2004.

[17] A. community, "grammars-v4." https://github.com/antlr/grammars-v4/blob/753536777d827ccc0c9b108531ea67375c2039ac/asm/ptx/ptx-isa-2.1/Ptx.g4, 2023.

[18] H. Abdelkhalik, Y. Arafa, N. Santhi, and A.-H. A. Badawy, "Demystifying the nvidia ampere architecture through microbenchmarking and instruction-level analysis," 2022 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–8, 2022.

[19] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in 2009 IEEE International Symposium on Workload Characterization (IISWC), pp. 44–54, 2009.

[20] ONNX Runtime developers, "ONNX Runtime," Nov. 2018.

[21] NVIDIA, "Cuda samples." https://github.com/NVIDIA/cuda-samples/tree/v12.5, 2024.

[22] B. Qiao, O. Reiche, F. Hannig, and J. Teich, "From loop fusion to kernel fusion: A domain-specific approach to locality optimization," in 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 242–253, 2019.

[23] J. Kim, J. Kim, and Y. Park, "Navigator: Dynamic multi-kernel scheduling to improve gpu performance," in 2020 57th ACM/IEEE Design Automation Conference (DAC), pp. 1–6, 2020.

[24] H. Wu, Y. Yu, J. Deng, S. Ibrahim, S. Wu, H. Fan, Z. Cheng, and H. Jin, "StreamBox: A lightweight GPU SandBox for serverless inference workflow," in 2024 USENIX Annual Technical Conference (USENIX ATC 24), (Santa Clara, CA), pp. 59–73, USENIX Association, July 2024.

[25] NVIDIA, "Cupti 12.5 documentation." https://docs.nvidia.com/cupti/index.html, 2024.

[26] J. Zhong and B. He, "Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling," IEEE Transactions on Parallel and Distributed Systems, vol. 25, no. 6, pp. 1522–1532, 2014.

[27] Z. Lin, Z. Mo, X. Huang, X. Zhang, and Y. Lu, "Kesco: Compiler-based kernel scheduling for multi-task gpu applications," in 2023 IEEE 41st International Conference on Computer Design (ICCD), pp. 247–254, IEEE, 2023.

[28] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram, "Warped-slicer: Efficient intra-sm slicing through dynamic resource partitioning for gpu multiprogramming," ACM SIGARCH Computer Architecture News, vol. 44, no. 3, pp. 230–242, 2016.

[29] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing," in 2016 IEEE international symposium on high performance computer architecture (HPCA), pp. 358–369, IEEE, 2016.

[30] M. Guevara, C. Gregg, K. M. Hazelwood, and K. Skadron, "Enabling task parallelism in the cuda scheduler," 2009.

[31] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron, "Fine-grained resource sharing for concurrent gpgpu kernels," in Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism, HotPar'12, (USA), p. 10, USENIX Association, 2012.

[32] N. C. Crago, S. Damani, K. Sankaralingam, and S. W. Keckler, "Wasp: Exploiting gpu pipeline parallelism with hardware-accelerated automatic warp specialization," in 2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA), pp. 1–16, 2024.