2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-5733-2/25/$31.00 ©2025 IEEE | DOI: 10.1109/ASE63991.2025.00185

Enhancing LLM to Decompile Optimized PTX to Readable CUDA for Tensor Programs

Xinyu Sun[1], Fugen Tang[1], Yu Zhang[1*], Han Shen[2], Chengru Song[2], Di Zhang[2]
[1]University of Science and Technology of China, Hefei, China
[2]Kuaishou Technology, Beijing, China
yuzhang@ustc.edu.cn[1], shenhan03@kuaishou.com[2]

***Abstract*—The growing demand for high-performance tensor programs on GPUs, especially for large language models (LLMs), necessitates advanced compilation and optimization techniques. However, the critical task of analyzing optimized, low-level PTX code for performance tuning or understanding poses significant challenges. While LLMs hold promise for PTX-to-CUDA decompilation to improve code intelligibility, their effectiveness is severely limited by the scarcity of aligned training data and the inherent complexity of highly optimized, unrolled PTX code.**

In this work, we explore methodologies to significantly enhance LLM capabilities for accurate and readable PTX-to-CUDA decompilation and present PtxDec, a decompilation prototype implementing our approach. To overcome the critical barrier of data scarcity, we develop a compiler-based data augmentation framework coupled with rigorous post-processing, enabling the creation of a large-scale, high-quality dataset of 400K aligned CUDA-PTX kernel pairs for effective LLM training. Furthermore, to empower LLMs to handle the complexity of optimized PTX, we introduce Rolled-PTX—an intermediate representation generated through heuristic loop rerolling during preprocessing. Rolled-PTX condenses unrolled patterns, drastically simplifying the input structure presented to the LLM and aligning it better with higher-level loop constructs.

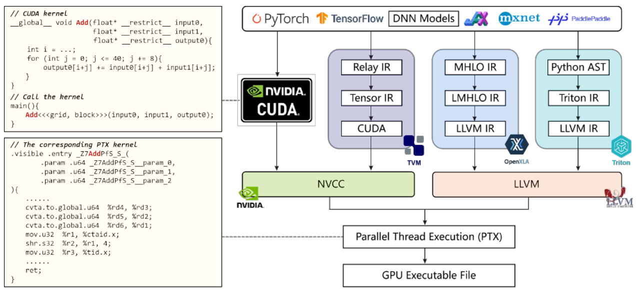
Comprehensive evaluation demonstrates that PtxDec achieves substantial performance gains: our approach yields a 2.3×–3.1× improvement in functional accuracy over baseline methods, alongside significant enhancements in generated code readability and scheduling consistency with the original optimized kernels. Ablation studies further validate the contribution of each proposed component to the overall performance.

To the best of our knowledge, this is the first work tackling PTX-to-CUDA decompilation, specifically focusing on and demonstrating effective strategies for augmenting LLMs to overcome the key challenges in this domain.

***Index Terms*—LLM, Decompilation, Deep learning compiler, GPU Programming**

I. Introduction

Decompilation is the reverse process of converting low-level code back into a high-level programming language. It aids various reverse engineering tasks, such as vulnerability identification, malware research, and legacy software migration. Existing decompilation tools (e.g., IDA Pro [1], Ghidra [2]) and research [3–13] primarily target CPUs, while GPU decompilation has received limited attention. However, we observe that the need for GPU decompilation has been growing steadily in recent years, primarily driven by increasing demands for high-performance tensor programs.



```
// CUDA kernel
__global__ void Add(float* __restrict__ input0,
                    float* __restrict__ input1,
                    float* __restrict__ output0){
    int i = ...;
    for (int j = 0; j <= 40; j += 8){
        output0[i+j] += input0[i+j] + input1[i+j];
    }
}
// Call the kernel
main(){
    Add<<<grid, block>>>(input0, input1, output0);
}
```

```
// The corresponding PTX kernel
.visible .entry _Z7AddPfS_S_(
        .param .u64 _Z7AddPfS_S__param_0,
        .param .u64 _Z7AddPfS_S__param_1,
        .param .u64 _Z7AddPfS_S__param_2
){
    ......
    cvta.to.global.u64  %rd4, %rd3;
    cvta.to.global.u64  %rd5, %rd2;
    cvta.to.global.u64  %rd6, %rd1;
    mov.u32  %r1, %ctaid.x;
    shr.s32  %r2, %r1, 4;
    mov.u32  %r3, %tid.x;
    ......
    ret;
}
```

Fig. 1: Compilation Pathways for GPU Tensor Programs, along with code examples of CUDA and PTX

Deep learning compilers are widely used to generate high-performance GPU tensor programs. Developing and optimizing these compilers remains highly specialized and challenging. To perform tasks like compiler behavior analysis, problem diagnosis, and performance optimization, developers often must work directly with low-level code of low readability, posing a significant challenge. As shown in Figure 1, these compilers (e.g., TVM [14], OpenXLA [15], Triton [16]) process Python-defined models from deep learning frameworks (e.g., TensorFlow [17], PyTorch [18]) and progressively lower them through multi-level intermediate representations (IRs), ultimately generating Parallel Thread Execution (PTX) [19] that is subsequently compiled into hardware-specific executable files. We note that PTX is an essential step for major deep learning compilers generating GPU tensor programs. Translating low-readability PTX into equivalent, high-readability CUDA would significantly facilitate their development and optimization.

Moreover, the substantial performance requirements of large language models have driven developers to implement extreme low-level optimizations at the PTX level, as demonstrated in DeepSeek-V3's optimization practices [20]. Decompiling optimized PTX into human-readable CUDA would significantly reduce the effort required for performance analysis and optimization tuning.

Furthermore, this approach enables effective human-compiler collaborative programming by generating readable CUDA blueprints from compiler-optimized PTX, thereby fa-

[*]Corresponding author.

cilitating further refinement by domain experts.

Finally, it contributes to the advancement of LLM-based code generation by producing high-quality, optimization-preserving parallel code datasets that capture sophisticated compiler transformations.

LLMs have demonstrated significant capabilities in various code-related tasks in recent years (e.g., code completion, bug fixing, and program translation). We investigate whether LLMs can serve as automated PTX-to-CUDA decompilers to address the gap in GPU decompilation tools. Unfortunately, our experiments show that existing LLMs cannot reliably perform PTX-to-CUDA tensor program decompilation, often generating low-readability, non-compilable, or erroneous code. Existing LLMs struggle with decompilation primarily due to domain-specific knowledge gaps. Despite understanding CUDA, they cannot sufficiently interpret PTX semantics and reconstruct equivalent CUDA implementations. Accordingly, we explore methods to enhance LLMs for decompiling optimized PTX into readable CUDA code for tensor programs in this work.

The primary challenge is **data scarcity**. Large-scale datasets are essential to fine-tune LLMs with domain knowledge, yet no substantial public corpus exists for GPU decompilation research. Furthermore, high-performance decompilation must not only preserve program semantics (i.e., computational logic) but also accurately reflect how computations are executed in optimized code (i.e., scheduling strategies). This necessitates exposing LLMs to diverse PTX implementations of identical computations under various optimizations, enabling them to faithfully reconstruct scheduling details in regenerated CUDA. Such requirements demand exceptional data diversity.

A second challenge stems from **PTX complexity**. As low-level code, PTX exhibits fragmented semantic information and poor readability. Compounding this issue, loop unrolling optimization—widely adopted to enhance loop efficiency—dramatically increases the volume of optimized PTX code. This verbosity further spreads semantic information thin, significantly impeding LLMs' comprehension of overall program logic.

To meet the demand for high-quality datasets in decompilation research, we propose a compiler-based data augmentation approach for dataset construction. Our method processes Python-defined DNN models through two distinct pipelines: (1) scheduling-diverse pipeline, and (2) subgraph-diverse pipeline. This dual-path approach ensures comprehensive data diversity across model architectures, subgraph patterns, and scheduling implementations.

Next, we perform post-processing on the compiler-generated raw CUDA programs to enhance data quality through two key steps: (1) CUDA kernel refactoring, and (2) similarity-based data filtering. This pipeline ultimately produces a high-quality dataset containing 400K CUDA-PTX kernel pairs, complete with the necessary configuration for execution testing.

To address the complexity of optimized PTX code, we introduce loop rerolling during data preprocessing. We propose Rolled-PTX, an intermediate representation that abstracts and condenses repeated, unrolled loops in PTX code. Our approach employs a pattern-matching heuristic algorithm to automatically identify loop patterns in original PTX code and perform loop rerolling transformations, converting the code into our Rolled-PTX representation. Our method effectively alleviates the challenge of feature sparsity in enabling LLMs to comprehend complete program semantics.

Finally, leveraging our proposed dataset and data preprocessing methodology, we perform supervised fine-tuning of base models, including Qwen2.5-Coder-7B[21] and Qwen3-32B[22], building **PtxDec**—a decompilation prototype for PTX-to-CUDA translation.

Comparative experiments with multiple baseline models demonstrate that PtxDec achieves 2.3×-3.1× improvement in functional accuracy for PTX-to-CUDA decompilation, while significantly outperforming baselines in both code readability and scheduling consistency metrics. Ablation studies further validate that: (1) our compiler-based data augmentation forms the essential foundation for effective decompilation research, (2) the CUDA kernel refactoring method substantially enhances dataset quality, and (3) the PTX loop rerolling technique dramatically improves LLM's semantic understanding capability.

In summary, we make the following novel contributions:

- We establish the critical need for PTX-to-CUDA decompilation – an emerging requirement driven by GPU high-performance computing demands.
- We propose a compiler-based data augmentation framework with post-processing techniques, creating a scalable dataset infrastructure that enables future research in GPU decompilation.
- We introduce Rolled-PTX, an intermediate representation employing heuristic loop rerolling to capture unrolled PTX patterns, demonstrating the continued relevance of compiler techniques in LLM-based decompilation.
- Our experiments demonstrate substantially enhanced LLM decompilation capability with significant improvements in functional accuracy, code readability, and scheduling consistency – establishing the first effective exploration of PTX-to-CUDA decompilation.

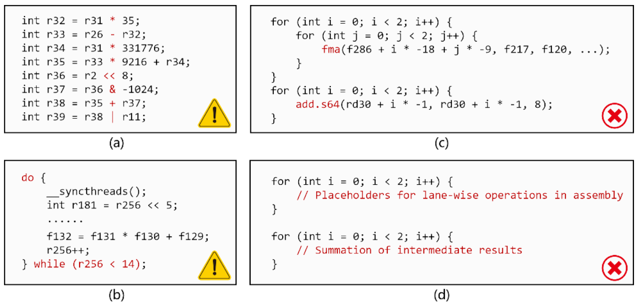
II. Background

In this section, we present examples illustrating limitations of existing LLMs in PTX-to-CUDA decompilation, establishing motivation for our work. We then analyze core challenges in enhancing LLMs for optimized PTX decompilation.

A. Motivating Example

To evaluate existing LLMs as automated PTX-to-CUDA decompilers, we tested three representative models (GPT-4o[23], DeepSeek-V3[20] and Qwen2.5-Coder-7B[21]) on real PTX kernels from deep learning compilers. Our experiments reveal significant limitations in current LLMs.

1) Line-by-line translation: LLMs tend to translate PTX instructions individually rather than reconstructing high-level expressions. As low-level code, each PTX statement contains



```
int r32 = r31 * 35;
int r33 = r26 - r32;
int r34 = r31 * 331776;
int r35 = r33 * 9216 + r34;
int r36 = r2 << 8;
int r37 = r36 & -1024;
int r38 = r35 + r37;
int r39 = r38 | r11;
```
(a)

```
do {
    __syncthreads();
    int r181 = r256 << 5;
    ......
    f132 = f131 * f130 + f129;
    r256++;
} while (r256 < 14);
```
(b)

```
for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 2; j++) {
        fma(f286 + i * -18 + j * -9, f217, f120, ...);
    }
}
for (int i = 0; i < 2; i++) {
    add.s64(rd30 + i * -1, rd30 + i * -1, 8);
}
```
(c)

```
for (int i = 0; i < 2; i++) {
    // Placeholders for lane-wise operations in assembly
}

for (int i = 0; i < 2; i++) {
    // Summation of intermediate results
}
```
(d)

Fig. 2: Examples of common limitations when decompiling PTX to CUDA by current LLMs directly.

a single opcode (e.g., add, multiply) with virtual registers/immediate operands. As shown in Figure 2(a), this produces fragmented CUDA code where opcodes become operators and registers become variables – failing to reassemble meaningful computational expressions.

2) Unnatural control flow: While GPU programmers typically use `for` loops, LLMs frequently generate `do-while` constructs (Figure 2(b)). This stems from PTX's branch-based loop implementation resembling `do-while` semantics, creating readability gaps versus human coding conventions.

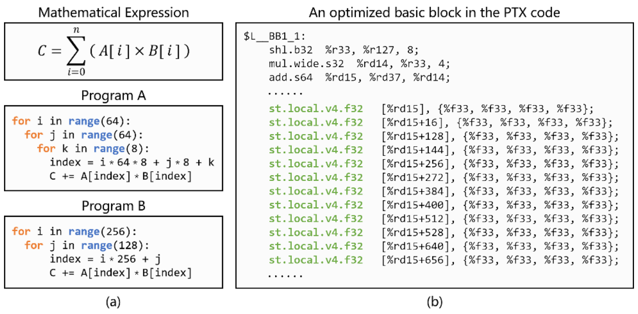
3) Low-level instruction misuse: LLMs directly transplant PTX-specific instructions into CUDA, such as using `add.s64` outside standard CUDA conventions (Figure 2(c)). This indicates confusion between PTX semantics and CUDA idioms.

4) Incomplete code generation: For complex segments, LLMs sometimes skip implementation details and insert placeholder comments instead of valid code (Figure 2(d)), compromising functional correctness.

These limitations fundamentally stem from LLMs' lack of domain-specific knowledge. While they may recognize CUDA patterns, they cannot adequately interpret low-level PTX semantics or reconstruct equivalent high-level implementations.

B. Challenges

We identify two fundamental challenges in enhancing LLMs for PTX-to-CUDA decompilation.



$$C = \sum_{i=0}^{n} (A[i] \times B[i])$$

```
for i in range(64):
  for j in range(64):
    for k in range(8):
      index = i * 64 * 8 + j * 8 + k
      C += A[index] * B[index]
```

```
for i in range(256):
  for j in range(128):
      index = i * 256 + j
      C += A[index] * B[index]
```
(a)

```
$L__BB1_1:
    shl.b32  %r33, %r127, 8;
    mul.wide.s32  %rd14, %r33, 4;
    add.s64  %rd15, %rd37, %rd14;
    ......
    st.local.v4.f32   [%rd15], {%f33, %f33, %f33, %f33};
    st.local.v4.f32   [%rd15+16], {%f33, %f33, %f33, %f33};
    st.local.v4.f32   [%rd15+128], {%f33, %f33, %f33, %f33};
    st.local.v4.f32   [%rd15+144], {%f33, %f33, %f33, %f33};
    st.local.v4.f32   [%rd15+256], {%f33, %f33, %f33, %f33};
    st.local.v4.f32   [%rd15+272], {%f33, %f33, %f33, %f33};
    st.local.v4.f32   [%rd15+384], {%f33, %f33, %f33, %f33};
    st.local.v4.f32   [%rd15+400], {%f33, %f33, %f33, %f33};
    st.local.v4.f32   [%rd15+512], {%f33, %f33, %f33, %f33};
    st.local.v4.f32   [%rd15+528], {%f33, %f33, %f33, %f33};
    st.local.v4.f32   [%rd15+640], {%f33, %f33, %f33, %f33};
    st.local.v4.f32   [%rd15+656], {%f33, %f33, %f33, %f33};
    ......
```
(b)

Fig. 3: (a) Example of different scheduling strategies. (b) Example of a basic block with loop unrolling optimization.

1) Data Scarcity: The absence of dedicated datasets for GPU decompilation poses a critical barrier. While general code repositories like The-Stack [24] contain CUDA kernels, they lack compilation configurations and runtime environments. Extracting pure CUDA kernels from libraries like CUTLASS [25] is impractical due to heavy templating. Domain-specific datasets are inadequate: Tenset [26] focuses on performance metrics rather than code translation, and LS-CAT's linear algebra kernels [27] remain inaccessible due to closed-source limitations. This contrasts sharply with CPU decompilation, where abundant resources like Exebench [28] and AnghaBench [29] accelerate research.

Critically, performance-oriented decompilation necessitates dual fidelity: preserving computational semantics while precisely reconstructing scheduling strategies. As shown in Figure Figure 3(a), Program A and Program B have identical inputs/outputs and mathematical logic but run at different speeds due to scheduling choices. For performance tuning, developers need to see the actual scheduling details used in the optimized code – confusing different schedules (like mistaking Program B for Program A) would mislead optimization efforts.

Consequently, effective LLM training requires diverse examples of how identical computations manifest differently in PTX under various optimizations, demanding exceptionally varied data to teach accurate scheduling reconstruction. This drives our work to automate building datasets meeting these strict requirements.

2) PTX Complexity: PTX [19], a low-level static single assignment (SSA) intermediate representation, exhibits fragmented semantics and poor readability. At this level, high-level loop structures are decomposed into sequentially executed basic blocks connected by explicit branch instructions, further obscuring program semantics.

Tensor programs typically contain numerous loops, where loop unrolling is widely adopted for performance gains. This optimization replicates loop bodies into straight-line code to reduce control overhead [30, 31], but dramatically expands PTX volume. As shown in Figure 3(b), unrolled PTX becomes highly verbose, diluting semantic coherence and significantly impeding LLMs' ability to comprehend program logic. Moreover, excessive code length may exceed LLMs' input limits, while longer sequences increase token costs.

Loop rerolling counteracts this by restoring unrolled code to compact loop forms. Existing rerolling techniques target either x86 assembly [32] or LLVM IR [31], leaving PTX unsupported. This motivates our novel PTX rerolling technique – deployed as critical preprocessing for decompilation [33] – to address PTX complexity.

III. Approach

In this section, we first outline the overall workflow of our approach, then introduce the details of the core method we proposed.

A. Overview

Figure 4 illustrates our three-phase framework for enhancing LLM-based PTX-to-CUDA decompilation of tensor programs:

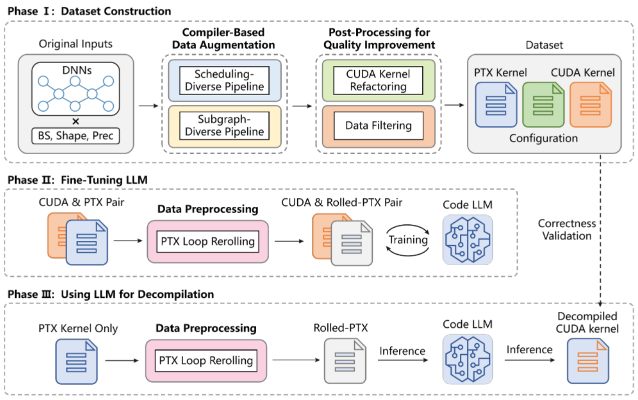


Fig. 4: The overall workflow of our approach.

Dataset Construction, Fine-Tuning LLM, and Using LLM for Decompilation.

1) Dataset Construction: In the first phase, we address the demand for a high-quality program corpus in decompilation. We source authentic DNN models from open-source communities as origin inputs, spanning computer vision, NLP, and recommendation systems. For each model instantiation, we systematically vary batch sizes, input shapes, and precision formats (FP32/FP16) to ensure comprehensive workload coverage and model-level diversity.

Each instantiated DNN model undergoes **Compiler-Based Data Augmentation**, where we first extract subgraphs from the computational graph, then generate multiple tensor program variants per subgraph at the operator scheduling level, each compiled into CUDA kernels. We employ two complementary search-based compilers specializing in subgraph and scheduling diversity, respectively, as detailed in section III-B.

The raw CUDA programs generated by the compiler undergo **Post-Processing** to enhance code readability through CUDA kernel refactoring. We further perform similarity-based data filtering to ensure sufficient sample diversity and improve overall data quality, as detailed in section III-C.

The post-processed CUDA code is compiled into PTX, from which we extract CUDA-PTX kernel pairs at the function level for dataset storage. Each kernel's configuration parameters - including input shapes, grid/block sizes - are preserved to enable test suite reconstruction.

2) Fine-Tuning LLM: In the second phase, the CUDA-PTX kernel pairs from the generated dataset are used to preprocess PTX code, followed by supervised fine-tuning of the LLM.

The **Data Preprocessing** module addresses the complexity of highly optimized PTX code through pattern-matching heuristics that automatically identify and reroll loops into our custom Rolled-PTX intermediate representation. Rolled-PTX abstracts unrolled loop patterns for improved readability while preserving semantics, as detailed in section III-D.

The preprocessed Rolled-PTX serves as input to the code LLM, paired with the corresponding CUDA code as training labels. Through supervised fine-tuning, the LLM learns kernel-level PTX-CUDA decompilation capabilities.

3) Using LLM for Decompilation: In the final phase, the fine-tuned LLM performs PTX-to-CUDA decompilation. User-provided PTX code undergoes identical preprocessing as in Phase 2, where kernel-level Rolled-PTX representations are generated and fed to the LLM for inference, producing decompiled CUDA kernel code.

B. Compiler-Based Data Augmentation

Our data augmentation starts with a DNN model, generating diverse CUDA tensor programs. We leverage deep learning compilers' hierarchical structure:

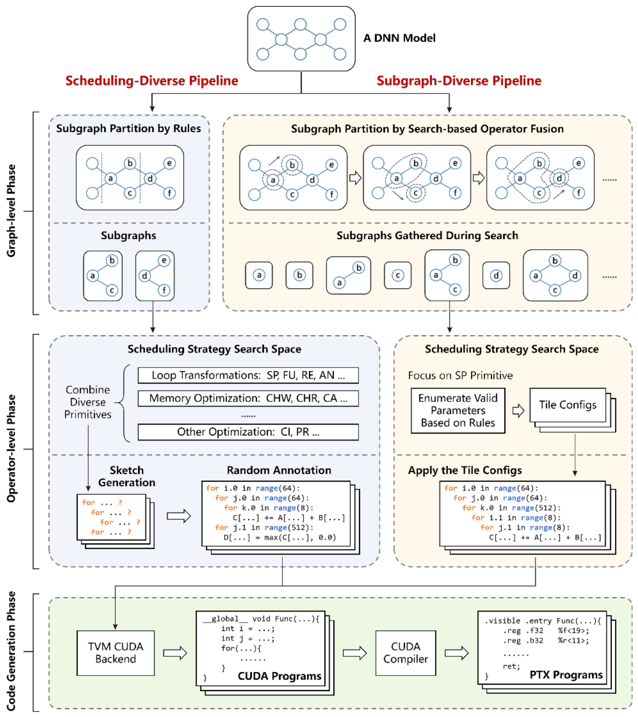
- **Graph-level**: Compilers partition DNN computational graphs into subgraphs.
- **Operator-level**: Each subgraph receives scheduling strategies for execution.

Different partitioning and scheduling approaches naturally create program variants from a single model. This inherent diversity makes compilers ideal data sources for large-scale, high-quality decompilation datasets.

As illustrated in Figure 5, our framework processes DNN models through dual complementary pipelines:

- **Scheduling-Diverse Pipeline**: Focuses on generating program variants with diverse computational schedules.
- **Subgraph-Diverse Pipeline**: Creates varied computational logic combinations through subgraph exploration.

These pipelines address distinct but complementary aspects of program diversity. The subgraph-level variations expose


A DNN Model
Scheduling-Diverse Pipeline
Subgraph-Diverse Pipeline
Graph-level Phase
Subgraph Partition by Rules
Subgraph Partition by Search-based Operator Fusion
a
b
c
d
e
f
......
Subgraphs
Subgraphs Gathered During Search
Operator-level Phase
Scheduling Strategy Search Space
Combine Diverse Primitives
Loop Transformations: SP, FU, RE, AN ...
Memory Optimization: CHW, CHR, CA ...
......
Other Optimization: CI, PR ...
Sketch Generation
Random Annotation
Scheduling Strategy Search Space
Focus on SP Primitive
Enumerate Valid Parameters Based on Rules
Tile Configs
Apply the Tile Configs
Code Generation Phase
TVM CUDA Backend
CUDA Programs
CUDA Compiler
PTX Programs

```
for ... ?
  for ... ?
    for ... ?
  for ... ?
```

```
for i.0 in range(64):
  for j.0 in range(64):
    for k.0 in range(8):
      C[...] += A[...] + B[...]
  for j.1 in range(512):
    D[...] = max(C[...], 0.0)
```

```
for i.0 in range(64):
  for j.0 in range(64):
    for k.0 in range(512):
      for i.1 in range(8):
        for j.1 in range(8):
          C[...] += A[...] + B[...]
```

```
__global__ void Func(...){
    int i = ...;
    int j = ...;
    for(...){
        ......
    }
}
```

```
.visible .entry Func(...){
    .reg .f32   %f<19>;
    .reg .b32   %r<11>;
    ......
    ret;
}
```

Fig. 5: Compiler-based data augmentation framework.

LLMs to different computational logic patterns, while the scheduling-level variations enable learning subtle implementation differences across optimization strategies.

1) Scheduling-Diverse Pipeline: Our Scheduling-Diverse Pipeline begins by partitioning the input DNN's computational graph using TVM's rule-based method. This approach groups operators into predefined categories and applies expert rules to determine segmentation points. For example, processing ResNet50 [34] yields 39 distinct subgraphs. Though this deterministic approach produces fewer subgraphs than search-based methods, it captures patterns commonly seen in real-world applications.

At the operator level, we generate program variants by sampling strategies from our scheduling strategy search space. This space incorporates fundamental transformation primitives, such as loop splitting (SP) for tiling computations, fusion (FU) to combine parallel loops, reordering (RE) to optimize loop nest structures, and annotations (AN) to specify loop properties. These primitives with adjustable parameters can be combined to create complex optimization strategies. To efficiently explore this large space, we first create architectural sketches that define high-level structures like loop frameworks, then apply random annotation to fill concrete parameters such as loop iteration counts.

This method balances flexible exploration of program architectures with efficient sampling of low-level details. For each subgraph, we systematically generate thousands of unique program variants, ensuring comprehensive diversity in scheduling implementations.

2) Subgraph-Diverse Pipeline: Our Subgraph-Diverse Pipeline first partitions the DNN's computational graph using search-based operator fusion. This method systematically explores the graph as a search space, progressively adding operators to existing fusion groups or creating new groups until covering the entire graph. We collect all unique fusion groups as candidate subgraphs during this process. Unlike rule-based approaches limited by expert knowledge, this search-driven method discovers a wider range of subgraph patterns—from single-operator primitives to complex multi-operator compositions—ensuring comprehensive subgraph diversity.

At the operator level, we similarly generate program variants by sampling scheduling strategies. Given the large number of subgraphs produced, we focus on the critical Split (SP) primitive to avoid a combinatorial explosion. This primitive implements computation tiling and significantly impacts program structure. Using Roller's heuristic approach [35], we enumerate valid parameters for splitting operations to generate multiple tile configurations. Applying these configurations to subgraphs typically yields dozens to hundreds of distinct program variants per subgraph.

We implement the Scheduling-Diverse Pipeline using the Ansor [36] compiler and the Subgraph-Diverse Pipeline via Welder [37]. In the final code generation phase, all program variants from both pipelines are compiled to CUDA using TVM's backend, then transformed into PTX via the CUDA compiler. We retain default optimization settings throughout compilation since these compiler-prescribed configurations already target high performance.

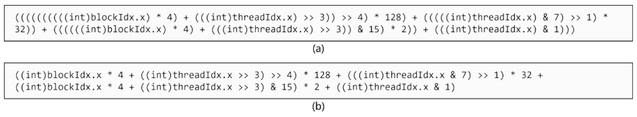
C. Post-Processing for Quality Improvement

While the previous section described generating large-scale CUDA tensor programs through compiler-based augmentation, this section details our post-processing approach using kernel refactoring and data filtering to significantly improve dataset quality.

1) Kernel Refactoring: Deep learning compilers prioritize correctness and performance over readability when generating CUDA code for downstream compilation. However, our decompilation task requires human-readable, maintainable outputs. This mismatch necessitates refactoring raw compiler-generated code. We address key readability issues through two targeted techniques.

The first technique focuses on **redundant parenthesis elimination**. Compiler-generated CUDA often contains expressions with excessive parentheses for correctness assurance, creating unreadable nested structures as shown in Figure 6(a). These impair both human comprehension and LLM learning, increasing syntax error rates. To address this, we developed an algorithm that systematically parses expressions from innermost to outermost layers while comparing operator precedence across nesting levels. This process removes only semantically redundant parentheses, significantly simplifying expressions as demonstrated in Figure 6(b) while preserving mathematical equivalence. The resulting cleaner syntax enhances both code readability and model learnability.

The second technique performs **common subexpression extraction(CSE)** for array indexing. Compiler-generated CUDA frequently contains array index expressions with significant computational redundancy, as shown in Figure 7(a), where repeated subexpressions create verbose and unreadable code.



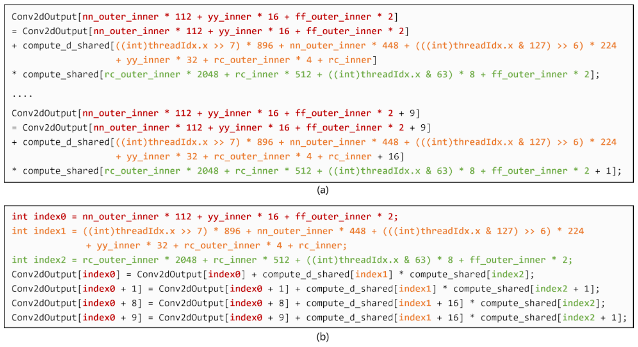
```
(((((((((int)blockIdx.x) * 4) + (((int)threadIdx.x) >> 3)) >> 4) * 128) + (((((int)threadIdx.x) & 7) >> 1) *
32)) + ((((((int)blockIdx.x) * 4) + (((int)threadIdx.x) >> 3)) & 15) * 2)) + (((int)threadIdx.x) & 1)))
```

(a)

```
((int)blockIdx.x * 4 + ((int)threadIdx.x >> 3) >> 4) * 128 + (((int)threadIdx.x & 7) >> 1) * 32 +
((int)blockIdx.x * 4 + ((int)threadIdx.x >> 3) & 15) * 2 + ((int)threadIdx.x & 1)
```

(b)

Fig. 6: Example of redundant parenthesis elimination.



```
Conv2dOutput[nn_outer_inner * 112 + yy_inner * 16 + ff_outer_inner * 2]
= Conv2dOutput[nn_outer_inner * 112 + yy_inner * 16 + ff_outer_inner * 2]
+ compute_d_shared[((int)threadIdx.x >> 7) * 896 + nn_outer_inner * 448 + (((int)threadIdx.x & 127) >> 6) * 224
                   + yy_inner * 32 + rc_outer_inner * 4 + rc_inner]
* compute_shared[rc_outer_inner * 2048 + rc_inner * 512 + ((int)threadIdx.x & 63) * 8 + ff_outer_inner * 2];

....

Conv2dOutput[nn_outer_inner * 112 + yy_inner * 16 + ff_outer_inner * 2 + 9]
= Conv2dOutput[nn_outer_inner * 112 + yy_inner * 16 + ff_outer_inner * 2 + 9]
+ compute_d_shared[((int)threadIdx.x >> 7) * 896 + nn_outer_inner * 448 + (((int)threadIdx.x & 127) >> 6) * 224
                   + yy_inner * 32 + rc_outer_inner * 4 + rc_inner + 16]
* compute_shared[rc_outer_inner * 2048 + rc_inner * 512 + ((int)threadIdx.x & 63) * 8 + ff_outer_inner * 2 + 1];
```

(a)

```
int index0 = nn_outer_inner * 112 + yy_inner * 16 + ff_outer_inner * 2;
int index1 = ((int)threadIdx.x >> 7) * 896 + nn_outer_inner * 448 + (((int)threadIdx.x & 127) >> 6) * 224
             + yy_inner * 32 + rc_outer_inner * 4 + rc_inner;
int index2 = rc_outer_inner * 2048 + rc_inner * 512 + ((int)threadIdx.x & 63) * 8 + ff_outer_inner * 2;
Conv2dOutput[index0] = Conv2dOutput[index0] + compute_d_shared[index1] * compute_shared[index2];
Conv2dOutput[index0 + 1] = Conv2dOutput[index0 + 1] + compute_d_shared[index1] * compute_shared[index2 + 1];
Conv2dOutput[index0 + 8] = Conv2dOutput[index0 + 8] + compute_d_shared[index1 + 16] * compute_shared[index2];
Conv2dOutput[index0 + 9] = Conv2dOutput[index0 + 9] + compute_d_shared[index1 + 16] * compute_shared[index2 + 1];
```

(b)

Fig. 7: Example of common subexpression extraction for array indexing expressions.

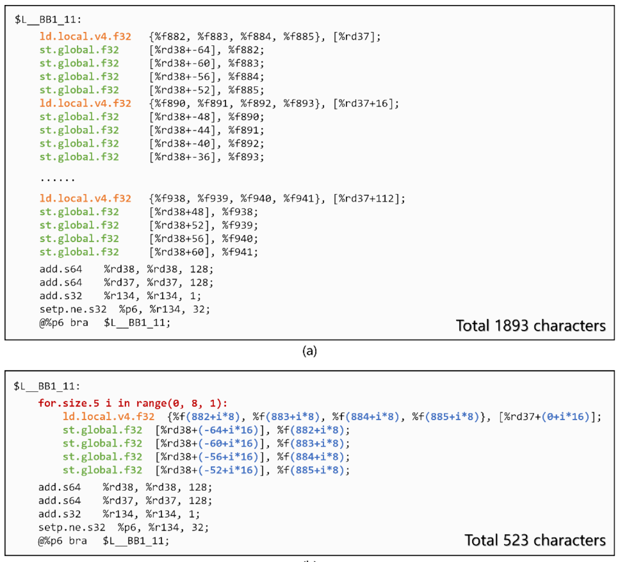
Although downstream compilers perform CSE during low-level compilation, these optimizations occur after CUDA code generation and thus cannot simplify the source code directly.

To address this, we developed a heuristic algorithm performing source-level CSE on CUDA code. Our approach traverses index expressions top-down, creating variables for unique subexpressions and replacing recurring patterns. For Index+Offset forms, we preserve the Offset while substituting redundant Index components. Variables are defined before first use and redefined at scope boundaries, transforming complex expressions into concise variable-based forms (Figure 7(b)) that enhance readability and LLM learnability.

2) Data Filtering: We further enhance dataset quality through approximate deduplication to ensure sufficient sample diversity, as studies [38, 39] demonstrate that such deduplication significantly improves language model performance on code tasks. Following established methods, we compute MinHash [40] signatures for all CUDA programs and apply Locality-Sensitive Hashing for efficient similarity detection. Samples exceeding a preset threshold are removed, effectively reducing redundancy while preserving key variations across our dataset.

This section introduces our preprocessing technique that transforms PTX code into an LLM-friendly format to enhance decompilation accuracy and efficiency. We achieve this through loop rerolling transformations implemented via Rolled-PTX - an intermediate representation designed to abstract unrolled loop patterns.

As shown in Figure 8(b), our pattern-matching heuristic automatically identifies loop structures in raw PTX (shown in Figure 8(a)) and condenses them into Rolled-PTX. This semantic-preserving transformation significantly reduces code volume while retaining program logic. The following subsections formalize Rolled-PTX and detail our rerolling algorithm.



```
$L__BB1_11:
    ld.local.v4.f32   {%f882, %f883, %f884, %f885}, [%rd37];
    st.global.f32     [%rd38+-64], %f882;
    st.global.f32     [%rd38+-60], %f883;
    st.global.f32     [%rd38+-56], %f884;
    st.global.f32     [%rd38+-52], %f885;
    ld.local.v4.f32   {%f890, %f891, %f892, %f893}, [%rd37+16];
    st.global.f32     [%rd38+-48], %f890;
    st.global.f32     [%rd38+-44], %f891;
    st.global.f32     [%rd38+-40], %f892;
    st.global.f32     [%rd38+-36], %f893;

    ......

    ld.local.v4.f32   {%f938, %f939, %f940, %f941}, [%rd37+112];
    st.global.f32     [%rd38+48], %f938;
    st.global.f32     [%rd38+52], %f939;
    st.global.f32     [%rd38+56], %f940;
    st.global.f32     [%rd38+60], %f941;
    add.s64    %rd38, %rd38, 128;
    add.s64    %rd37, %rd37, 128;
    add.s32    %r134, %r134, 1;
    setp.ne.s32  %p6, %r134, 32;
    @%p6 bra   $L__BB1_11;
```

Total 1893 characters

(a)

```
$L__BB1_11:
    for.size.5 i in range(0, 8, 1):
        ld.local.v4.f32  {%f(882+i*8), %f(883+i*8), %f(884+i*8), %f(885+i*8)}, [%rd37+(0+i*16)];
        st.global.f32  [%rd38+(-64+i*16)], %f(882+i*8);
        st.global.f32  [%rd38+(-60+i*16)], %f(883+i*8);
        st.global.f32  [%rd38+(-56+i*16)], %f(884+i*8);
        st.global.f32  [%rd38+(-52+i*16)], %f(885+i*8);
    add.s64    %rd38, %rd38, 128;
    add.s64    %rd37, %rd37, 128;
    add.s32    %r134, %r134, 1;
    setp.ne.s32  %p6, %r134, 32;
    @%p6 bra   $L__BB1_11;
```

Total 523 characters

(b)

Fig. 8: Example of Raw PTX and Rolled-PTX

D. Data Preprocessing

1) Formal Definition of Rolled-PTX: At the PTX level, high-level loop constructs are replaced by sequentially executed basic blocks connected through explicit branch instructions (e.g., `bra`). Unrolled loops become straight-line code within single basic blocks. To concisely represent these unrolled loops in a high-level-like form, we extend standard PTX with Rolled-PTX as shown in Figure 8.

$$
\begin{array}{rcl}
\langle PTX\_Statement\rangle & ::= & [\langle Label\rangle \texttt{" : "}]\ [\texttt{"@"}\ \texttt{"\%p"}\ [0-9]+] \\
 & & \langle Opcode\rangle\ \{\texttt{"."}\langle\ Modifier\rangle\}\ \langle OperandList\rangle \\
 & & [\texttt{"; "} \mid \langle Comment\rangle] \\
\langle Opcode\rangle & ::= & \langle MemoryOp\rangle \mid \langle ArithOp\rangle \mid \langle ControlOp\rangle \\
 & & \mid \langle SyncOp\rangle \mid \langle SpecialOp\rangle \mid \langle WmmaOp\rangle \\
 & & \mid \oplus\texttt{"}for\texttt{"} \\
\langle Modifier\rangle & ::= & \langle AddrSpace\rangle \mid \langle VecWidth\rangle \mid \langle DataType\rangle \\
 & & \mid \langle CacheHint\rangle \mid \langle AtomicOp\rangle \mid \ldots \\
 & & \mid \oplus\texttt{"}.size.\texttt{"}\ \langle Nat\rangle \\
\langle OperandList\rangle & ::= & \langle StandardOperands\rangle \\
 & & \mid \oplus\langle LoopControl\rangle \\
\langle StandardOperands\rangle & ::= & \langle Opreand\rangle\ \{\texttt{", "}\ \langle Operand\rangle\} \\
\langle Operand\rangle & ::= & \langle Register\rangle \mid \langle AddrExpr\rangle \mid \langle VectorGroup\rangle \\
 & & \mid \langle Immediate\rangle \mid \langle SymbolRef\rangle \\
\oplus\langle LoopControl\rangle & ::= & \langle Var\rangle\ \texttt{"}in\texttt{"}\ \texttt{"}range\texttt{"} \\
 & & \texttt{"("}\ \langle Nat\rangle\ \texttt{", "}\ \langle Nat\rangle\ \texttt{", "}\ \langle Nat\rangle\ \texttt{")"}\ \texttt{" : "} \\
\langle Var\rangle & ::= & [a-z][a-z0-9\_]* \\
\langle Nat\rangle & ::= & [0-9]+
\end{array}
$$

$$
\begin{array}{rcl}
\oplus\langle VarExpr\rangle & ::= & \langle Nat\rangle\{\texttt{" + "}\ \langle\ Offset\rangle\} \\
\langle Offset\rangle & ::= & \langle Var\rangle\ \texttt{" * "}\ \langle\ Nat\rangle \\
\oplus\langle VarReg\rangle & ::= & \langle RegPrefix\rangle\ \texttt{"("}\ \langle VarExpr\rangle\ \texttt{")"} \\
\langle RegPrefix\rangle & ::= & \texttt{"\%r"} \mid \texttt{"\%f"} \mid \texttt{"\%rd"}
\end{array}
$$

$$
\begin{array}{rcl}
\langle Immediate\rangle & ::= & \langle DecInt\rangle \mid \langle HexInt\rangle \mid \langle FloatConst\rangle \\
 & & \mid \langle BitPattern\rangle \\
 & & \mid \oplus\langle VarExpr\rangle \\
\langle Register\rangle & ::= & \langle ScalarReg\rangle \mid \langle AddrReg\rangle \mid \langle PredReg\rangle \\
 & & \mid \langle SpecialReg\rangle \\
 & & \mid \oplus\langle VarReg\rangle \\
\langle AddrExpr\rangle & ::= & \texttt{"["}\ \langle BaseAddr\rangle\ \{\langle OffsetTerm\rangle\}\ \texttt{"]"} \\
\langle BaseAddr\rangle & ::= & \langle AddrReg\rangle \mid \langle Symbol\rangle \\
 & & \mid \oplus\langle VarReg\rangle \\
\langle OffsetTerm\rangle & ::= & ('+' \mid '-')\ (\langle Immediate\rangle \mid \langle ScaledIndex\rangle \\
 & & \mid \oplus\langle VarExpr\rangle) \\
\langle VectorGroup\rangle & ::= & \texttt{"\{"}\ \langle RegItem\rangle\ (\texttt{", "}\ \langle RegItem\rangle +\ \texttt{"\}"} \\
\langle RegItem\rangle & ::= & \langle ScalarReg\rangle \\
 & & \mid \oplus\langle VarReg\rangle
\end{array}
$$

Rolled-PTX extends the standard PTX instruction format to concisely represent unrolled loops while maintaining semantic equivalence. A conventional PTX statement consists of three core components: the ⟨Opcode⟩ field specifying the operation, a sequence of ⟨Modifier⟩ elements, and an ⟨OperandList⟩. Our extensions introduce three key additions to implement loop constructs. First, we augment the ⟨Opcode⟩ field with a new “for” opcode designating loop headers. Second, we extend the ⟨Modifier⟩ sequence with a loop size specifier (e.g., “.size.5”) indicating the number of subsequent statements comprising the loop body. Third, we introduce a specialized ⟨LoopControl⟩ operand in the ⟨OperandList⟩, adopting Python-inspired syntax

$\langle$Var$\rangle$(initial, limit, step) where $\langle$Var$\rangle$ denotes the iteration variable and the three natural numbers define loop parameters.

To enable loop body implementation, we define two novel operand types. The $\langle$VarExpr$\rangle$ expression combines natural numbers with Offset terms (computed as iteration variables multiplied by scalars), explicitly supporting multiple Offset terms for nested loop scenarios. The $\langle$VarReg$\rangle$ virtual register integrates register type prefixes with $\langle$VarExpr$\rangle$ expressions, allowing dynamic register naming tied to iteration variables.

These constructs systematically integrate into PTX's four fundamental operand types—immediate values $\langle$Immediate$\rangle$, registers $\langle$Register$\rangle$, address expressions $\langle$AddrExpr$\rangle$, and vector operation groups $\langle$VectorGroup$\rangle$—through direct insertion of $\langle$VarExpr$\rangle$ and $\langle$VarReg$\rangle$ elements. This comprehensive extension preserves PTX's basic block execution model while enabling compact loop representation.

2) PTX Loop Rerolling Algorithm: During data preprocessing, our heuristic algorithm applies pattern matching to automatically identify loop patterns in raw PTX code and perform loop rerolling transformations. This process converts standard PTX into Rolled-PTX—our specialized intermediate representation—as formalized in Algorithm 1.

Let $C = \{s_1, s_2, ..., s_n\}$ denote the basic block context containing n statements. The algorithm proceeds as follows:

Algorithm 1 PTX Loop Rerolling

```
Input: Context $C = \{s_1, s_2, ..., s_n\}$
Output: Context $C'$ in Rolled-PTX format
 1: $C' \leftarrow C$
 2: $S \leftarrow \text{first}(C')$
 3: while $S \neq \text{end}(C')$ do
 4:     $T \leftarrow \emptyset$
 5:     while $S \neq \text{end}(C')$ do
 6:         $S_{\text{match}} \leftarrow \text{MatchPattern}(S, C')$
 7:         if $S_{\text{match}} = \emptyset$ then
 8:             $T \leftarrow \text{CreateTemplate}(S, S_{\text{match}}, C')$
 9:             if $T = \emptyset$ then
10:                 break
11:             else
12:                 $S \leftarrow \text{Next}(S_{\text{match}}, C')$
13:                 continue
14:             end if
15:         else
16:             break
17:         end if
18:     end while
19:     if $T \neq \emptyset$ then
20:         $T_{\text{valid}} \leftarrow \text{ValidateTemplate}(T, C')$
21:         $C' \leftarrow \text{ApplyTemplate}(T_{\text{valid}}, C')$
22:         $S \leftarrow \text{Next}(T_{\text{valid}}, C')$
23:         continue
24:     end if
25:     $S \leftarrow \text{Next}(S, C')$
26: end while
27: return $C'$
```

The algorithm automatically identifies and rerolls unrolled loops in PTX code through systematic pattern matching during top-down statement traversal. The process begins with pattern detection: for each candidate statement S, we search subsequent statements for S_{match} sharing identical opcodes, modifiers, and operand categories—register matches require consistent prefixes, immediates must share formats, and other operands follow category-based equivalence.

Upon finding S_{match}, template construction commences by comparing subsequent statements of S and S_{match}. The distance between S and S_{match} defines the potential loop size, with all body statements requiring consistent patterns to form valid template T. We then validate T by determining how many consecutive statements match the template starting from S, establishing the actual iteration count and generating annotated template T_{Valid} with $\langle$VarExpr$\rangle$ markers for varying operands.

Finally, template application replaces matched statements with compact loop constructs from T_{Valid}, resuming processing at the first unmatched statement. The algorithm recursively handles nested loops by reapplying these steps to Rolled-PTX output, progressively folding multi-level unrolled structures into concise representations.

This transformation preserves semantic equivalence while significantly reducing input sequence lengths for LLMs. By condensing unrolled loops, we alleviate information sparsity challenges that hinder semantic comprehension and reduce computational costs during fine-tuning and inference. Notably, Rolled-PTX and its generation algorithm provide standalone value beyond decompilation research, serving as independent tools for developers analyzing PTX code.

IV. Evaluation

We comprehensively evaluate our approach for enhancing LLM-based PTX-to-CUDA decompilation of tensor programs, addressing the following research questions:

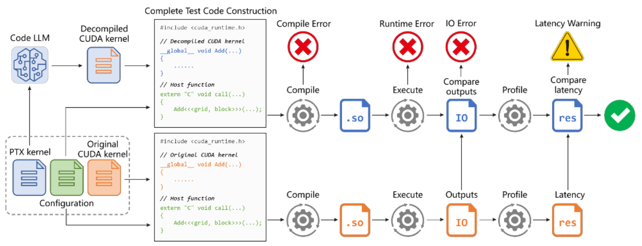
- **RQ1**: Functional accuracy and readability improvements
- **RQ2**: Optimization Scheduling consistency preservation
- **RQ3**: Impact of training data volume
- **RQ4**: Contribution of CUDA kernel refactoring
- **RQ5**: Effectiveness of PTX loop rerolling
- **RQ6**: Generalization of the decompilation approach

A. Experimental Setup

Our evaluation centers on **PtxDec**—a PTX-to-CUDA decompilation prototype implementing our full workflow on Qwen2.5-Coder-7B[21] and extended to Qwen3-32B [22]. We fine-tune the models using low-rank adaptation(LoRA) and supervised fine-tuning(SFT) via llama-factory [41] with batch size of 4 samples (8192 tokens each), an initial learning rate of 5×10^{-5}, and cosine decay scheduling. All experiments run on 8×NVIDIA H20 GPUs. During inference, we configure deterministic output (temperature=0, top_p=1.0) with a maximum token length of 4096, covering all CUDA samples in our dataset.

Benchmark. As no public benchmark exists for PTX-to-CUDA decompilation, we establish an evaluation benchmark by randomly sampling 4,000 test cases from our 400K dataset, providing a comprehensive foundation for analysis. The remaining dataset samples are reserved for model training.

Baselines. We evaluate PtxDec against four representative baselines: (1) Qwen2.5-Coder-7B [21]—our initial fine-tuning



```
#include <cuda_runtime.h>

// Decompiled CUDA kernel
__global__ void Add(...)
{
    ......
}
// Host function
extern "C" void call(...)
{
    Add<<<grid, block>>>(...);
}
```

```
#include <cuda_runtime.h>

// Original CUDA kernel
__global__ void Add(...)
{
    ......
}
// Host function
extern "C" void call(...)
{
    Add<<<grid, block>>>(...);
}
```

Fig. 9: Correctness verification workflow.

base model—to isolate performance improvements from our methodology; (2) GPT-4o[23] as the leading commercial LLM; (3) DeepSeek-V3 [20] as the representative open-source LLM; and (4) Qwen3-32B [22] as a stronger base model to assess scalability.

Metric. We employ three metric categories for evaluation:

- **Functional Metrics**: Compilation rate, runtime correctness rate, and I/O correctness rate assess semantic equivalence preservation. These established decompilation metrics validate whether the generated CUDA maintains the original computational logic.
- **Readability Metric**: CodeBLEU[42] score (equally weighted AST, DFG, keyword, and n-gram components) measures code structural similarity to human-written implementations, representing readability.
- **Scheduling Consistency**: The preservation of optimization schedules is quantified by performance and memory access deviations (in Latency and Gld Efficiency) between original and decompiled I/O-correct programs, visualized via kernel density estimation curves.

Correctness Verification. We validate decompilation correctness through a comprehensive testing pipeline, as shown in Figure 9. For each test case, the PTX kernel serves as input to the LLM, generating a decompiled CUDA kernel. Using stored configuration data, we construct complete test environments—including host functions and necessary headers—for both the decompiled kernel and the original reference CUDA kernel from our dataset. These implementations are compiled into Python-invocable shared libraries (.so).

Successful compilation initiates execution with identical inputs. We then verify functional equivalence by comparing outputs between the decompiled and original kernels. Specifically, we compute normalized absolute error (NAE) for each tensor pair across output lists, identifying maximum deviation values. If any tensor's maximum deviation exceeds our unified threshold of 0.001 (applicable to all tasks), we flag an I/O correctness failure. Concurrently, we profile execution latencies through multiple runs to obtain averaged measurements for assessing scheduling consistency.

B. RQ1: Functional accuracy and readability improvements

Table I demonstrates PtxDec's significant advantages across all functional metrics, achieving 2.3×–3.1× improvement over its 7B base model Qwen2.5-Coder-7B. Notably, PtxDec-32B further elevates performance, surpassing even the strongest baseline by over 60% absolute in functional correctness. Our progressive metrics reveal key insights.

TABLE I: Performance Comparison of PtxDec and Baseline Models on Functional and Readability Metrics

Tools	Functional Metrics			Readability Metric
	Compilation	Runtime Correctness	I/O Correctness	CodeBLEU Score
DeepSeek-v3	27.39%	17.85%	12.36%	0.2132
GPT-4o	31.35%	19.23%	13.23%	0.1761
Qwen2.5-Coder-7B	42.16%	34.61%	29.26%	0.1307
Qwen3-32B Think	50.69%	32.00%	24.44%	0.2395
PtxDec-7B	**97.24%**	**96.40%**	**90.89%**	**0.9664**
PtxDec-32B	**98.15%**	**97.24%**	**91.45%**	**0.9751**

PtxDec's superior compilation rate demonstrates significantly enhanced CUDA syntax/semantic understanding from fine-tuning on our dataset. This improvement stems primarily from kernel refactoring, which simplifies code structures while preserving semantics to boost model learning efficacy. In contrast, baseline models frequently generate syntactically invalid CUDA due to a lack of specialized training.

Runtime correctness rate evaluates semantic comprehension of PTX memory operations—critical in GPU programming, where explicit memory management prevents illegal accesses. Baseline models show significant accuracy drops from compilation to runtime, revealing PTX domain knowledge gaps. PtxDec overcomes this through domain-specific fine-tuning and PTX loop rerolling, which helps capture essential semantic patterns to maintain runtime correctness versus baselines' memory-related failures.

As the ultimate end-to-end metric, I/O correctness verifies identical output production under identical inputs—requiring precise computational logic translation through comprehensive PTX understanding. Our experiments show: (1) general-purpose LLMs struggle with this holistic requirement, while (2) code-optimized LLMs outperform general models due to architectural specialization. PtxDec advances this further via fine-tuning, achieving over 90% accuracy through enhanced PTX pattern recognition.

Meanwhile, PtxDec achieves a 3.5×–7.5× CodeBLEU improvement over all baseline models (shown in Table I), demonstrating superior code readability. This stems from overcoming baseline limitations where models produce fragmented outputs with unnatural control flows (shown in Figure 2). The specialized kernel refactoring ensures that models learn to generate highly readable CUDA code. Through fine-tuning on our refactored CUDA dataset, PtxDec learns comprehensive PTX-to-CUDA mapping relationships, generating human-readable code rather than low-level translations.

Answer to RQ1

While existing LLMs show limited PTX-to-CUDA decompilation capability, our approach significantly enhances functional correctness and code readability.

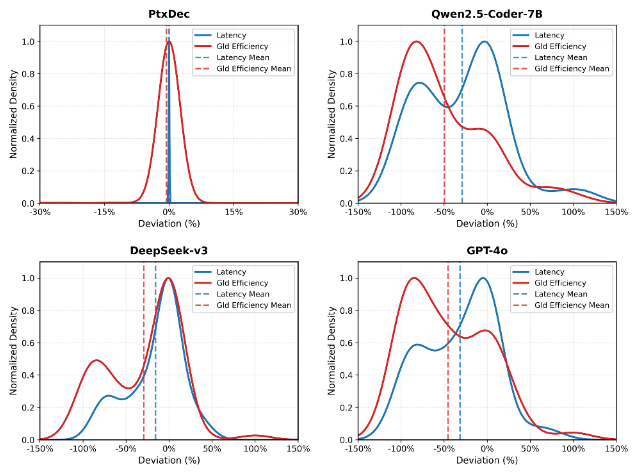


Fig. 10: Performance and memory access efficiency of PtxDec and baselines on scheduling consistency.

C. RQ2: Optimization Scheduling consistency preservation

Figure 10 illustrates the performance and memory access deviation distributions between decompiled and original CUDA programs, evaluated from both runtime latency and global load efficiency perspectives. Values near zero on the x-axis indicate better preservation of scheduling behavior. PtxDec-7B exhibits tightly clustered deviations around zero in both latency and Gld Efficiency, confirming minimal performance degradation and faithful scheduling preservation. In contrast, baseline models show widely dispersed distributions across both metrics. While they produce computationally correct CUDA code, their scheduling inaccuracies lead to significant performance variations and suboptimal memory access patterns.

This superiority stems from our dual-pipeline data augmentation strategy: the subgraph diversity component teaches robust computational logic variations, while the scheduling diversity component enables effective learning of optimization scheme implementations. Together, they facilitate precise translation of performance-critical details essential for maintaining both execution speed and memory efficiency.

Answer to RQ2

Our compiler-based data augmentation enhances scheduling diversity, significantly improving optimization scheduling consistency in LLM decompilation.

D. RQ3: Impact of training data volume

Figure 11 reveals distinctive learning trajectories during PtxDec-7B's fine-tuning, where compilation rate—reflecting basic CUDA syntax mastery—plateaus after 100k samples, while runtime correctness—demanding PTX semantic understanding—requires 300k samples to stabilize due to low-level code complexity. Most critically, I/O correctness measuring end-to-end accuracy continues improving even at 400k samples, demonstrating PTX-to-CUDA decompilation's exceptional dependence on large-scale, high-quality data.

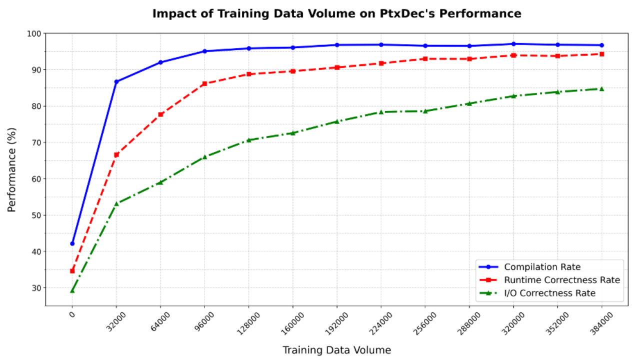


Fig. 11: Impact of training data volume on PtxDec's performance.

Answer to RQ3

Our automated dataset construction addresses LLM decompilation's substantial data demands, enhancing decompilation capability by generating a scaling dataset.

E. RQ4: Contribution of CUDA kernel refactoring

We quantify CUDA kernel refactoring's impact through rigorous ablation: recreating our training set without refactoring while keeping identical PTX samples and other experimental setup.

Table II presents the ablation study results. Fine-tuning the ablation model on this variant yields significantly degraded performance, with compilation rate dropping over 20% due to impaired CUDA syntax/semantic expression. Runtime and I/O correctness further deteriorate, confirming decompilation as a systematic process where foundational deficiencies cascade to final output quality.

TABLE II: Ablation Study of CUDA Kernel Refactoring on PtxDec's Performance.

	Compilation	Runtime Correctness	I/O Correctness
PtxDec-7B	97.24%	96.40%	90.89%
Ablation	75.07% ↓ 22.17%	70.15% ↓ 26.26%	59.49% ↓ 31.40%

Detailed compilation error analysis (Figure 12) reveals stark contrasts: PtxDec without Kernel Refactoring exhibits frequent syntax errors and incomplete code errors - both dramatically reduced in standard PtxDec.

Syntax errors primarily stem from complex computational expressions where excessive nested parentheses create unnecessary syntactic complexity. Our redundant parenthesis elimination technique addresses this by simplifying expressions while preserving semantics. Incomplete code errors occur when verbose outputs hit token limits, indicating poor code conciseness. Common subexpression extraction combats this

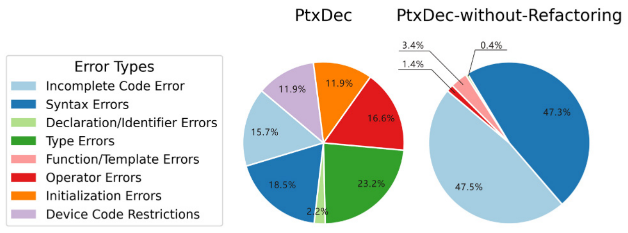


Fig. 12: Compilation error distributions between PtxDec and PtxDec-without-refactoring.

by eliminating redundancy - reducing average code length by 38% (Figure 13) and teaching compact coding styles. This dual refinement transforms training data into optimal learning material for decompilation tasks.

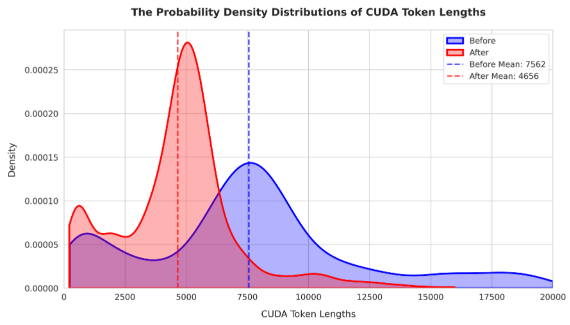


Fig. 13: The probability density distributions of CUDA token lengths.

Answer to RQ4

Essential for performance breakthroughs, CUDA kernel refactoring elevates training data quality to boost LLM decompilation.

F. RQ5: Effectiveness of PTX loop rerolling

TABLE III: Ablation Study of PTX Loop Rerolling on PtxDec's Performance.

	Compilation	Runtime Correctness	I/O Correctness
PtxDec-7B	97.24%	96.40%	90.89%
Ablation	89.05% ↓ 8.19%	72.17% ↓ 24.23%	52.57% ↓ 38.32%

PTX loop rerolling tackles the complexity of low-level code by converting unrolled loops into compact Rolled-PTX. When disabled in ablation tests (Table III), runtime correctness drops significantly and I/O correctness declines substantially. This decline directly shows that LLMs struggle to understand sparse patterns in lengthy, unrolled PTX code. The main challenge comes from sparse features: expanded loops force models to rebuild control flow from scattered instructions, making it hard to grasp the full program meaning.

Our pattern-based solution addresses this by condensing repeated structures. It reduces average PTX length by 41% (Figure 14) while keeping the same functionality. This compression brings three key benefits: (1) Better understanding of code patterns, (2) Ability to handle longer kernels within token limits, and (3) Faster processing due to shorter inputs.

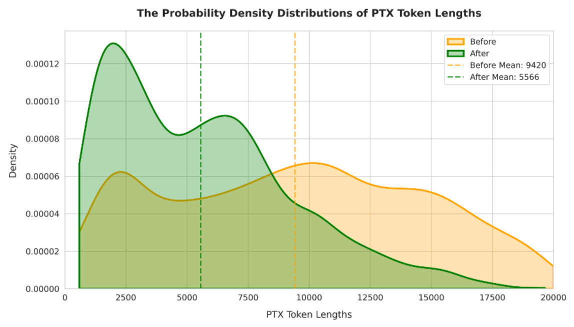


Fig. 14: The probability density distributions of PTX token lengths.

Answer to RQ5

PTX loop rolling transforms semantically fragmented PTX into LLM-friendly representations, essential for achieving both precise and efficient decompilation.

G. RQ6: Generalization of the Decompilation Approach

We evaluate the generalization capability of our method to out-of-domain PTX kernels, specifically hand-optimized codes and those generated by other compilers, through quantitative accuracy metrics and qualitative case studies.

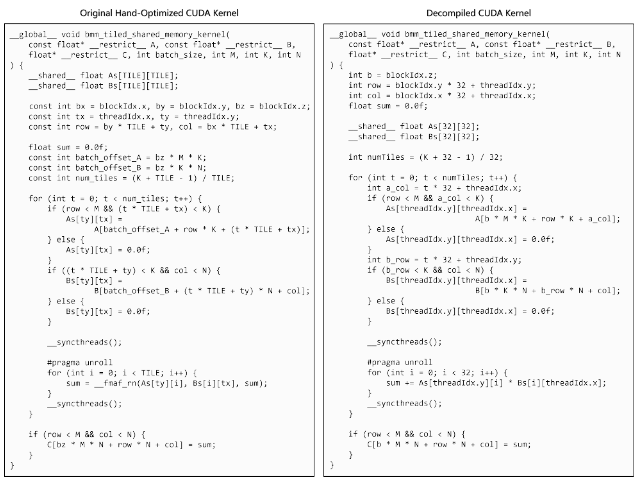
Quantitative Analysis on Accuracy Metrics. We perform zero-shot evaluation using the public kernelbench[43] benchmark, which comprises hundreds of DNN computation tasks defined in PyTorch. For each task, Jiaqi Lv et al.[44] provide a corresponding high-performance, hand-optimized CUDA kernel, offering a testbed for assessing generalization to complex, real-world code not encountered during training.

TABLE IV: Decompilation Performance of PtxDec and Baselines on Hand-Optimized Kernels.

	Compilation	Runtime Correctness	I/O Correctness
Qwen2.5-Coder-7B	28.35%	22.38%	6.71%
Qwen3-32B Think	49.25%	38.06%	15.67%
PtxDec-7B	**75.37%**	**62.68%**	**29.10%**
PtxDec-32B	**85.82%**	**81.34%**	**60.31%**

As shown in Table IV, our method demonstrates commendable generalization to these challenging, out-of-domain kernels. Both PtxDec variants maintain a substantial performance advantage over their base models across all functional metrics. The observed performance levels, though lower than those achieved on the in-distribution test set in our main experiments, remain significant. This decrease is expected given the distinct nature of hand-optimized kernels, yet the results confirm the robust transfer of PTX-to-CUDA knowledge learned through our methodology.

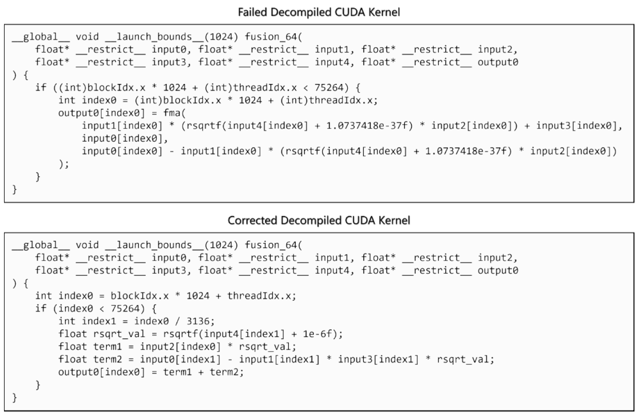
A key observation is the pronounced benefit of model scale in this generalization setting. The performance gap between PtxDec-32B and PtxDec-7B is notably larger here than in in-distribution tests, especially for the most demanding I/O correctness metric. This indicates that larger models more effectively generalize the complex semantic and structural patterns from our training data, yielding superior robustness when confronted with unfamiliar optimization patterns in hand-optimized code.



```
__global__ void bmm_tiled_shared_memory_kernel(
    const float* __restrict__ A, const float* __restrict__ B,
    float* __restrict__ C, int batch_size, int M, int K, int N
) {
    __shared__ float As[TILE][TILE];
    __shared__ float Bs[TILE][TILE];

    const int bx = blockIdx.x, by = blockIdx.y, bz = blockIdx.z;
    const int tx = threadIdx.x, ty = threadIdx.y;
    const int row = by * TILE + ty, col = bx * TILE + tx;

    float sum = 0.0f;
    const int batch_offset_A = bz * M * K;
    const int batch_offset_B = bz * K * N;
    const int num_tiles = (K + TILE - 1) / TILE;

    for (int t = 0; t < num_tiles; t++) {
        if (row < M && (t * TILE + tx) < K) {
            As[ty][tx] =
                  A[batch_offset_A + row * K + (t * TILE + tx)];
        } else {
            As[ty][tx] = 0.0f;
        }
        if ((t * TILE + ty) < K && col < N) {
            Bs[ty][tx] =
                  B[batch_offset_B + (t * TILE + ty) * N + col];
        } else {
            Bs[ty][tx] = 0.0f;
        }

        __syncthreads();

        #pragma unroll
        for (int i = 0; i < TILE; i++) {
            sum = __fmaf_rn(As[ty][i], Bs[i][tx], sum);
        }
        __syncthreads();
    }

    if (row < M && col < N) {
        C[bz * M * N + row * N + col] = sum;
    }
}
```

```
__global__ void bmm_tiled_shared_memory_kernel(
    const float* __restrict__ A, const float* __restrict__ B,
    float* __restrict__ C, int batch_size, int M, int K, int N
) {
    int b = blockIdx.z;
    int row = blockIdx.y * 32 + threadIdx.y;
    int col = blockIdx.x * 32 + threadIdx.x;
    float sum = 0.0f;

    __shared__ float As[32][32];
    __shared__ float Bs[32][32];

    int numTiles = (K + 32 - 1) / 32;

    for (int t = 0; t < numTiles; t++) {
        int a_col = t * 32 + threadIdx.x;
        if (row < M && a_col < K) {
            As[threadIdx.y][threadIdx.x] =
                                   A[b * M * K + row * K + a_col];
        } else {
            As[threadIdx.y][threadIdx.x] = 0.0f;
        }
        int b_row = t * 32 + threadIdx.y;
        if (b_row < K && col < N) {
            Bs[threadIdx.y][threadIdx.x] =
                                   B[b * K * N + b_row * N + col];
        } else {
            Bs[threadIdx.y][threadIdx.x] = 0.0f;
        }

        __syncthreads();

        #pragma unroll
        for (int i = 0; i < 32; i++) {
            sum += As[threadIdx.y][i] * Bs[i][threadIdx.x];
        }
        __syncthreads();
    }

    if (row < M && col < N) {
        C[b * M * N + row * N + col] = sum;
    }
}
```

Fig. 15: Successful reconstruction of hand-optimized CUDA kernel.


Failed Decompiled CUDA Kernel
Corrected Decompiled CUDA Kernel

```
__global__ void __launch_bounds__(1024) fusion_64(
    float* __restrict__ input0, float* __restrict__ input1, float* __restrict__ input2,
    float* __restrict__ input3, float* __restrict__ input4, float* __restrict__ output0
) {
    if ((int)blockIdx.x * 1024 + (int)threadIdx.x < 75264) {
        int index0 = (int)blockIdx.x * 1024 + (int)threadIdx.x;
        output0[index0] = fma(
            input1[index0] * (rsqrtf(input4[index0] + 1.0737418e-37f) * input2[index0]) + input3[index0],
            input0[index0],
            input0[index0] - input1[index0] * (rsqrtf(input4[index0] + 1.0737418e-37f) * input2[index0])
        );
    }
}
```

```
__global__ void __launch_bounds__(1024) fusion_64(
    float* __restrict__ input0, float* __restrict__ input1, float* __restrict__ input2,
    float* __restrict__ input3, float* __restrict__ input4, float* __restrict__ output0
) {
    int index0 = blockIdx.x * 1024 + threadIdx.x;
    if (index0 < 75264) {
        int index1 = index0 / 3136;
        float rsqrt_val = rsqrtf(input4[index1] + 1e-6f);
        float term1 = input2[index0] * rsqrt_val;
        float term2 = input0[index1] - input1[index1] * input3[index1] * rsqrt_val;
        output0[index0] = term1 + term2;
    }
}
```

Fig. 16: Error analysis and correction for XLA-Generated PTX kernel.

Qualitative Case Studies. We analyze a handwritten batched matrix multiplication kernel to demonstrate decompilation capabilities. As shown in Figure 15, the original kernel employs tiled computation with shared memory and the `TILE` macro for generality. The decompiled version successfully reconstructs key elements, including 3D grid mapping, tiled iterations, and boundary handling. While implementation differs in details such as the implementation approach of the `__fmaf_rn` instruction, the core logic remains consistent with improved readability through clear naming conventions.

Further analysis of an XLA-generated kernel (Figure 16) reveals both strengths and limitations. While correctly recovering the overall structure and thread organization, the decompilation exhibits several characteristic errors: (1) in index computation, the model fails to distinguish access patterns for different tensor shapes, applying uniform spatial indices where channel-specific indexing was required; (2) in expression reconstruction, it unnecessarily introduces `fma` calls absent in the original PTX, creating nested expressions from simple arithmetic sequences; and (3) in constant resolution, it misinterprets explicit values like `1e-6f`, generating implausible numerical constants. These issues highlight the central challenge of balancing precise low-to-high-level translation with code readability. Through targeted correction of these errors, we obtain a semantically equivalent and correct version.

Answer to RQ6

Despite certain challenges in precise low-level detail recovery, our approach demonstrates meaningful generalization to out-of-domain kernels while effectively preserving core algorithmic logic and maintaining code readability.

V. Conclusion

The rising computational demands of GPU-accelerated tensor programs necessitate effective tools for analyzing optimized PTX code.

This work establishes the first exploratory methodology for PTX-to-CUDA decompilation by enhancing LLM capabilities through two key innovations. Our compiler-based data augmentation framework overcomes critical data scarcity barriers, generating 400K high-quality CUDA-PTX kernel pairs through scheduling diversification and kernel refactoring. Complementing this, the Rolled-PTX intermediate representation addresses PTX complexity by heuristically condensing unrolled loops into comprehensible structures.

Experimental results demonstrate substantial improvements: our approach achieves 2.3×-3.1× higher functional accuracy than leading LLMs while significantly enhancing code readability and optimization scheduling consistency. These gains validate the synergistic value of compiler-inspired data engineering and semantic simplification techniques for low-level code understanding.

This research provides both a foundational decompilation pipeline and scalable dataset infrastructure, enabling future advancements in GPU code analysis and optimization. Our implementation and dataset are publicly available at https://github.com/S4Plus/PtxDec.

Acknowledgment

This work was funded in part by the National Natural Science Foundation of China (Grant No. 62272434) and in part by Kuaishou Technology.

References

[1] Hex-Rays, "Ida pro: The interactive disassembler." [Online]. Available: https://hex-rays.com/ida-pro

[2] N. S. A. (NSA), "Ghidra: Software reverse engineering framework (github repository)." [Online]. Available: https://github.com/NationalSecurityAgency/ghidra

[3] D. S. Katz, J. Ruchti, and E. Schulte, "Using recurrent neural networks for decompilation," in *2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2018, pp. 346–356.

[4] C. Fu, H. Chen, H. Liu, X. Chen, Y. Tian, F. Koushanfar, and J. Zhao, "Coda: An end-to-end neural program decompiler," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[5] R. Liang, Y. Cao, P. Hu, and K. Chen, "Neutron: an attention-based neural decompiler," *Cybersecurity*, vol. 4, pp. 1–13, 2021.

[6] I. Hosseini and B. Dolan-Gavitt, "Beyond the c: Retargetable decompilation using neural machine translation," *arXiv preprint arXiv:2212.08950*, 2022.

[7] Y. Cao, R. Liang, K. Chen, and P. Hu, "Boosting neural networks to decompile optimized binaries," in *proceedings of the 38th annual computer security applications conference*, 2022, pp. 508–518.

[8] A. Al-Kaswan, T. Ahmed, M. Izadi, A. A. Sawant, P. Devanbu, and A. van Deursen, "Extending source code pre-trained language models to summarise decompiled binaries," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2023, pp. 260–271.

[9] J. Armengol-Estapé, J. Woodruff, C. Cummins, and M. F. O'Boyle, "Slade: A portable small language model decompiler for optimized assembly," in *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2024, pp. 67–80.

[10] H. Tan, Q. Luo, J. Li, and Y. Zhang, "Llm4decompile: Decompiling binary code with large language models," *arXiv preprint arXiv:2403.05286*, 2024.

[11] X. She, Y. Zhao, and H. Wang, "Wadec: Decompiling webassembly using large language model," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 481–492.

[12] R. Wu, T. Kim, D. J. Tian, A. Bianchi, and D. Xu, "{DnD}: A {Cross-Architecture} deep neural network decompiler," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2135–2152.

[13] Z. Liu, Y. Yuan, S. Wang, X. Xie, and L. Ma, "Decompiling x86 deep neural network executables," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 7357–7374.

[14] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "{TVM}: An automated {End-to-End} optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.

[15] OpenXLA, "Xla: Optimizing compiler for machine learning." [Online]. Available: https://openxla.org/xla

[16] NVIDIA Corporation, "Triton Inference Server: An Optimized Cloud and Edge Inferencing Solution." [Online]. Available: https://github.com/triton-inference-server/server

[17] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, R. Jozefowicz, Y. Jia, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, M. Schuster, R. Monga, S. Moore, D. Murray, C. Olah, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow, Large-scale machine learning on heterogeneous systems," Nov. 2015.

[18] 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. Luk, B. Maher, Y. Pan, C. Puhrsch, M. Reso, M. Saroufim, M. Y. Siraichi, H. Suk, M. Suo, P. Tillet, E. Wang, X. Wang, W. Wen, S. Zhang, X. Zhao, K. Zhou, R. Zou, A. Mathews, G. Chanan, P. Wu, and S. Chintala, "PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation," in *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, Apr. 2024. [Online]. Available: https://pytorch.org/assets/pytorch2-2.pdf

[19] NVIDIA, "Nvidia parallel thread execution isa documentation." [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/

[20] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan *et al.*, "Deepseek-v3 technical report," *arXiv preprint arXiv:2412.19437*, 2024.

[21] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu *et al.*, "Qwen2. 5-coder technical report," *arXiv preprint arXiv:2409.12186*, 2024.

[22] A. Yang, A. Li, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Gao, C. Huang, C. Lv *et al.*, "Qwen3 technical report," *arXiv preprint arXiv:2505.09388*, 2025.

[23] A. Hurst, A. Lerer, A. P. Goucher, A. Perelman, A. Ramesh, A. Clark, A. Ostrow, A. Welihinda, A. Hayes, A. Radford *et al.*, "Gpt-4o system card," *arXiv preprint arXiv:2410.21276*, 2024.

[24] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, T. Liu, M. Tian, D. Kocetkov, A. Zucker, Y. Belkada, Z. Wang, Q. Liu, D. Abulkhanov, I. Paul, Z. Li, W.-D.

Li, M. Risdal, J. Li, J. Zhu, T. Y. Zhuo, E. Zheltonozhskii, N. O. O. Dade, W. Yu, L. Krauß, N. Jain, Y. Su, X. He, M. Dey, E. Abati, Y. Chai, N. Muennighoff, X. Tang, M. Oblokulov, C. Akiki, M. Marone, C. Mou, M. Mishra, A. Gu, B. Hui, T. Dao, A. Zebaze, O. Dehaene, N. Patry, C. Xu, J. McAuley, H. Hu, T. Scholak, S. Paquet, J. Robinson, C. J. Anderson, N. Chapados, M. Patwary, N. Tajbakhsh, Y. Jernite, C. M. Ferrandis, L. Zhang, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder 2 and the stack v2: The next generation," 2024.

[25] V. Thakkar, P. Ramani, C. Cecka, A. Shivam, H. Lu, E. Yan, J. Kosaian, M. Hoemmen, H. Wu, A. Kerr, M. Nicely, D. Merrill, D. Blasig, F. Qiao, P. Majcher, P. Springer, M. Hohnerbach, J. Wang, and M. Gupta, "CUTLASS," Jan. 2023. [Online]. Available: https://github.com/NVIDIA/cutlass

[26] L. Zheng, R. Liu, J. Shao, T. Chen, J. E. Gonzalez, I. Stoica, and A. H. Ali, "Tenset: A large-scale program performance dataset for learned tensor compilers," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[27] L. Bjertnes, J. O. Tørring, and A. C. Elster, "Ls-cat: a large-scale cuda autotuning dataset," in *2021 International Conference on Applied Artificial Intelligence (ICAPAI)*. IEEE, 2021, pp. 1–6.

[28] J. Armengol-Estapé, J. Woodruff, A. Brauckmann, J. W. d. S. Magalhães, and M. F. O'Boyle, "Exebench: an ml-scale dataset of executable c functions," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 50–59.

[29] A. F. Da Silva, B. C. Kind, J. W. de Souza Magalhães, J. N. Rocha, B. C. F. Guimaraes, and F. M. Q. Pereira, "Anghabench: A suite with one million compilable c benchmarks for code-size reduction," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 378–390.

[30] J.-C. Huang and T. Leng, "Generalized loop-unrolling: a method for program speedup," in *Proceedings 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology. ASSET'99 (Cat. No. PR00122)*. IEEE, 1999, pp. 244–248.

[31] R. C. Rocha, P. Petoumenos, B. Franke, P. Bhatotia, and M. O'Boyle, "Loop rolling for code size reduction," in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2022, pp. 217–229.

[32] T. Ge, Z. Mo, K. Wu, X. Zhang, and Y. Lu, "Rollbin: reducing code-size via loop rerolling at binary level," in *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, 2022, pp. 99–110.

[33] Z. D. Sisco, J. Balkind, T. Sherwood, and B. Hardekopf, "Loop rerolling for hardware decompilation," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 420–442, 2023.

[34] S. Targ, D. Almeida, and K. Lyman, "Resnet in resnet: Generalizing residual architectures," *arXiv preprint arXiv:1603.08029*, 2016.

[35] H. Zhu, R. Wu, Y. Diao, S. Ke, H. Li, C. Zhang, J. Xue, L. Ma, Y. Xia, W. Cui *et al.*, "{ROLLER}: Fast and efficient tensor compilation for deep learning," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 233–248.

[36] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen *et al.*, "Ansor: Generating {High-Performance} tensor programs for deep learning," in *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, 2020, pp. 863–879.

[37] Y. Shi, Z. Yang, J. Xue, L. Ma, Y. Xia, Z. Miao, Y. Guo, F. Yang, and L. Zhou, "Welder: Scheduling deep learning memory access via tile-graph," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 701–718.

[38] K. Lee, D. Ippolito, A. Nystrom, C. Zhang, D. Eck, C. Callison-Burch, and N. Carlini, "Deduplicating training data makes language models better," *arXiv preprint arXiv:2107.06499*, 2021.

[39] D. Kocetkov, R. Li, L. B. Allal, J. Li, C. Mou, C. M. Ferrandis, Y. Jernite, M. Mitchell, S. Hughes, T. Wolf *et al.*, "The stack: 3 tb of permissively licensed source code," *arXiv preprint arXiv:2211.15533*, 2022.

[40] A. Z. Broder, "Identifying and filtering near-duplicate documents," in *Annual symposium on combinatorial pattern matching*. Springer, 2000, pp. 1–10.

[41] Y. Zheng, R. Zhang, J. Zhang, Y. Ye, Z. Luo, Z. Feng, and Y. Ma, "Llamafactory: Unified efficient fine-tuning of 100+ language models," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*. Bangkok, Thailand: Association for Computational Linguistics, 2024. [Online]. Available: http://arxiv.org/abs/2403.13372

[42] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020.

[43] A. Ouyang, S. Guo, S. Arora, A. L. Zhang, W. Hu, C. Ré, and A. Mirhoseini, "Kernelbench: Can llms write efficient gpu kernels?" *arXiv preprint arXiv:2502.10517*, 2025.

[44] J. Lv, X. He, Y. Liu, X. Dai, A. Shen, Y. Li, J. Hao, J. Ding, Y. Hu, and S. Yin, "Hpctranscompile: An ai compiler generated dataset for high-performance cuda transpilation and llm preliminary exploration," *arXiv preprint arXiv:2506.10401*, 2025.