PIM-SW
PIM 컴파일러 구성 및 사용법

2023.12.30

작성자

연세대학교　최희림

Introduction

최근 어플리케이션들이 처리해야 하는 데이터의 양이 증가함에 따라, 데이터 이동 및 메모리 접근에 관련된 성능 문제가 주요 관심사로 부상했다. Processing-in-memory(PIM)은 메모리에서 컴퓨팅 작업을 실행하여 데이터 이동 부하를 줄일 수 있고, 메모리에 병렬적으로 접근 가능한 처리 장치를 활용하여 실행 시간도 줄일 수 있다.

PIM 관련 연구가 진행됨에 따라 타겟 어플리케이션과 기반 메모리 구조에 따라 다양한 하드웨어 구조가 제안되었다. 본 프로젝트에서는 PIM 하드웨어 구조를 연산기의 위치와 연산 방법에 따라 세 가지 주요 범주로 분류한다: Processing Near Memory (PNM), Digital Processing in Memory (Digital-PIM), 그리고 Analog Processing in Memory (Analog-PIM). 본 프로젝트에서는 각 범주에 해당하는 하드웨어를 추상화하여 지원한다.

PIM 하드웨어의 소프트웨어적 지원 측면을 보면, PIM 하드웨어 개발자 측에서 PIM용 커널을 미리 작성하여 특정 조건을 만족한 경우 실행되도록 하는 방식이 일반적이다. 이 접근법은 소프트웨어 개발자가 PIM 하드웨어의 복잡성을 자세히 이해하지 않아도 된다는 이점이 있지만, 확장성과 범용성 측면에서는 한계를 가진다.

본 프로젝트는 컴파일러를 활용하여 이러한 문제를 해결하려 한다. 기존 구현의 효율성을 유지하면서 확장성을 높이는 것이다. 본 문서는 다양한 PIM하드웨어를 지원하기에 적합한 컴파일러의 설계 방식과 그 구조에 대하여 설명한다.

Background: MLIR

본 프로젝트에서 구현한 컴파일러는 MLIR을 기반으로 한다. MLIR (Multi-Level Intermediate Representation)는 LLVM 컴파일러 개발 그룹에 의해 개발된 컴파일러 프레임워크이다. MLIR은 다양한 계층의 중간 언어의 구현과 이러한 중간 언어들 간의 변환 및 최적화 구현을 지원한다. LLVM을 기반으로 하는 만큼 MLIR은 단일 중간 언어에서의 최적화 구현도 지원한다. MLIR 프로젝트는 기존 컴파일러의 소프트웨어 단편화 문제를 해결하기 위해 다양한 도메인 및 여러 하드웨어 플랫폼을 대상으로 하는 컴파일러의 구현을 지원한다. 다양한 도메인 특화 언어(Domain-Specific Language, DSL)을 쉽고 효율적인 지원하는 것이 가능하여 머신러닝, 고성능 컴퓨팅, 임베디드 시스템 등 다양한 컴퓨팅 환경과 언어 등에서 점점 영향력을 넓히고 있다. 하드웨어 중립적으로 설계되어 있기 때문에 다양한 PIM 하드웨어를 타겟으로 하는 컴파일러를 설계해야 하는 본 프로젝트의 목표에 적합하다.

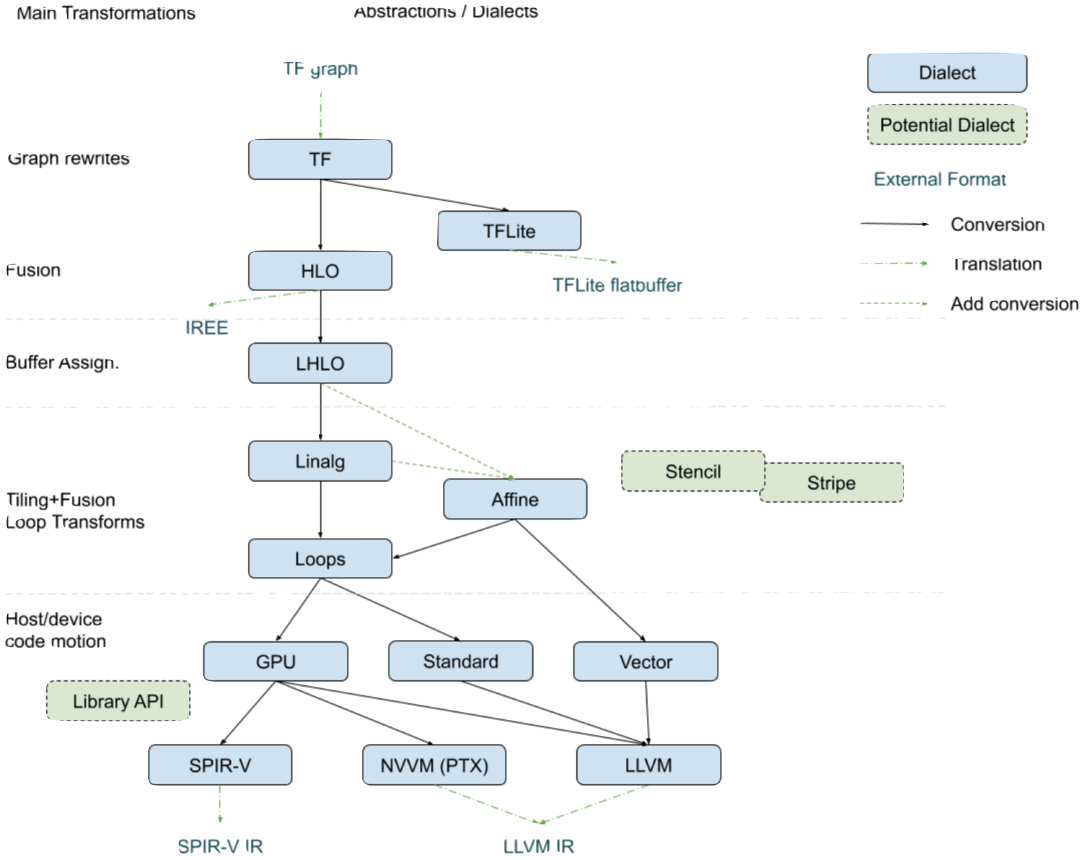

그림1. MLIR Dialect(IR) 계층구조

Compiler Design

컴파일러는 상위 수준의 API 함수를 입력받아 MLIR 프레임워크에서 처리 가능한 형태로 front-end, 중간언어 수준에서의 변환 및 최적화를 수행하는 middle-end, 중간언어 코드를 PIM 하드웨어 동작을 위한 코드로 변환하는 back-end로 구성되어 있다. 전체적인 구조는 그림2에, 중간언어의 구성 operation은 표1에 작성되어 있다.

먼저 front-end는 다양한 언어로 작성된 API function을 MLIR 로 변환해주는 역할을 한다. front-end 컴파일러를 바닥부터 다시 구현하는 것은 굉장히 비효율적이며, 본 컴파일러의 주요 목적과도 맞지 않기 때문에 MLIR 기반의 C언어 frontend인 Polygeist를 용도에 맞게 수정하여 활용하였다. PIM 아키텍처의 연산 유닛을 기반으로 공통으로 지원하는 연산을 정리하여 PIM general IR을 설계 및 구현하였다.

middle-end의 경우 PIM general IR 에 더하여 하드웨어 특성을 반영한 IR인 PNM IR / D-PIM IR / A-PIM IR로 구성된다. 하드웨어별 IR은 각 하드웨어의 연산 단위, 메모리 접근 방식 등을 고려하여 구성하였다. 그리고 PIM general IR에서부터 하드웨어별 IR로 변환하는 과정과 각 IR별 최적화 등을 포함한다. 이때, IR 사이의 변환을 lowering이라고 부른다.

back-end의 경우 실제 하드웨어에서 동작하는 형태로 코드를 변환하는 과정을 포함한다. 본 프로젝트에서는 시뮬레이터 환경을 가정하고 있으므로 memory transaction등의 형태로 결과 코드가 생성된다.

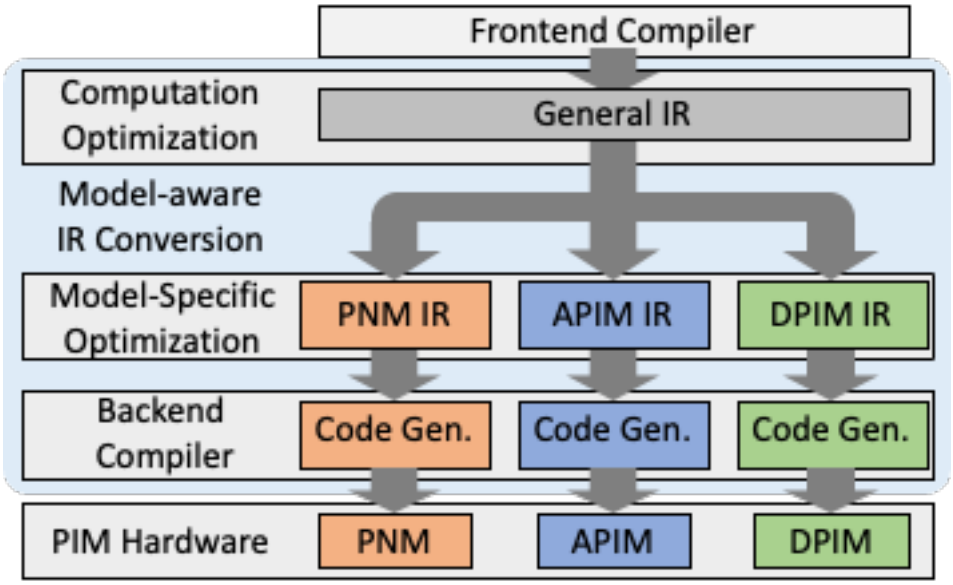

그림2. PIM 컴파일러 구조

General PIM IR	Hardware specific IR		
	APIM IR	DPIM IR	PNM IR
pim.simd_add_op	apim.vector_op	dpim.vector_comp	pnm.vector_op
pim.simd_sub_op	apim.vector_imm_	dpim.read_op	pnm.vector_imm_o
pim.simd_mul_op	apim.mat_vec_op	dpim.ganged_write	pnm.set_op
pim.simd_div_op	apim.set_op	dpim.ganged_act_	pnm.copy_op
pim.simd_scal_add	apim.copy_op		pnm.load_op
pim.simd_scal_sub	apim.load_op		pnm.store_op
pim.simd_scal_mul	apim.store_op		
pim.simd_scal_div	apim.send_op		
pim.mac_op	apim.receive_op		
pim.acc_op			

표 1. 중간언어를 구성하는 operation 목록

Conclusion

본 프로젝트의 컴파일러는 상위 수준의 API 함수를 MLIR 프레임워크를 통해 PIM에서 실행 가능한 형태로 변환하는 역할을 수행한다. 본 컴파일러는 front-end, middle-end, 및 back-end의 세 부분으로 구성되어 있으며, 각 부분은 PIM 하드웨어의 특성에 맞게 최적화된 작업을 담당한다. Front-end는 다양한 프로그래밍 언어로 작성된 함수를 MLIR 형식으로 변환하고, Middle-end는 PIM general IR과 함께 하드웨어별 특성을 반영하는 여러 IR로 구성되어 있으며, 이들 간의 변환과 최적화를 수행합니다. 마지막으로, back-end는 최종적으로 실제 하드웨어에서 실행될 수 있는 코드로 변환한다. 본 컴파일러는 PIM 기술의 효율적인 활용을 위한 기반을 제공하며, 다양한 애플리케이션 및 하드웨어 환경에서 PIM이 활용되기 위한 기초를 쌓았다는 의의가 있다.