Kernel instruction optimization based on the triton compiler

Lin Han[1st]
Zhengzhou University, Zhengzhou, Henan, China, 450001

Yubo Du[2nd]
Zhengzhou University, Zhengzhou, Henan, China, 450001

Jianan Li[3rd]
Zhengzhou University, Zhengzhou, Henan, China, 450001

Wei Gao[4th, *]
National Supercomputing Center in Zhengzhou,
Zhengzhou, Henan, China, 450001
Corresponding author's email: yongwu22@126.com

2025 6th International Conference on Computer Engineering and Application (ICCEA) | 979-8-3315-4330-3/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICCEA65460.2025.11102282

***Abstract*—With the continuous growth of model parameter scale and computational density, kernel execution performance has gradually become an important part of overall model inference and training. The programming language Triton, developed by OpenAI, can serve as the backend of Torch Inductor to assist in generating high-performance GPU kernels. Aiming at the problem of low-quality kernels in the Inductor kernel generation process, the Triton compiler can introduce optimization strategies such as redundant instruction elimination. However, when dealing with program features such as complex memory dependencies, repeated loading, and multiple consecutive division operations, there are still issues of insufficient optimization depth and inadequate recognition and mitigation of performance bottlenecks. Therefore, this paper proposes a unified optimization strategy targeting the intermediate representation level based on the Triton compiler. Through the combination of static analysis and equivalence rewriting, it performs fused optimization on instruction redundancy and algebraic computation patterns. Under the premise of preserving semantic consistency, the method restructures data dependency paths and kernel expression forms, effectively reducing redundant memory accesses, decreasing repetitive and inefficient computation overhead, and improving kernel execution efficiency. Experimental results show that the average speedup of redundant instruction optimization reaches 1.47, while the average speedup of algebraic transformation optimization reaches 1.29.**

Keywords-Redundancy Elimination; Compiler Optimization; Algebraic Transformation; Triton; MLIR;

I. INTRODUCTION

With the continuous expansion of deep learning model scales and the increasing complexity of training, GPU compilers face increasingly stringent performance challenges in generating efficient executable code[1]. Torch Inductor adopts the Triton compiler to generate high-performance GPU kernels in order to improve kernel execution efficiency. However, an actual analysis of Triton kernels generated from models such as fastNLP_Bert and BERT_PYTORCH reveals that some kernels still suffer from redundant operations and suboptimal execution efficiency. This is especially true for reduction-type kernels, which, after loop unrolling, often include multiple load instructions that repeatedly access the same memory address but with differing attributes, as well as multiple consecutive division operations and redundant reduce operations. These redundant instructions and inefficient division operations at the kernel level have become key bottlenecks affecting model execution performance[2]. On one hand, redundant instructions frequently appear in automatically generated computational graphs, leading to memory bandwidth waste and repeated computation[3]; on the other hand, GPUs lack dedicated hardware for division, which must be implemented via iterative algorithms, resulting in significantly higher latency than multiplication, thus further constraining performance[4].

In recent years, researchers have proposed various optimization strategies to improve the performance of GPU compilers by reducing redundant computation and memory access overhead. Li J et al. eliminated redundant computation by optimizing the computation graph[5]. Katel N rewrote the intermediate representation using the MLIR framework, replacing division with multiplication to reduce latency[6]. Jia Z proposed optimizing memory access patterns to remove redundant load operations[7]. and Kumar R integrated data flow analysis with automatic code generation techniques to merge repeated divisions and improve operator throughput[8]. These studies demonstrate that instruction-level restructuring and memory access optimization at the operator level play a critical role in compiler performance improvement.

Against this background, the Triton compiler developed by OpenAI is employed by Torch Inductor as a backend to generate high-performance GPU kernels[9]. As shown in Figure 1. the execution pipeline integrates multiple intermediate representations (Triton Dialect and TritonGPU Dialect) with the LLVM backend and supports optimization strategies such as CSE and DCE[10]. However, empirical analysis reveals that Triton still has blind spots when handling complex memory dependencies and sequential algebraic expressions. For instance, it fails to merge redundant load and equivalent reduce operations, and does not rewrite multi-level divisions, thereby limiting further kernel performance improvements[11]. To address these issues, this paper proposes a unified optimization strategy at the intermediate representation level, combining static analysis with equivalence rewriting. Two optimization passes are introduced at the TritonGPU Dialect layer: elimination of redundant Load/Reduce instructions and transformation of sequential divisions into multiplications. Experimental results demonstrate that the proposed strategy achieves significant speedups on models such as fastNLP_Bert and BERT_PYTORCH, showing strong generality and effectiveness.

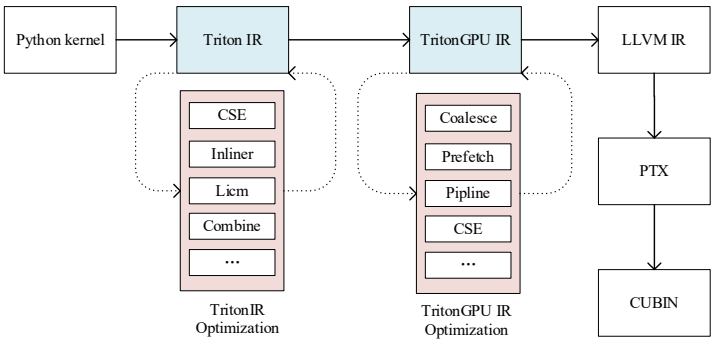

Figure 1. Execution flow of the triton compiler

II. Redundant Instruction OptimizatioN

A. Redundant instruction performance bottleneck analysis

In the compilation and execution process of deep learning models, instruction-level optimization is often obscured by complex computation graphs and underlying kernel structures. Taking the fastNLP_Bert model under the TorchBench framework as an example, although the Triton kernel generated by PyTorch Inductor has undergone some automatic optimization, redundant instructions still exist, especially in reduction-type kernels. It was found that the load instructions in each for-loop access the same memory address, but due to differences in properties such as eviction_policy, the Triton compiler failed to merge them. Triton uses the CSE optimization strategy in the MLIR framework, but this mechanism does not fully account for differences in memory access properties, leading to the failure to eliminate redundancy effectively. Additionally, some reduce operations were not correctly merged, further exacerbating redundant computations, increasing memory bandwidth pressure, and reducing kernel throughput.

B. Redundant instruction pass implementation

The redundancy elimination process is illustrated in Figure 2. This process extracts all Load and Reduce instructions from the generated TTGIR. For Load instructions, redundancy is identified by checking whether there are operations that load from the same address with identical values; for Reduce instructions, redundancy is determined based on whether the computation logic is equivalent. Once the conditions are satisfied, redundant references are replaced, redundant instructions are removed, and the optimized TTGIR is generated, followed by the output of the target executable code.

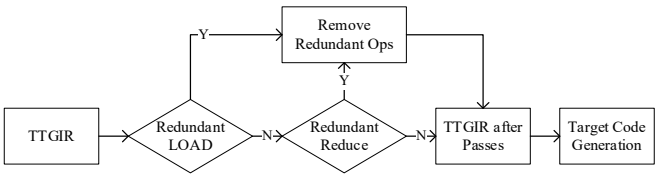

Figure 2. Redundant instruction optimization workflow.

To provide a clearer view of the optimization effect, Table I presents a comparison between TTGIR fragments before and after transformation. In the original version, two Load operations access the same memory address but differ in their eviction_policy attributes, along with two logically equivalent Reduce operations (%6 and %7). After optimization, the redundant Load and Reduce instructions are successfully merged, retaining only %2 and %6, with all dependent references correctly updated.

TABLE I. Structural changes in ttgir before and after redundant instruction optimization.

Ttgir before optimization:	Ttgir after optimization:
%2=tt.load(ptr, evict=3) %3=tt.load(ptr, evict=1) %6=tt.reduce(%2, op=add) -> … %7=tt.reduce(%3, op=add) -> …	%2=tt.load(ptr, evict=3) %6=tt.reduce(%2, op=add) -> …

The key to redundant instruction optimization is to identify and remove duplicate or ineffective instructions while maintaining the program's semantics and data dependencies. The specific process includes traversing all Load and Reduce operations. For Load instructions, if two Load instructions access the same address and there is no Store operation modifying the address in between, the latter is considered redundant, to be deleted and replaced with the result of the former; if a Store operation modifies the address, it is considered non-redundant. For Reduce instructions, if the reduction dimensions, types, return value types, and reduction body structures of two operations are identical, the latter is considered redundant, to be deleted and replaced with the result of the former. Through this optimization, unnecessary memory access and computation overhead can be reduced, with the core logic as shown in Algorithm 1.

Algorithm 1: Redundant instruction optimization

```
for op in allOps do
  nextOp = op.getNextNode()
  if loadA = dyn_cast<LoadOp>(op) then// Redundant Load Instruction O
ptimization
    while nextOp != null do
     if store = dyn_cast<StoreOp>(nextOp) and Equivalent(store.getPtr, l
oadA.getPtr) then
       break  //There exists a Store that rewrites the load address
     if loadB = dyn_cast<LoadOp>(nextOp) and areOperandsEqual(load
A, loadB) then
       loadB.getResult().replaceAllUsesWith(loadA.getResult())
     nextOp = nextOp.getNextNode()
  if reduceA = dyn_cast<ReduceOp>(op) then // Redundant Reduce Instr
uction Optimization
    while nextOp != null do
     if reduceB = dyn_cast<ReduceOp>(nextOp) and SameOpLogic(redu
ceA, reduceB) then
       reduceB.getResult().replaceAllUsesWith(reduceA.getResult())
     nextOp = nextOp.getNextNode()
```

III. ALGEBRAIC TRANSFORMATION OPTIMIZATION

A. Algebraic transformation performance bottleneck analysis

In automatically generated code scenarios, algebraic transformation is commonly used for performance optimization, especially in programs with control dependencies, where existing compiler optimization strategies are often insufficient. Taking the Triton kernel generated by PyTorch Inductor from the BERT_PYTORCH model as an example, it is observed that the division instructions in the kernel exhibit strong data dependencies. For instance, the result of the first division is used solely as the input to the second division and is not referenced by any subsequent operations. Such patterns can be optimized through division transformation. In parallel programming, multiplication is typically a single-cycle operation, and fused multiply-add (FMA) instructions on GPUs like the A100 execute with high efficiency. In contrast, division lacks dedicated hardware support and must be implemented via iterative algorithms such as Newton-Raphson or Goldschmidt, resulting in significantly higher latency. Particularly in scenarios with strong dependency chains, division often becomes a bottleneck for GPU execution efficiency, making it difficult to fully utilize computational resources even in highly parallel architectures.

B. Algebraic transformation pass Implementation

The optimization process of division transformation is illustrated in Figure 3. After generating the TTGIR, the compiler first checks whether the kernel contains multiple division operations. If so, it further analyzes data dependencies and structural rewritability. When the conditions are met, the compiler performs division-to-multiplication transformations, updates the TTGIR, and proceeds with code generation to enhance execution performance. If the conditions are not satisfied, the transformation step is skipped, and the compilation continues with standard code generation.

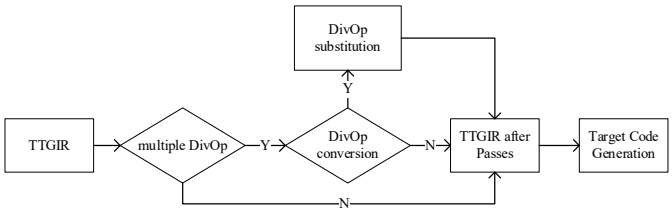

Figure 3. Division transformation optimization workflow.

This optimization process can be applied to various patterns of division chains. To better illustrate the transformation of the intermediate representation (IR) before and after optimization, Table II presents two typical cases of division rewriting. In these examples, the original IR contains two-level nested divf operations. By analyzing the underlying data dependencies, the transformation successfully rewrites them into a combination of mulf and divf operations. This replacement effectively shortens the critical path of the division chain, thereby improving execution efficiency and enhancing parallelism.

TABLE II. STRUCTURAL CHANGES IN TTGIR BEFORE AND AFTER DIVISION TRANSFORMATION OPTIMIZATION.

Test	Ttgir before optimization:	Ttgir after optimization:
case1	%24=divf %20,%23:tensor %25=divf %24,%22:tensor	%24=mulf %23,%22:tensor %25=divf %20,%24:tensor
case2	%24=divf %20,%23:tensor %25=divf %22,%24:tensor	%24=mulf %23,%22:tensor %25=divf %24,%20:tensor

The division transformation optimization is implemented through pattern matching and rewriting. The specific steps are as follows: Traverse all division operations in the kernel and check if the result of a division is used only by a single subsequent division instruction; if so, perform IR rewriting. If the result of the first division is the dividend of the second division, a new multiplication operator is constructed with the divisors of the two divisions as inputs, and a new division instruction is created using the multiplication result as the divisor. The dividend of the new division is the dividend of the first division. If the result of the first division is the divisor of the second division, the operand order is adjusted, with the same processing logic. Finally, the new division replaces the second division and its users, and the first division is deleted, completing the optimization. The core algorithm of the division transformation optimization is shown in Algorithm 2.

Algorithm 2: Division transformation optimization

```
for op in allOps do
//Optimize consecutive division expressions:a/b/c -> a/(b*c)
 if divA = dyn_cast<DivOp>(op) and divA.hasOneUse() then
   user = divA.getSingleUser()
   if divB = dyn_cast<DivOp>(user) then
     if divB.getLhs == divA.getResult then // divA result as the dividend
       mulOp = rewriter.create<arith::MulFOp>(divB.getLoc, divA.getRh
s, divB.getRhs)
       newDiv = rewriter.create<arith::DivFOp>(divB.getLoc, divA.getL
hs, mulOp.getResult)
     if divB.getRhs == divA.getResult then  // divA result as the divisor
       mulOp = rewriter.create<arith::MulFOp>(divB.getLoc, divA.getRh
s, divB.getLhs)
       newDiv = rewriter.create<arith::DivFOp>(divB.getLoc, mulOp.get
Result, divA.getLhs)
     rewriter.replaceOp(op, mulOp.getResult)
     divB.replaceAllUsesWith(newDiv.getResult)
     rewriter.eraseOp(divUse)
```

IV. EXPERIMENTAL EVALUATION

The experiments in this work are conducted on a domestic GPU platform. Under the premise of passing accuracy verification, we evaluate and analyze the performance improvement brought by the redundant instruction elimination and division transformation optimizations. The software environment includes OpenAI Triton version 2.1.0, PyTorch version 2.1.2, and LLVM version 17.0.0.

A. Performance metrics

Before performance evaluation, this paper verifies the accuracy of kernel outputs before and after optimization to ensure that the application of redundant instruction elimination and division transformation does not affect numerical correctness. The validation is conducted using element-wise error comparison, with the maximum relative error constrained within 1×10^{-4}, ensuring that the optimization affects only performance and not correctness. On this basis, acceleration ratio is adopted as the performance metric, evaluated as the ratio

of kernel execution time before optimization (T_{before}) to that after optimization (T_{after}). The final average acceleration ratio S is computed over 50 consecutive runs, as shown in Equation.

$$S = \frac{1}{50} \sum_{k=0}^{50} \frac{T_{before}}{T_{after}} \quad (1)$$

B. *Redundant instruction evaluation*

The Triton kernels generated from multiple deep learning models are collected to construct a test set for redundant instruction optimization. All tests are conducted using a unified fp16 data type and cover a range of tensor sizes. Prior to performance evaluation, all kernels undergo accuracy verification to ensure consistency of computation results before and after optimization. Table III summarizes the tensor sizes, number of redundant instructions, and other details for each test case.

TABLE III. REDUNDANT INSTRUCTION TEST SET

Mode Name	Test Cases	Data Scale	R-Load	R-Reduce
coat_lite_mini	triton_red_8	[8192,3137]	1	0
fastNLP_Bert	triton_red_3	[182784,476]	1	1
fastNLP_Bert	triton_red_7	[65536,2048]	1	1
eca_botnet26ts	triton_red_24	[131072,256]	3	0
coat_lite_mini	triton_red_29	[16384,785]	1	0
convit_base	triton_red_5	[401408,196]	1	0
botnet26t_256	triton_red_25	[131072,256]	2	0
beit_base_patch16	triton_red_6	[302592,197]	4	0

The performance evaluation results before and after redundant instruction optimization are illustrated in Figure 4. The figure presents the average execution time across 50 consecutive runs for each test case, along with the corresponding acceleration ratio. It can be observed that performance gains exhibit a positive correlation with both the number of redundant instructions and the size of input data: the more redundant instructions present and the larger the data scale, the more significant the speedup. These results indicate that the proposed optimization method demonstrates strong adaptability and scalability in improving the execution efficiency of Triton kernels.

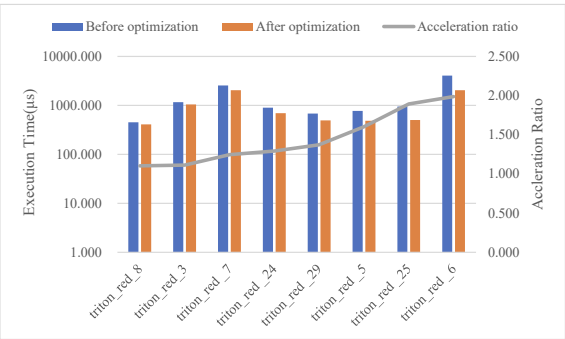

Figure 4. Redundant instruction optimization results.

C. *Algebraic transformation evaluation*

This study takes triton_per_7 and triton_per_8, generated during the compilation of the BERT_pytorch model using PyTorch Inductor, as test cases to evaluate the impact of algebraic transformation optimization on program performance. The tests uniformly adopt the fp32 data type, and the adaptability and effectiveness of the optimization are analyzed by adjusting the size of input tensors. All test cases have passed precision verification. The division transformation test set in Table IV summarizes the data scale, number of division operations, and related information for each test case.

TABLE IV. DIVISION TRANSFORMATION TEST SET

Model Name	Test Cases	Data Scale	Count of Division
BERT_pytorch	triton_per_7	[65536,256]	2
		[262114,512]	
		[1048576,1024]	
		[16384,2560]	
		[1024,3200]	
	triton_per _8	[4096,1330]	3
		[4096,640]	
		[16384,1280]	
		[16384,768]	
		[65536,2000]	

The performance test results before and after the division transformation optimization are shown in Figure 5. The results indicate that the acceleration effect of the Triton kernel is positively correlated with the number of division instructions: when the kernel contains two division operations, the average acceleration ratio is 1.13; when the number increases to three, the acceleration ratio rises to 1.29, significantly reducing execution time. The effectiveness of this optimization is mainly due to the limited hardware support for division on GPUs, which typically rely on iterative algorithms to perform division, resulting in much higher latency than multiplication. By rewriting consecutive division operations into a combination of multiplication, this optimization not only reduces the usage of high-latency operations but also improves the data dependency path, thereby enhancing execution throughput. Especially under large-scale input scenarios, this strategy demonstrates notable performance gains and adaptability.

Figure 5. Redundant Instruction Optimization Results.

V. Conclusions

This paper proposes a unified intermediate representation based optimization strategy to address the issues of redundant instructions and inefficient division operations in the Triton compiler. By leveraging static analysis and equivalence rewriting, the approach eliminates redundant Load/Reduce instructions and rewrites consecutive divisions, effectively alleviating memory and computation bottlenecks. Experimental results demonstrate significant performance improvements across various mainstream models, indicating strong generalizability. Future work will extend this strategy to the Torch Inductor frontend to further enhance the overall compilation system efficiency.

Acknowledgment

This research was supported by the 2023 National Key R&D Program of China (High-Performance Computing Special Project): High-Performance Cosmological Simulation Software for New-Generation Domestic Supercomputing Systems (Grant No. 2023YFB3002505). We extend our heartfelt thanks to them for their support.

References

[1] Kaplan J, McCandlish S, Henighan T, et al. Scaling laws for neural language models[J]. arXiv preprint arXiv:2001.08361, 2020. https://doi.org/10.48550/arXiv.2001.08361.

[2] Chen R, Ding Z, Zheng S, et al. Magis: Memory optimization via coordinated graph transformation and scheduling for dnn[C]//Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3. 2024: 607-621. https://dl.acm.org/doi/abs/10.1145/3620666.3651330.

[3] Wei R, Schwartz L, Adve V. A modern compiler infrastructure for deep learning systems with adjoint code generation in a domain-specific IR[J]. 2017. https://openreview.net/forum?id=SJo1PLzCW.

[4] Ngo D, Ahn S, Son J, et al. Accelerating Pattern Recognition with a High-Precision Hardware Divider Using Binary Logarithms and Regional Error Corrections[J]. Electronics, 2025, 14(6): 1066. https://doi.org/10.3390/electronics14061066.

[5] Li J, Qin Z, Mei Y, et al. onednn graph compiler: A hybrid approach for high-performance deep learning compilation[C]//2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE, 2024: 460-470. https://doi.org/10.1109/CGO57630.2024.10444871.

[6] Katel N, Khandelwal V, Bondhugula U. MLIR-based code generation for GPU tensor cores[C]//Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction. 2022: 117-128. https://doi.org/10.1145/3497776.3517770.

[7] Jia Z, Padon O, Thomas J, et al. TASO: optimizing deep learning computation with automatic generation of graph substitutions [C]//Proceedings of the 27th ACM Symposium on Operating Systems Principles. 2019: 47-62. https://dl.acm.org/doi/abs/10.1145/3341301.3359630.

[8] Kumar R, Negi K C, Sharma N K, et al. Deep Learning-Driven Compiler Enhancements for Efficient Matrix Multiplication[J]. Journal of Computers, Mechanical and Management, 2024, 3(2): 08-18. https://doi.org/10.57159/gadl.jcmm.3.2.240122.

[9] Mishra P, Gale T, Zaharia M, et al. Implementing block-sparse matrix multiplication kernels using Triton[C]//Workshop on Efficient Systems for Foundation Models@ ICML2023.https://openreview.net /forum?id=doa11nN5vG.

[10] Tillet P, Kung H T, Cox D. Triton: an intermediate language and compiler for tiled neural network computations[C]//Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages. 2019: 10-19. https://doi.org/10.1145/3315508.3329973.

[11] Kim H, Ahn S, Oh Y, et al. Duplo: Lifting redundant memory accesses of deep neural networks for gpu tensor cores[C]//2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2020: 725-737. https://doi.org/10.1109/MICRO50266.2020.00065.