OpenCL Host Framework

Easy Kernel Development, Zero Host Code

60-Minute Technical Presentation

Agenda

Part I: OCLExample Introduction

- Introduction / Features
- General OpenCL Host Flow
- Standard CL vs CL Extension
- Caching Mechanism
- JSON Configuration

Part II: Two-Phase Architecture

- Recap
- Why Two-Phase Architecture
- Compilation Phase
- Execution Phase
- Cross-Platform Deliverable
- Summary & Q&A

The Problem We Solve

Traditional OpenCL Development Pain Points

- ✗ Writing 500+ lines of host code for each algorithm
- ✗ Managing OpenCL contexts, queues, buffers manually
- ✗ Compiling kernels and handling errors
- ✗ Setting up arguments and work sizes
- ✗ Verifying results against CPU reference
- ✗ Repeating this for every kernel variant

Solution: This framework eliminates all the boilerplate!

Introduction / Features

- The problem: 500+ lines of boilerplate for each OpenCL algorithm
- The solution: JSON-driven configuration, zero host code
- Six key benefits of the framework
- User journey: Build → Run → Learn
- High-level architecture overview

Framework Benefits

✓

Zero Host Code

JSON-driven configuration

✓

Instant Comparison

Run multiple variants

✓

Platform Switching

Standard GPU vs CL Extension

✓

Auto-Verification

GPU output vs CPU reference

✓

Binary Caching

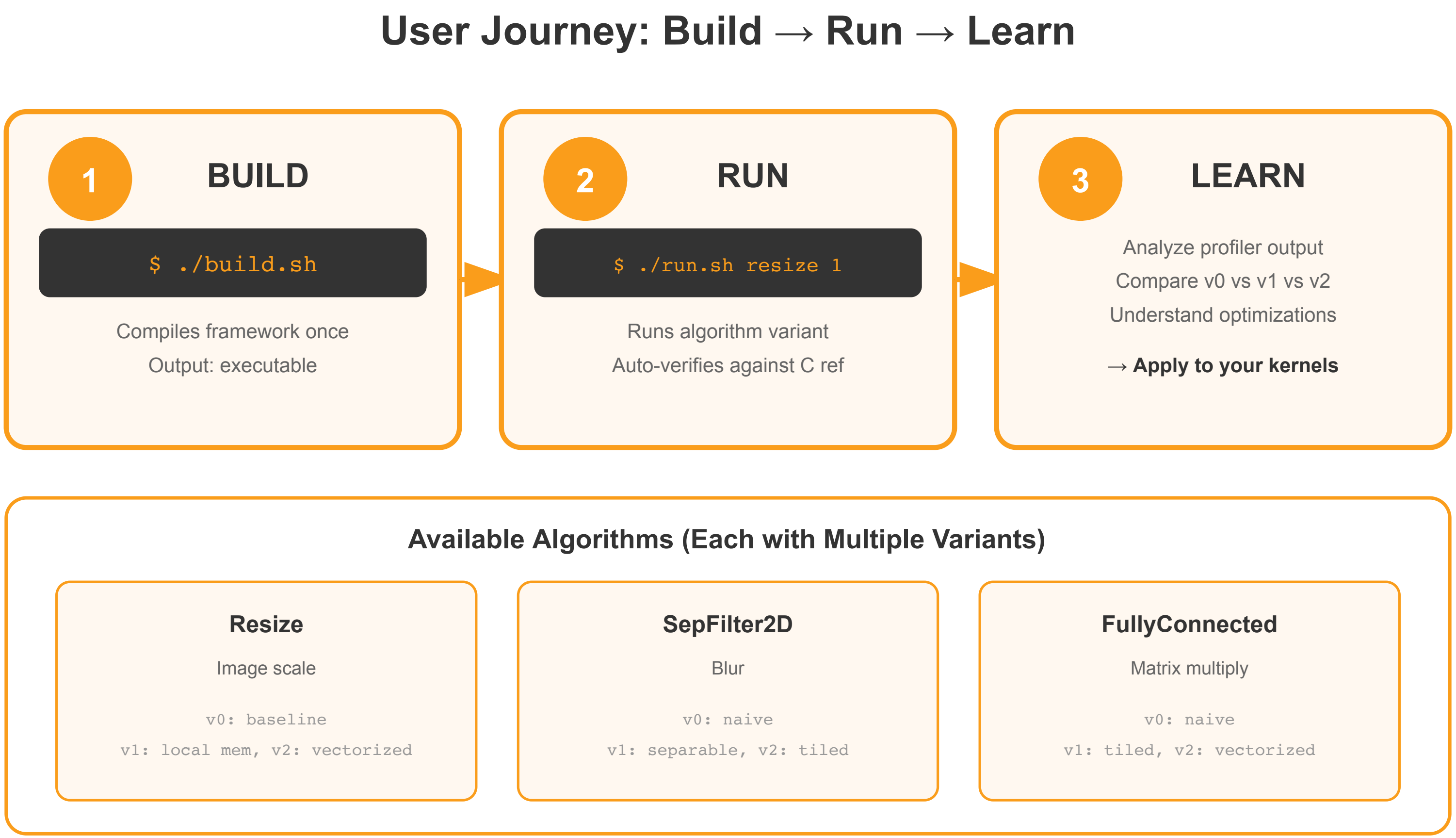
Fast re-runs with cached kernels

✓

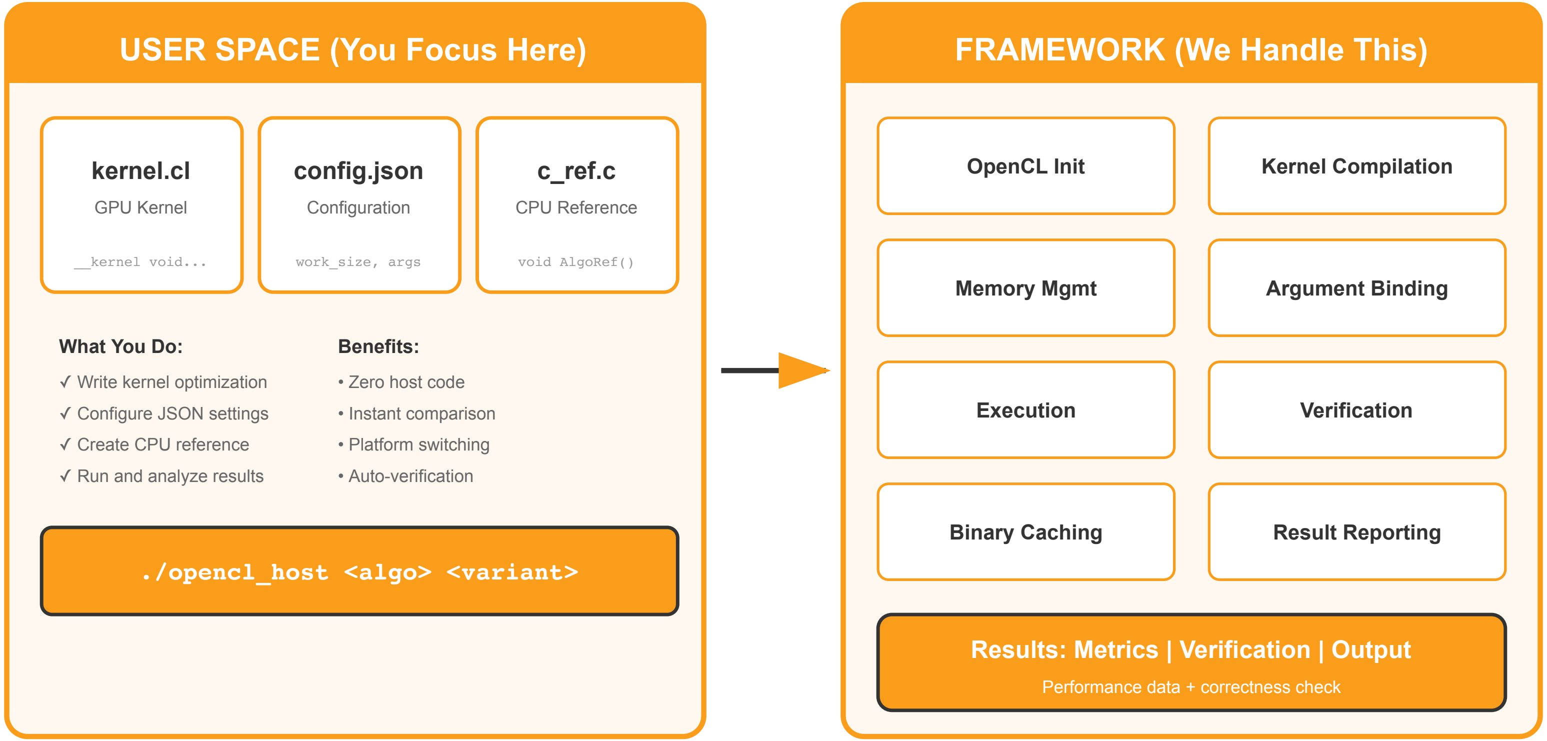
Easy Extension

Add new algorithms with scaffold

User Journey: Build → Run → Learn



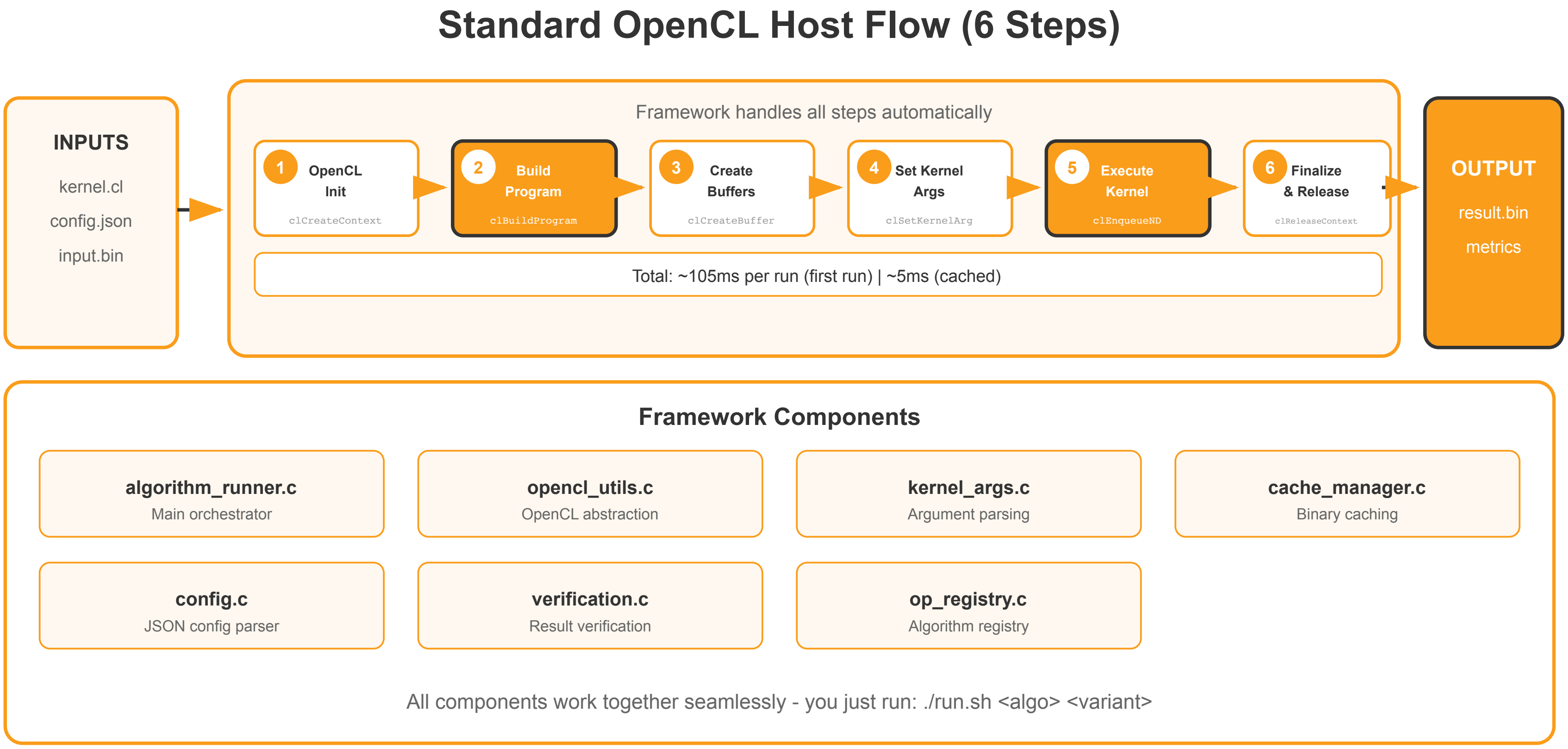
High-Level Architecture



General OpenCL Host Flow

- The 6-step standard OpenCL host flow
- What the framework handles automatically
- Key timing: ~100ms compilation overhead
- Why this matters for iteration speed

Standard OpenCL Host Flow



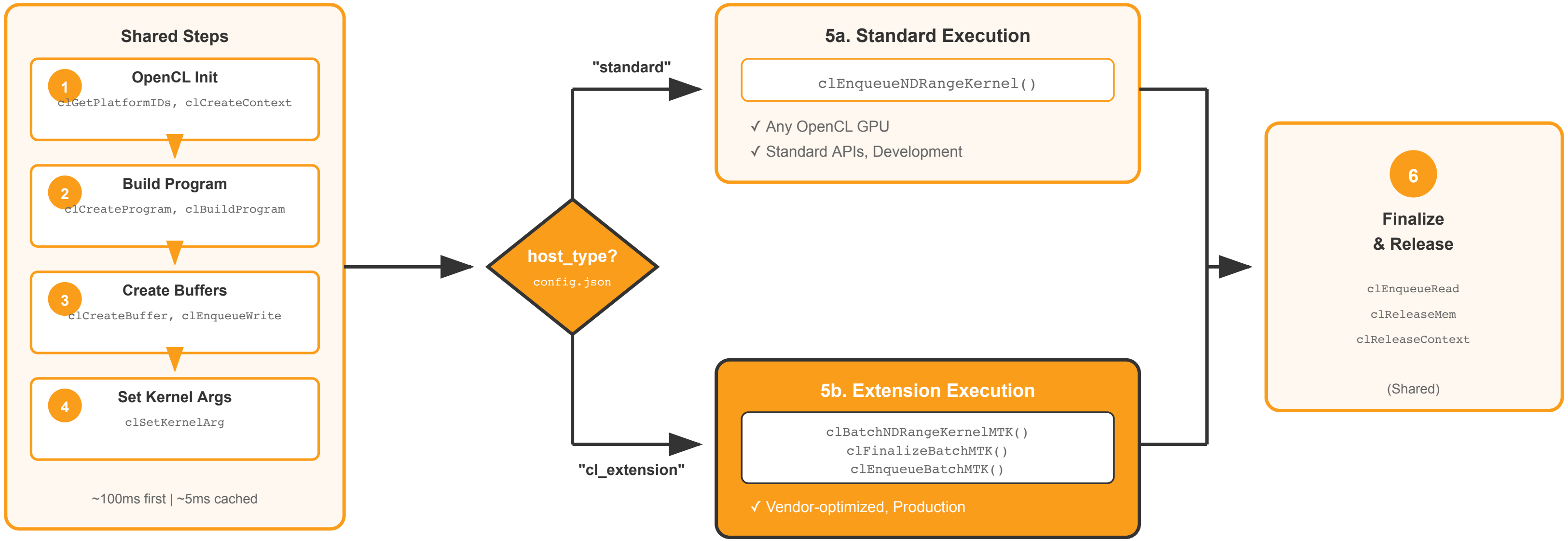
Standard CL vs CL Extension

- Two platform types: Standard OpenCL and CL Extension
- Standard CL: Works on any GPU, great for development
- CL Extension: Vendor-optimized, maximum performance
- One-line config change to switch between them

Platform Branching

Standard OpenCL vs CL Extension

Shared flow with branching only at kernel execution



Key Insight

Only Step 5 differs: kernel execution

Same kernel.cl works on both!

One-Line Config Switch

`"host_type": "standard"`

`"host_type": "cl_extension"`

← Steps 1-4 → Decision → Step 5a/5b → Step 6 →

Platform Configuration in JSON

```
"kernels": {
  "v0": {
    "host_type": "standard",
    "kernel_file": "resize1.cl"
  },
  "v1": {
    "host_type": "cl_extension",
    "kernel_file": "resize2.cl"
  }
}
```

Key Benefits

- ✓ Runtime platform selection
- ✓ No code changes required
- ✓ No rebuild needed
- ✓ Switch with one line change
- ✓ Run both variants to compare
- ✓ Same kernel.cl works on both

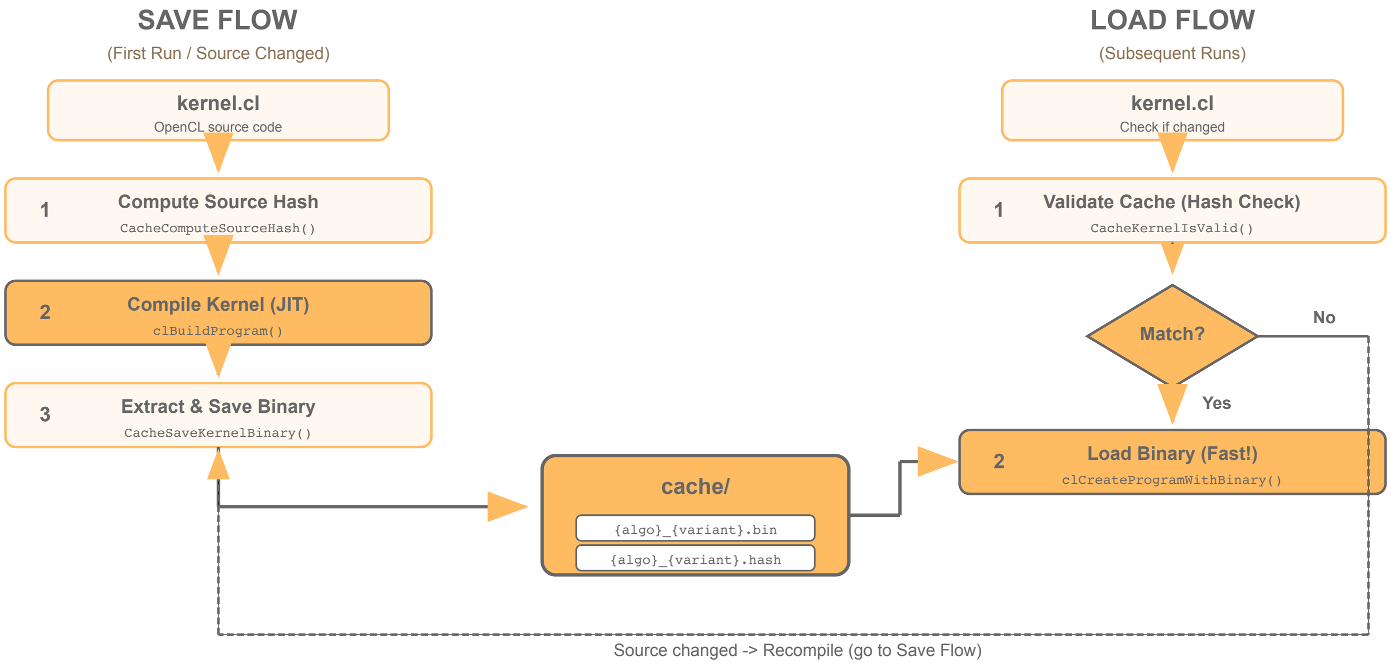
Easy to switch between Standard CL and CL Extension

Caching Mechanism

- Binary caching eliminates repeated compilation
- Cache miss: First run compiles (~100ms), saves binary
- Cache hit: Load cached binary (~5ms) — 20x faster!
- Automatic invalidation when source changes (SHA256)

Binary Caching

OpenCL Kernel Binary Cache Mechanism



Performance Gain

First run: ~100ms (compile)

Cached: ~5ms (load binary)

20x faster!

How It Works

1. Hash source code to detect changes
2. If hash matches -> load cached binary
3. If hash differs -> recompile and update cache

Code Reference

`cache_manager.c`

`opencl_utils.c:438-519`

Hash: FNV-1a (16 bytes)

JSON Configuration

- Configuration file structure in config/ directory
- Global inputs.json and algorithm-specific configs
- Complete kernel configuration with all parameters
- Six supported argument types for any kernel pattern

```
config/
├── inputs.json           # Global input definitions
├── resize.json           # Algorithm-specific
├── sepfilter2d.json      # Algorithm-specific
└── fullyconnected.json   # Algorithm-specific
```

Kernel Configuration

```
"kernel": {
  "description": "standard cl implementation",
  "host_type": "standard",
  "kernel_option": "-g",
  "kernel_file": "examples/resize/cl/resize0.cl",
  "kernel_function": "resize",
  "work_dim": 2,
  "global_work_size": [1920, 1088],
  "local_work_size": [16, 16],
  "kernel_args": [
    {"i_buffer": ["uchar", "src"]},
    {"o_buffer": ["uchar", "dst"]},
    {"param": ["int", "src_width"]}
  ]
}
```

Supported Argument Types

Type	Syntax	Description
Input Buffer	{"i_buffer": ["uchar", "src"]}	Read-only image
Output Buffer	{"o_buffer": ["uchar", "dst"]}	Write-only result
Parameter	{"param": ["int", "width"]}	Auto from image
Scalar	{"scalar": ["float", "sigma"]}	Custom value
Custom Buffer	{"buffer": ["float", "wts", 100]}	User-defined
Struct	{"struct": ["a", "b", "c"]}	Packed scalars

Part I Summary

Topic	Key Points
Introduction / Features	Zero host code, JSON-driven configuration
OpenCL Host Flow	6-step flow handled by framework
Platform Options	Standard CL vs CL Extension — one-line switch
Caching	Binary caching: 100ms → 5ms
Configuration	JSON-based kernel and I/O setup

Any questions before Part II?

Agenda

Part I: OCLExample Introduction

- Introduction / Features
- General OpenCL Host Flow
- Standard CL vs CL Extension
- Caching Mechanism
- JSON Configuration

Part II: Two-Phase Architecture

- Recap
- Why Two-Phase Architecture
- Compilation Phase
- Execution Phase
- Cross-Platform Deliverable
- Summary & Q&A

Part I Recap

Framework Benefits

Zero host code, JSON-driven

User Journey

Build → Run → Learn

Platform Options

Standard CL vs CL Extension

JSON Configuration

Kernel config, I/O setup

Now: How we split the host into two phases

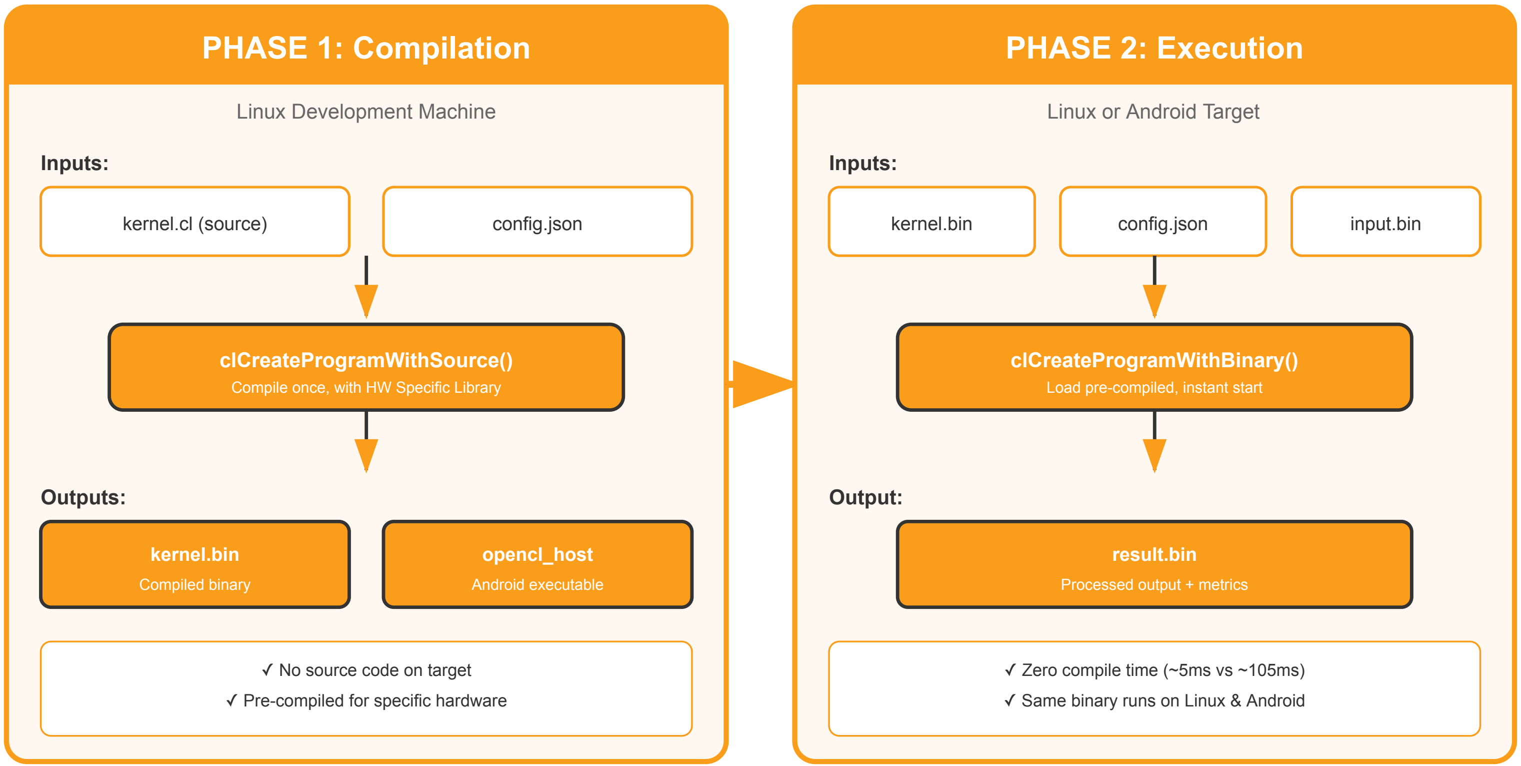
Why Two-Phase Architecture

- ✗ Every run recompiles kernel (~100ms overhead)
- ✗ Source code must be present on target
- ✗ Can't pre-compile for different devices
- ✗ No binary distribution possible

Solution: Split into Compilation + Execution phases

Two-Phase Architecture

Two-Phase Architecture: Compile Once, Run Anywhere



Compilation Phase (WIP)

- Runs once on Linux development machine
- Input: kernel.cl source + config.json
- Output: kernel.bin (compiled) + gpu_host (executor) OR mvpu_host (executor)
- kernel.bin includes metadata for target device (how about batch binary ?!)

Compilation Phase (WIP)

Purpose: Convert source + config into deliverable-ready binaries

Inputs

- kernel.cl — Source code
- config.json — Configuration

Outputs

- kernel.bin — Compiled binary
- gpu_host — Linux executable
- mvpu_host (Player) — Linux executable
- mvpu_host (Player) — Android executable

kernel.bin contains:

- Compiled kernel binary (device-specific)
- Metadata (device name, OpenCL version)
- Compile options used

Execution Phase (WIP)

- Runs many times on target device (Linux/Android)
- Input: kernel.bin (pre-compiled) + config + data
- Uses clCreateProgramWithBinary() — no compilation!
- Performance: ~5ms vs ~105ms — 20x faster startup

Execution Phase (WIP)

Purpose: Run kernel without compilation overhead

Inputs

- kernel.bin — Pre-compiled
- config.json — Runtime config
- Input data

Output

- result.bin — Processed output

Using clCreateProgramWithBinary()

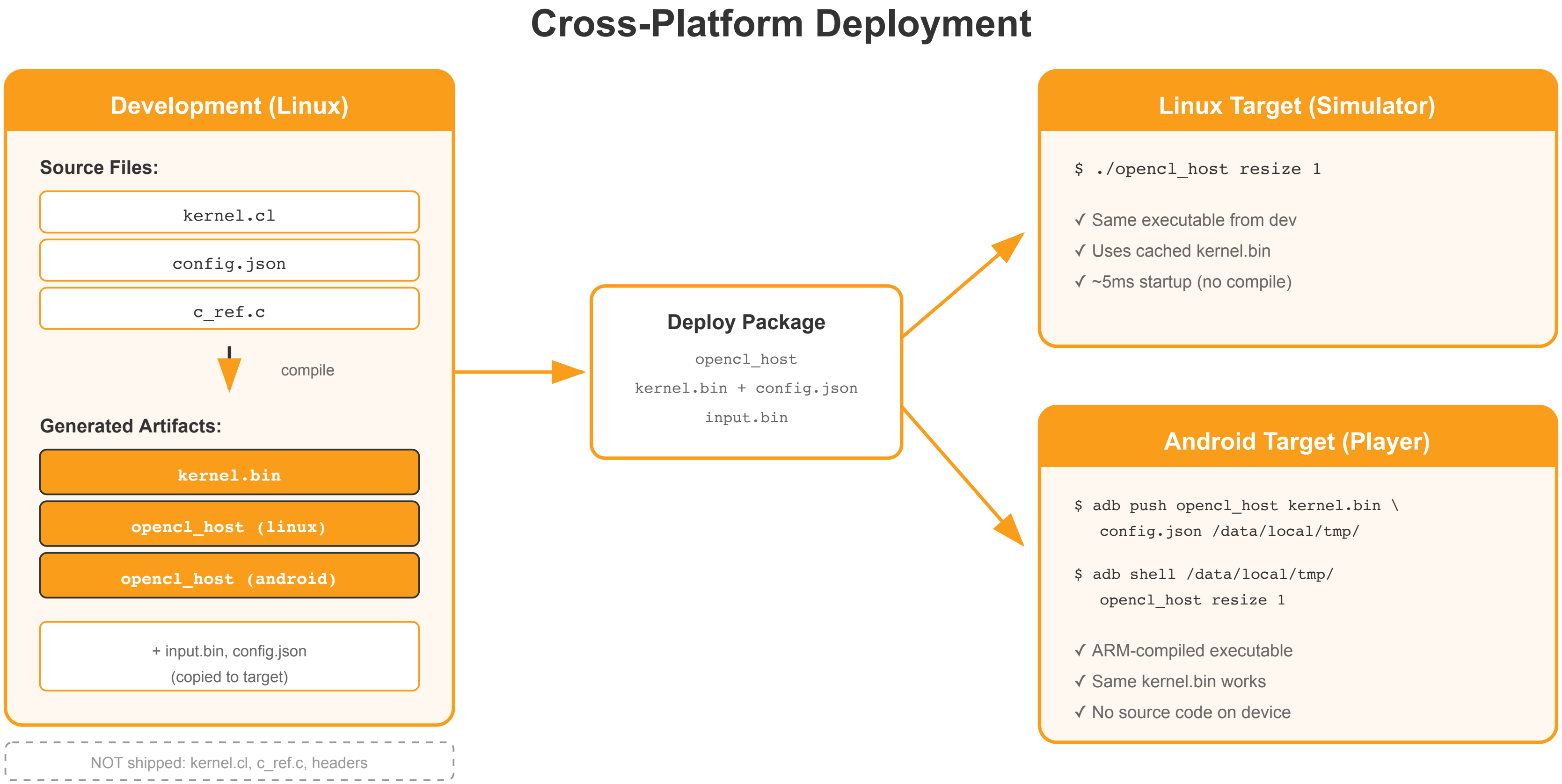
Performance: ~5ms vs ~105ms (with compilation)

Loading pre-compiled binary eliminates 100ms compilation overhead

Cross-Platform Deliverable

- Develop and compile on Linux (full toolchain)
- Deploy to Linux (GPU/MVPU) OR Android (MVPU) devices
- Only 3 types of files: opencl_host(s) + kernel.bin + config.json
- Deliver offline compiler: No host source code, headers are needed on target

Cross-Platform Deliverable



What to Deliver

Execution Package:

```
out/
├── opencl_host(s)      # Executor binary
├── kernel.bin          # Compiled kernel
└── config.json         # Runtime config
```

NOT required in execution:

✗ kernel.cl (source code)

✗ include/ (headers)

✗ Host code build tools

✗ Host code development dependencies

Benefits: Smaller footprint • No source exposure • Faster startup

Summary

Phase	When	Where	Output
Compilation	Once	Linux	kernel.bin + opencl_host
Execution	Many times	Linux/Android	result.bin

Key Benefits:

✓ Zero compile time after first run

✓ Binary-only deliverable

✓ Cross-platform support (GPU + MVPU)

✓ Smaller execution footprint

Questions & Discussion

Thank you!