Buddy Compiler

An End-to-End AI Compiler from DSL to DSA

0 Buddy Compiler Overview

Homepage: https://buddy-compiler.github.io/

GitHub: https://github.com/buddy-compiler

Buddy Compiler is a "buddy system" to bridge the gap between AI models and hardware platforms. It consists of a frontend for AI framework and language integration, a middle-section leveraging MLIR for efficient optimization and tuning, and a backend for advanced code generation and hardware collaboration. By providing an end-to-end pipeline from domain-specific language (DSL) to domain-specific architecture (DSA), Buddy Compiler aims to establish a collaborative AI compiler ecosystem[1] to address fragmentation challenges in the current AI software stack.

1 Buddy Compiler Frontend

Buddy Compiler's frontend seamlessly connects with the DSL framework and PyTorch 2.x compilation framework via a graph infrastructure. The frontend integrates PyTorch AI models and computation kernels written in DSL into a unified graph representation. Moreover, the graph driver performs graph-level transformations and determines the compilation strategy, effectively converting the graph representation into MLIR.

Multimodal Input
Text / Audio / Image

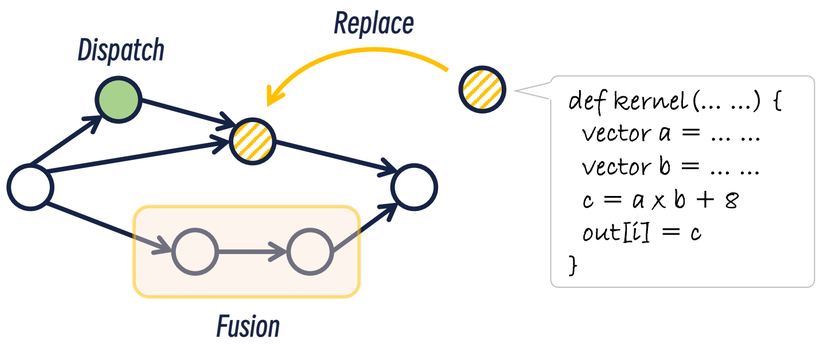
TorchDynamo Custom Compiler

Graph Capture → FX Graph → Aten IR

```
class GraphModule(torch.nn.Module):
 def forward(self, ...):
   ... = torch.ops.aten.arange.start(...)
   ... = torch.ops.aten.unsqueeze.default(...)
   ... = torch.ops.aten.view.default(...)
   ... ...
   return [... ...]
```

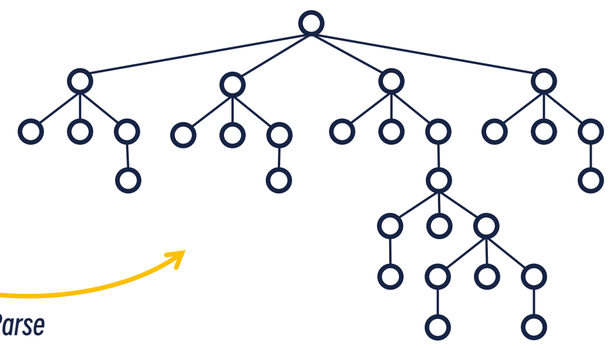
Graph Infrastructure

Graph Representation (Ops Registry) ← Graph Driver (Compilation Strategies)



DSL Framework

ANTLR (Parser Rules / Visitor) ← Standalone DSL (Syntax)



Core MLIR Dialect + Custom MLIR Dialect

2 Buddy Compiler Middle-Section

Buddy Compiler's middle-section focuses on performance optimization and domain-specific MLIR extensions (digital audio processing dialect and digital image processing dialect) to support efficient multimodal AI computation. It implements configurable rewrite patterns as compilation optimization strategies and builds the AutoConfig[2] mechanism for performance tuning. This approach ensures that optimizations are implemented once, delivering consistent impacts across different platforms.

AutoConfig Mechanism

Configurable Rewrite Patterns

```
void BroadcastPattern(Operation *op,
     ConversionPatternRewriter &rw, int64_t vs) {
  ... ...
  // Configure Vector Type.
  auto vectorTy = mlir::VectorType::get({vs}, f32);
  rw.create<vector::BroadcastOp>(..., vectorTy, ...);
  ... ...
}
```

```
void Img2ColPattern(Operation *op,
     ConversionPatternRewriter &rw, int64_t vs) {
  ... ...
  // Configure Vector Type.
  auto vectorTy = mlir::VectorType::get({vs}, f32);
  rw.create<affine::AffineVectorLoadOp> >(..., vectorTy, ...);
  ... ...
  rw.create<affine::AffineVectorStoreOp>(..., vectorTy, ...);
  ... ...
}
```

AutoConfig Driver

{ Rewrite Pattern - Analysis Model }

Algorithm

- Broadcast-based Convolution Optimization — Broadcast / FMA / Memory Ops
- Img2Col Convolution Optimization — Img2Col / GEMM / Memory Ops

Analysis Model

Algorithm Analysis
Dynamic Cost Measurement
Parameter Configuration

$$Model = \lambda(vs)FPO + \lambda(vs)DM + \sum_{i=1}^{M} \lambda(vs)SpecIns_i$$

- **Hardware Info** — Registers, Cache ...
- **Measurement** — Ld/St, FMA, Broadcast ...

MLIR HW Dialects:

Extended Vector Dialect | GPU Dialect | NPU-Specific Dialect

System Tools:

LLVM Toolchain + Hardware-Specific Toolchain / Toolkit / Library + Execution Engine

Target Platforms:

SIMD / Vector Processors (X86 AVX / Arm Neon / RVV)	GPU Platforms (Nvidia GPUs)	NPU Platforms (Gemmini Accelerator)

3 Buddy Compiler Backend

Buddy Compiler's backend is dedicated to hardware-specific code generation and collaborative design, fully leveraging the acceleration capabilities of hardware architectures. When generic abstractions are insufficient to encapsulate the hardware-specific traits, it creates customized representations and compilation pipelines to enable specific optimization strategies.

[1] Hongbin Zhang et al.: Compiler Technologies in Deep Learning Co-Design: A Survey

[2] Hongbin Zhang et al.: AutoConfig: A Configuration-Driven Automated Mechanism for Deep Learning Compilation Optimization

Authors & Contributors:

Hongbin Zhang, Xulin Zhou, Liutong Han, Taiqi Zheng, Hongyu Lin, Hanghang Cao, Meng Li, Weijia Li, Jiongjia Lu, Mingjie Xing, Yanjun Wu (ISCAS)

Jiuyang Liu, Zikang Liu, Linquan Wei, Yuliang Li, Zhongyu Qin, Sen Yang, Prathamesh Tagore (PLCT Internship)

ISCAS 中国科学院大学