Compiling Halide Programs to our CGRA
Jeff Setter

Halide application → CGRA hardware

// Algorithm
brighten(x, y) = input(x, y) * 2;
blur(x, y) = (brighten(x, y) + brighten(x+1, y) +
brighten(x, y+1) + brighten(x+1, y+1))/4;

// Schedule
blur.in().tile(x, y, xo, yo, xi, yi, 63, 63)
  .hw_accelerate(xi, xo);
brighten.store_at(blur.in(), xo)
  .compute_at(blur.in(), xo);
input.stream_to_accelerator();
Unified Buffer: Compiling Halide Programs to Push-Memory Accelerators

Qiaoyi (Joey) Liu, Dillon Huff, Jeff Setter, Maxwell Strange, Mark Horowitz, Priyanka Raina, Fredrik Kjolstad
Introduction

Motivation

• Accelerators use highly optimized push memories
• Many efforts describe memory designs, but not how to compile to memory

Challenges

• Difficult to generalize the mapping process
  • Frontend algorithm: Stencil + DNNs
  • Backend architecture: HLS, library-based method

• Hard to optimize the compile result:
  • Large design space
  • Manual effort for a specific application
Solution: the Unified Buffer

- Create an abstraction
  - Describe the information **when** and **where** data flows
  - Bundled with memory **port**
  - In terms of **operations**
- Leverage compiler Optimization
  - Polyhedral Analysis
  - Vectorization
Modification Idea!
Real Application Benchmark Suite → Compiler → RTL → Measure Performance
Real Application Benchmark Suite → Compiler → RTL → Measure Performance

Broken!
Real Application Benchmark Suite → Compiler (Manually Modify) → RTL → Measure Performance
Probably Still Broken!
Need to test and debug
Finally can benchmark but that was time consuming and error prone!
Many Modifications To Try!!

Real Application Benchmark Suite -> Compiler -> RTL -> Measure Performance

ISA.pdf
Automatically Synthesize Rewrite Rules using SMT

Real Application Benchmark Suite

Only need to modify the RTL!

Compiler

RTL

Measure Performance
Synthesizing Rewrite Rules for Diverse Architectures

Ross Daly, Caleb Donovick, Jack Melchert, Raj Setaluri, Nestan Tsiskaridze, Priyanka Raina, Clark Barrett, Pat Hanrahan
Olivia Hsu

Accelerating Sparse Tensor Algebra

Too many tensor kernels for fixed-function libraries and backends

\[ a = Bc + a \quad a = Bc \]  \( a = Bc + b \quad A = B + C \quad a = \alpha Bc + \beta a \)
\[ a = B^Tc \quad A = \alpha B \quad a = B(c + d) \]
\[ a = B^Tc + d \quad A = B + C + D \quad A = BC \]
\[ A = B \odot C \quad a = b \odot cA = 0 \quad A = B \odot (CD) \]
\[ A = BCd \quad A = B^T \quad a = B^T Bc \]
\[ a = b + c \quad A = B \quad K = A^TCA \]

\[ A_{ij} = \sum_{kl} B_{ikl}C_{lj}D_{kj} \quad A_{kj} = \sum_{i} B_{iki}C_{ij}D_{ij} \]
\[ A_{ij} = \sum_{ik} B_{ikl}C_{lj}D_{kj} \quad A_{ij} = \sum_{k} B_{ijk}C_k \]
\[ A_{ijk} = \sum_{kl} B_{ikl}C_{lj} \quad A_{ik} = \sum_{k} B_{ikl}C_k \]
\[ A_{jk} = \sum_{i} B_{ijk}C_i \quad A_{ij} = \sum_{k} B_{ikl}C_{kj} \]

\[ C = \sum_{ijkl} M_{ij}P_{jk}M_{kl}P_{li} \]
\[ a = \sum_{ijklmnop} M_{ij}P_{jk}M_{kl}P_{lm}M_{nm}P_{no}M_{po}P_{sp} \]

Linear Algebra

Dense Matrix
- CSR
- DCSR
- BCSR

COO
- ELLPACK
- CSB

Blocked COO
- CSC

DIA
- Blocked DIA
- DCSC

Sparse vector
- Hash Maps

Data analytics (tensor factorization)

Coordinates
- Dense Tensors
- Blocked Tensors

Linked Lists
- Database

Compression Schemes
- Cloud Storage

CPU

GPUs

TPUs

FPGA

CGRAs

Sparse Tensor Hardware

Cloud Computers

Supercomputers

\( \tau = \sum_{i} z_i \left( \sum_{j} z_{ij} \theta_{ij} \right) \left( \sum_{k} z_{ik} \theta_{ik} \right) \)

reorder

precompute

parallelize

split

map

divide

vectorize

unroll

position
Olivia Hsu

Accelerating Sparse Tensor Algebra

Long Tail of Expressions (algorithm) \times Varying Compression Structures (format) \times More Performant Backends (platform) \times Backend-Specific Transformations (schedules)

Users \leftrightarrow Performance \leftrightarrow Usability \leftrightarrow Accelerators
Olivia Hsu

Accelerating Sparse Tensor Algebra

Legend

- Sparse Dataflow Compiler
- Other
- TACO Literature
PEak:
The Single Source of Truth

Caleb Donovick
2.2 Base Instruction Formats

In the base RISCV ISA, there are four core instruction formats (R1/R2/R3), as shown in Figure 2.2. All core instructions have fixed-length fields and are defined by a sequence of registers. A branch instruction utilizes untagged exceptions generated as a simple branch (assuming all target synonyms are in a hardware-aligned register. No instruction for hardware-aligned exception is generated for a conditional branch that is not taken.

The alignment provided for these ISA instructions is shown in the left boundary where instructions can be removed from left side labels of the table. The second column contains a symbolic code that identifies the instruction. The third column contains a symbolic code that identifies the instruction type.

![Instruction Formats](image)

Figure 2.2: RISCV Instruction Formats. Each immediate field is labeled with the bit position [0-31] to indicate the immediate value.

The RISCV ISA also contains core instructions, each with a length of 32 bits. The following instruction formats are shown in Figure 2.3.

![Instruction Formats](image)

Figure 2.3: RISCV Instruction Formats. Each immediate field is labeled with the bit position [0-31] to indicate the immediate value.

2.3 Immediate Encoding Variants

There are four immediate variants of the instruction format (R1/R2/R3) based on the handling of immediates, as shown in Figure 2.2.

![Immediate Variants](image)

Figure 2.4: Immediate Variants. The fields are labeled with the bit position [0-31] to indicate the immediate value.

Textual Specification

- Functional Model
- RTL
- Formal Model
What went wrong?

- Functional Model might be wrong
- RTL might be wrong
- Might be unspecified behavior
- Tests might differ
Textual Specification

Functional Model

RTL

Formal Model

Counter Example

Formal Tools
No way to test textual specification
PEak has many features

See my poster
Automated Design Space Exploration of CGRA Processing Element Architectures using Frequent Subgraph Analysis

Jackson Melchert, Kathleen Feng, Caleb Donovick, Ross Daly
How can we generate an optimal PE architecture?

1. Analyze application domain benchmarks to find possible optimizations
2. Quickly create PE designs that explore the design space
3. Automatically generate full compiler to run applications
Application Analysis - Frequent Subgraph Mining

Frequent subgraphs represent common computational blocks
Producing PEs - Frequent Subgraph Merging

Merging frequent subgraphs results in efficient and performant PEs

Subgraph 1

Subgraph 2

Reconstructed Merged Graph

Merging frequent subgraphs results in efficient and performant PEs
Design Space Exploration Framework
Dynamic Partial Reconfiguration

Kalhan Koul - Rising 3rd Year PhD
Motivation for Dynamic Partial Reconfiguration

- Definition: Reconfigure parts of the CGRA (partial) without affecting other parts at runtime (dynamic)
- Example: fixed-function accelerators vs reconfigurable accelerators

**Workload over time**

<table>
<thead>
<tr>
<th>DSP</th>
<th>Video + ML</th>
<th>Crypto</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Video Accelerator</td>
<td>MEM</td>
</tr>
<tr>
<td>MEM</td>
<td>ML Accelerator</td>
<td>BSP Accelerator</td>
</tr>
<tr>
<td>MEM</td>
<td>Crypto Accelerator</td>
<td>Crypto</td>
</tr>
</tbody>
</table>

**DSP + Video + ML + Crypto**
- + Highly optimized accelerators
- - Cannot add new accelerators
- - Low hardware utilization

**DSP->(Reconf.)->Video+ML->(Reconf.)->Crypto**
- + Flexible to run any accelerator
- + High hardware utilization
- - Reconfigurable hardware overhead
- - Reconfiguration time overhead
• Goal: develop and quantify the benefits of hardware architectural additions on top of a baseline CGRA, including: (a) Relocatable bitstream (b) Partial Reconfiguration (PR) region shape and interconnect network (c) Parallel Reconfiguration (d) Double Buffer Reconfiguration

• Visit my poster to see the implementation/exploration of each!

Ex. Parallel Reconfiguration

Ex. Double Buffer Reconfiguration
Lack of abstraction and imprecise specifications leads to debugging using gate level simulations.

**Smart Components** surfaces these bugs in RTL:
- **Abstract actions** capture component interfaces
- **Session types** verify composition of components
- **Unit testing** verifies concrete implementations
Synthesizing Formal Models of Hardware from RTL for Efficient Verification of Memory Model Implementations

Yao Hsiao, Dominic P. Mulligan*, Nikos Nikoleris*, Gustavo Petri*, Caroline Trippel
Stanford University, *ARM Research

• **Motivation:**
  - Memory Consistency Models (MCM) specify legal outcomes of shared memory programs in multiprocessor
  - Check tools requires manually-constructed formal microarchitecture specifications (μspec models) as input

• **Goal:** Efficient Check-based verification of hardware MCM implementations

• **Key Challenges:**
  - Decomposition of complete μspec models
  - Gap between operational RTL and axiomatic μspec models
Improving Energy Efficiency for DNNs on CGRAs with Local Storage in the PEs

Ankita Nayak

- With proper blocking schemes many dataflows can achieve close-to-optimal energy efficiency
- Memory resource allocation has a larger impact on DNN energy
- **Goal:** Introduce a new low-access-cost memory hierarchy (Ponds) to improve energy efficiency
- **Challenge:** Should be extremely area and energy efficient, yet flexible enough to map different energy efficient schedules

Design space of dataflow with optimal loop blocking schemes

DNN Accelerator Template with Hierarchical Memories

Introducing Ponds in Amber CGRA
Keyi Zhang

System-Level SoC Verification Framework

Thread 1
(1) A = 1
(2) print(B)

Thread 2
(3) B = 1
(4) print(A)

Core 1
L1 Cache
L2 Cache
L3 Cache

Core 2
L1 Cache
L2 Cache

Interrupt not received properly?

Is there a bug? HW or SW?
Utilizing Hardware Generators for Agile RTL Refinement

Raj Setaluri, Alex Carsello, James Thomas, and Christopher Torng

Stanford University
RTL refinement is a slow, iterative process

- Architectural design-space exploration
- Early RTL prototyping
- RTL Refinement

- Refinement is a cross-team process
- Bogged down by tool spin-time
- Have to up-level reports to source
A Source-Level Development Platform for Agile RTL Refinement

- Intelligently slice the circuit to get the parts you care about
- Query against source-level names
- Abstract away tool complexity

```plaintext
u_over_4 = u >> 2
u_over_2 = u >> 1
u_over_2_plus_B_over_2 = csa(u_over_2, B) >> 1
u_plus_B_over_4 = csa(u, B) >> 2
u_plus_B_over_2 = csa(u, B) >> 4
u_plus_B_over_2_B_over_2 = csa(u_plus_B_over_2, B) >> 1
odd_update_0 = odd_update(csa(u, y), B)
odd_update_1 = odd_update(csa(u, y), B)

>>> report_timing(from_=[u, B], to=[u_plus_B_over_2_B_over_2])
{
  "B[0]": 0.01,
  "add_inst2.in1[0]": 0.01,
  "add_inst2.out[15]": 1.12,
  "lshr_inst4.in0[15]": 1.12,
  "lshr_inst4.out[9]": 1.21,
  "add_inst3.in0[9]": 1.21,
  "add_inst3.out[14]": 1.57,
}
A General-Purpose Memory System for Data-Intensive Accelerators

James Thomas
Data-Intensive Accelerator Design Platform Required

- Designing complex accelerator from scratch is way too expensive

- CUDA programming is relatively easy -- you write code for one thread and then it is run in parallel on a huge number of cores to get high performance
  - Can we have a similar model for accelerator design?
CUDA-like Accelerator Design Model

- Design and verify a single processing element (PE) that communicates in an AXI-like protocol to memory
- Platform replicates this PE (100x+) into memory access fabric that handles communication with DRAM
Toy CGRA:
Evaluating the Technology Portability of Agile Hardware Design Flow

Po-Han Chen
Overview

- Toy CGRA is a course project of EE272B

Stanford AHA! Tool Chain
- HW Generation
- App Scheduling
- App/HW Mapping

OpenROAD Digital OpenLane flow

SRAM Macro

130nm Technology

EE 272B fast tape-out in 3 months!
Copy-and-Patch Compilation

Haoran Xu and Fredrik Kjolstad
Stanford University
The Need For Fast Compilation

* JIT compilers: compilation at runtime.
  - database engine: SQL query → machine code
  - web browser: WebAssembly module → machine code

* Need to compile fast **AND** generate good code!

* Our solution: Copy-and-Patch.

* Provides extremely fast compilation **AND** decent generated code.
The Need For Fast Compilation

* JIT compilers: compilation at runtime.
  - database engine: SQL query → machine code
  - web browser: WebAssembly module → machine code
* Need to compile fast **AND** generate good code!
* Our solution: Copy-and-Patch.
* Provides extremely fast compilation **AND** decent generated code.
Copy-and-Patch Compilation

* Two example use cases: SQL compiler, WebAssembly compiler.

* Significantly outperforms existing approaches for fast compilation:
  - LLVM -O0 (100x faster compilation, 15% better code)
  - State-of-the-art baseline compilers from Chrome and Wasmer (5-20x faster compilation, 50%-60% better code)
  - Interpreters (10x faster execution)

* Works for both high-level languages (C-like) and low-level bytecodes (WebAssembly-like).
How it works

* No free lunch.
* But we can stand on the shoulders of giants.
* Cleverly using Clang+LLVM as a preprocessor.
* Pre-compute a lot, so little work to do at runtime.
* For more details: poster!
Fast Extended GCD for Large Integers for Verifiable Delay Functions

Kavya Sreedhar, Mark Horowitz, Christopher Torng
Verifiable delay function (VDF)

Allows one party (prover) to convince the other party (verifier) that a certain amount of time has passed

Mathematical Puzzle

\[ F(x) = y \]

Delay: Inherently sequential work that is slow to compute

Verifiable: The output of the puzzle is easy to verify to be correct
The crypto community is excited about VDFs

Chia Network Announces 2nd VDF Competition with $100,000 in Total Prize Money

Protocol Labs and Ethereum Foundation Team Up to Research Verifiable Delay Functions

At the cutting edge of blockchain research is a potential $15 million dollar venture by the Ethereum Foundation centered around a technology called Verifiable Delay Functions (VDFs).

The speed of VDF evaluation directly impacts the security of these blockchains
Accelerating VDFs depends on fast extended GCD computations

Extended GCDs are cool again!

Come hear about arithmetic circuit optimizations in the context of fast VDFs

Come to our poster!