Project Acceleragent - An Agentic Approach to Using LLMs to Port Large HPC Codes to GPUs

Ben Ryan
David Appelhans
Yifei Wu

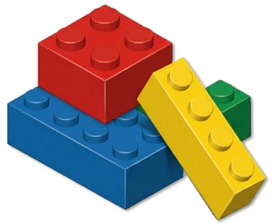
Large Language Models Can Write Good Snippets Fast

- LLMs, especially frontier reasoning models since~ GPT-5 and Claude Sonnet 4.5, are now strong programmers
  - **A narrow, well-posed request** reliably generates a few hundred lines of self-consistent code in ~any language
- However, AI performance **degrades with problem size and complexity**
  - **Try prompting an LLM to read a 5000-line file and produce a correct GPU ported code in one completion. It will fail.**
  - Coding **agents** (Cursor, Claude Code, Codex) that interleave LLM inference with tool calls are **rapidly improving.**
  - But even for the best agents, simple prompting (“Port this code to GPU. No mistakes.”) doesn’t let the user walk away until an entire code is ported correctly, optimally, and suitably for production.

Can We Use Today's LLMs to Port Large Codes Fast?

- **Inspired by human workflow**
  - Humans can produce correct, complex codes after **years** of development and iterative improvement.
  - An LLM only takes a few **minutes** to write lots of code—but can return incomplete or unaligned results.
  - Successful programmers use step by step porting of bite size pieces, testing and fixing along the way.

- A workflow organizing the LLM responses, builds up something correct from smaller pieces.
  - Repeated LLM calls interact with tools and their previous responses.
  - Agents call their own tools and handle more complex tasks
  - **Can we build an agentic workflow to port large codes to GPUs?**



1000x scale separation

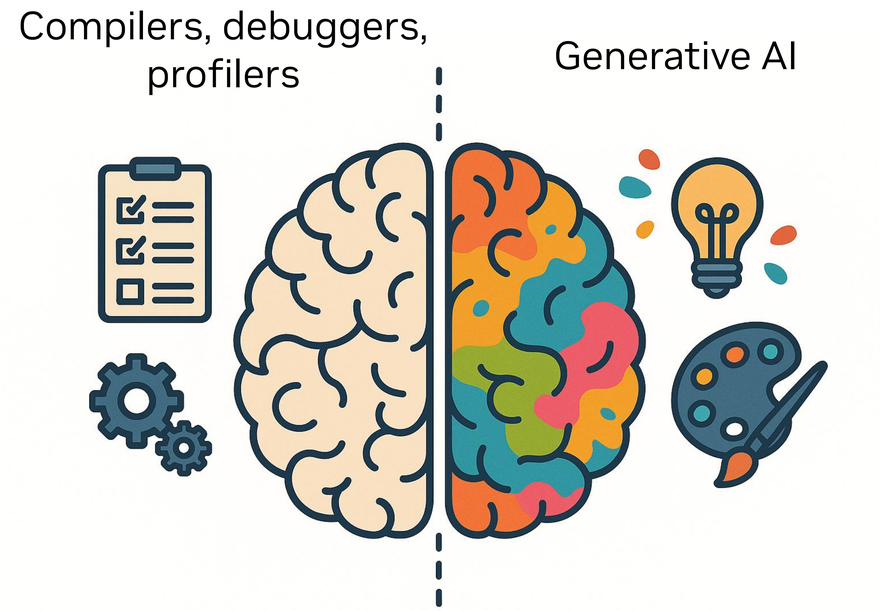
Agent assembling LLM responses

Single LLM response
Small, many shapes

Production target
Large, numerous constraints

Applying AI to Production Codebases

- A refactor of an existing code must **maintain correctness**
  - Production HPC codes have hundreds of tests, users who require stability
- An LLM can’t measure correctness during a response, and only has finite context
- **But compilers and test problems measure correctness and indicate the sources of errors**
- How do we enforce correctness with AI? Return compiler/test results in follow-on prompts!

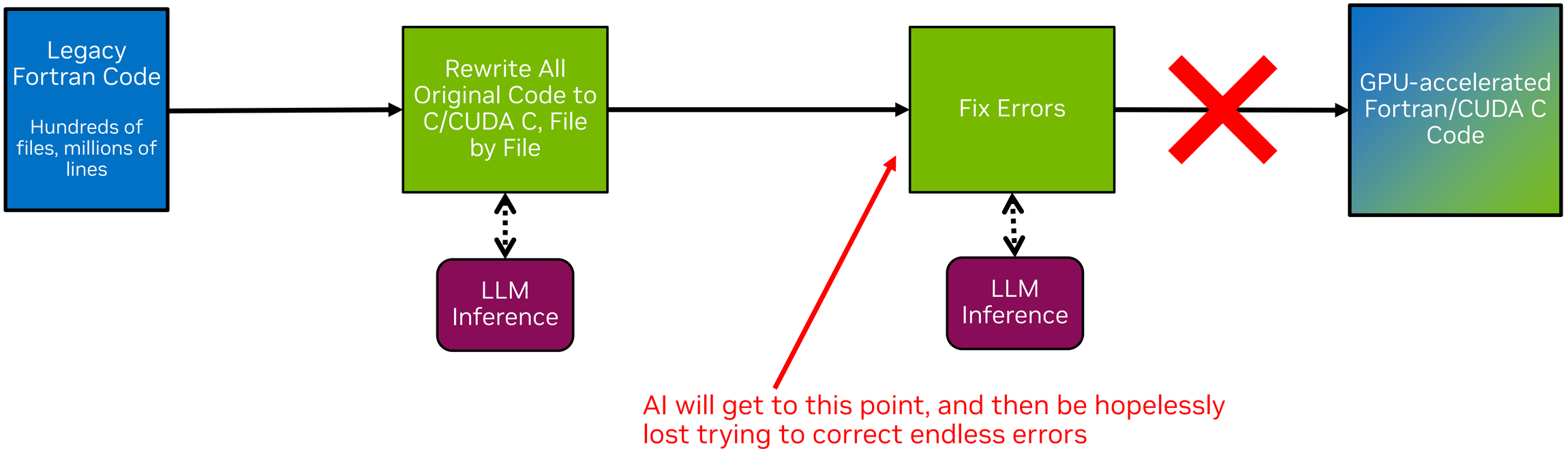


Two Kinds of AI Inference We Use

1. Generate new code in response to feature requests
2. Generate repairs to new code in response to errors

The Problem of Applying AI to Production Codebases

- Simplest approach: rewrite entire codebase file-by-file and then fix any errors

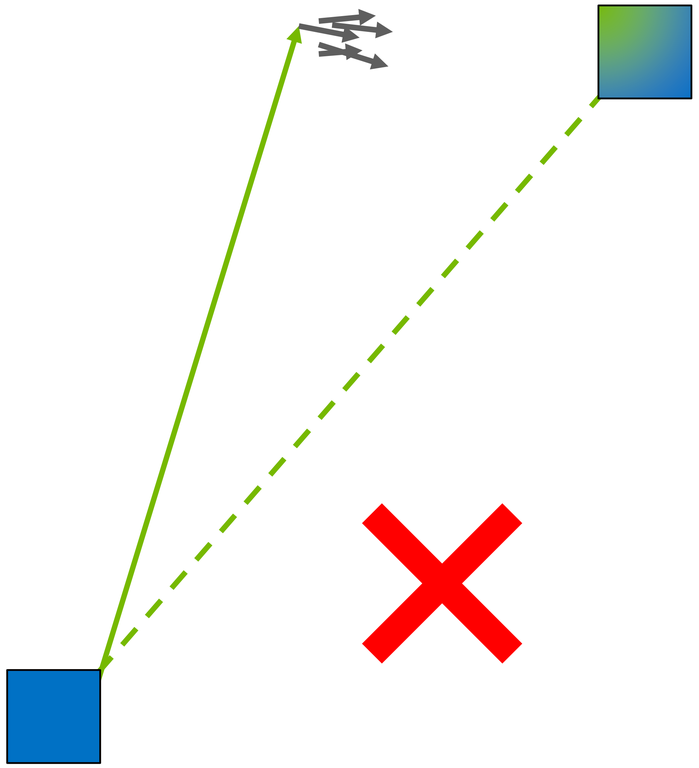


This is not what we do

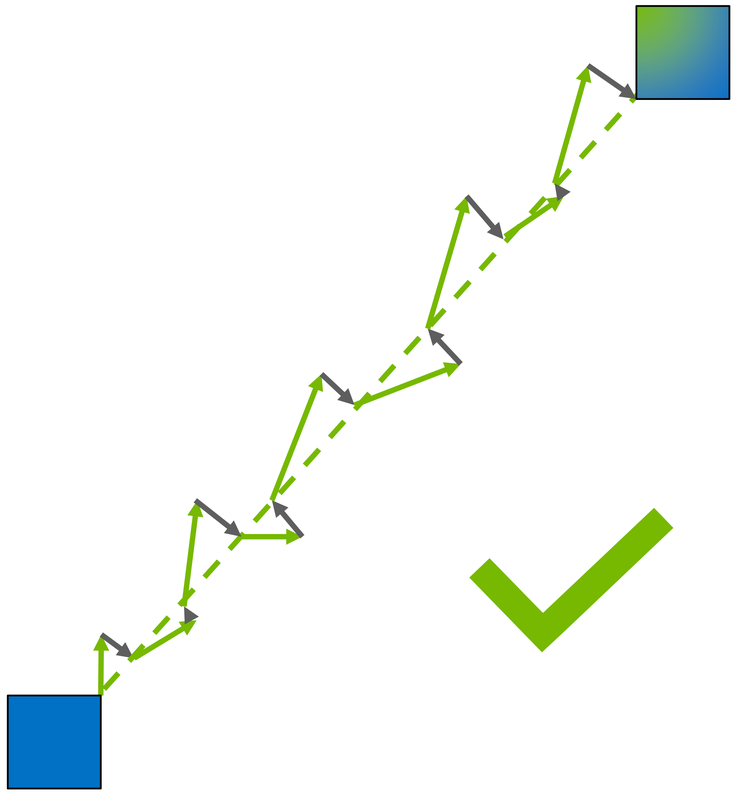
Addressing The Problem of Applying AI to Production Codebases

AI can only reliably repair “small enough” codebase changes

A big leap is unrecoverable



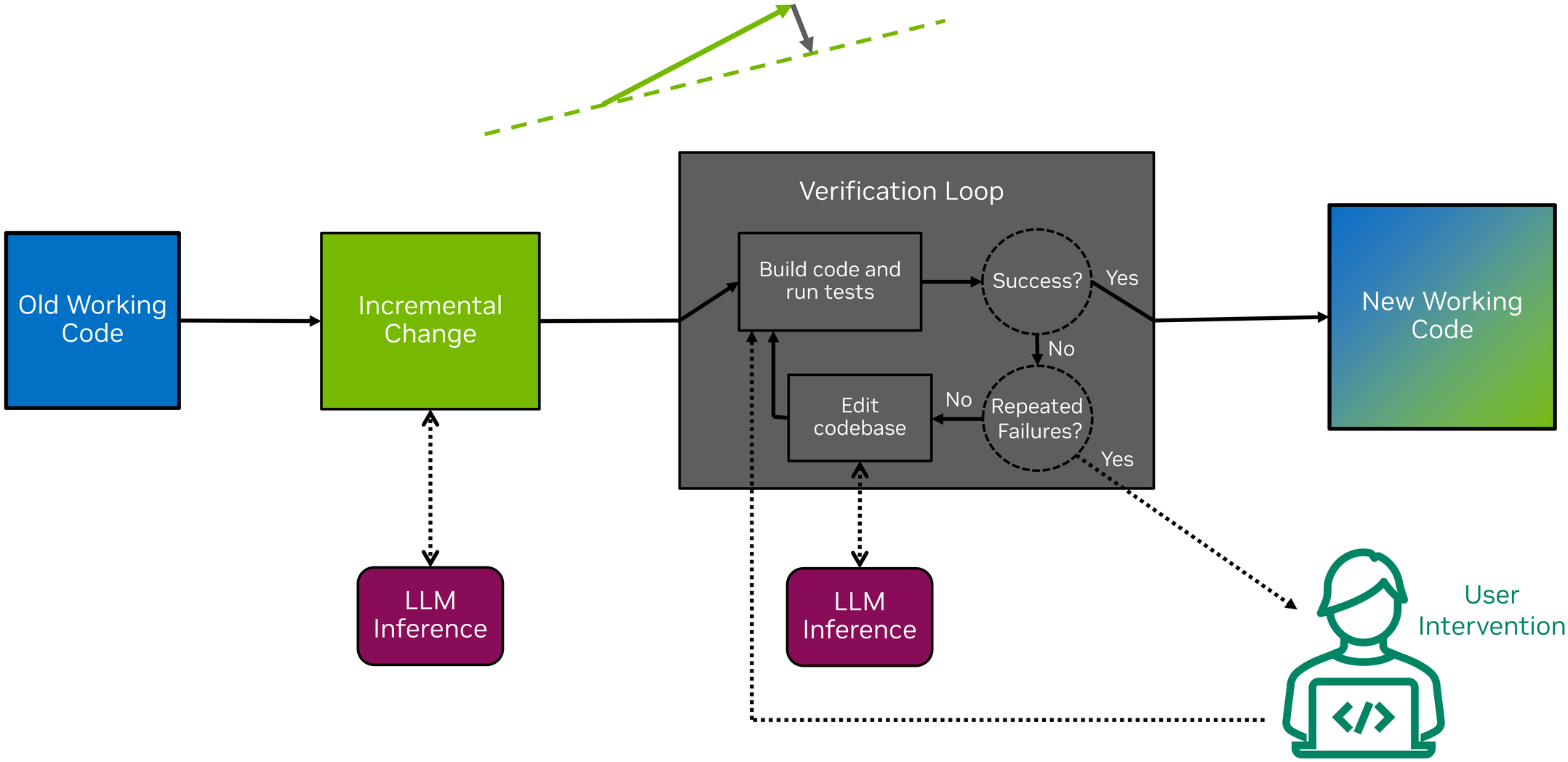
Sufficiently small steps are correctable



This is what we do

Our AI Engineering Building Block

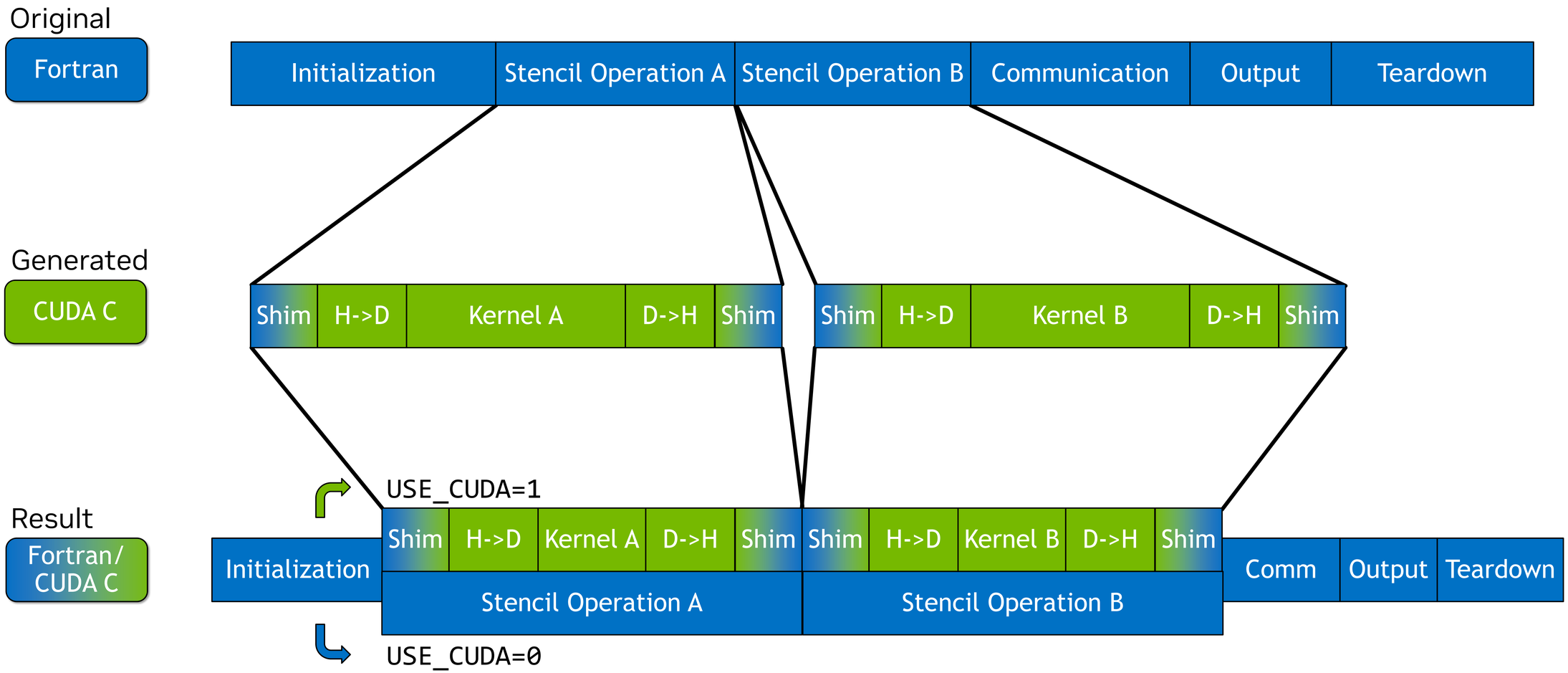
Minimal Unit To Elevate LLM Responses to Guaranteed Correctness



Our Region-Based Approach

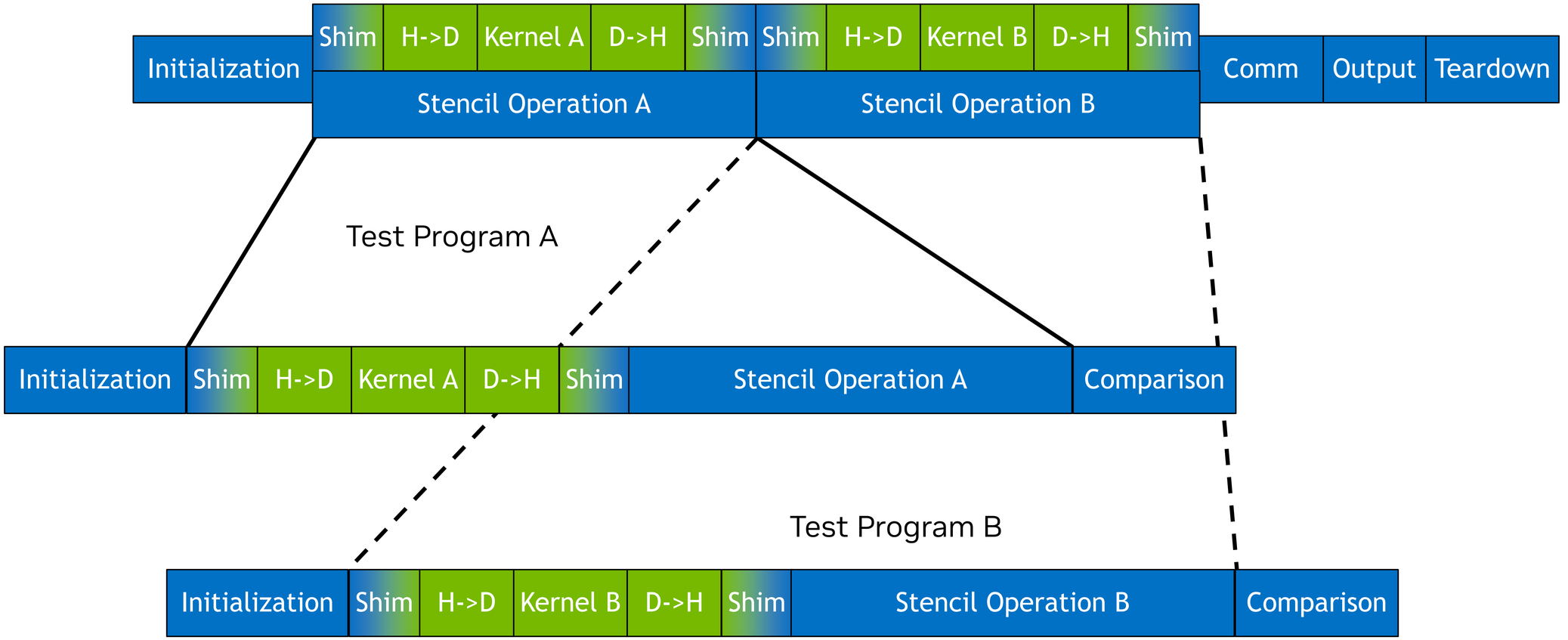
Our AI Agent Identifies and Ports Composable Regions of Code to GPU Language X

X = CUDA C in below example, but can also be OpenACC (Fortran) or Kokkos



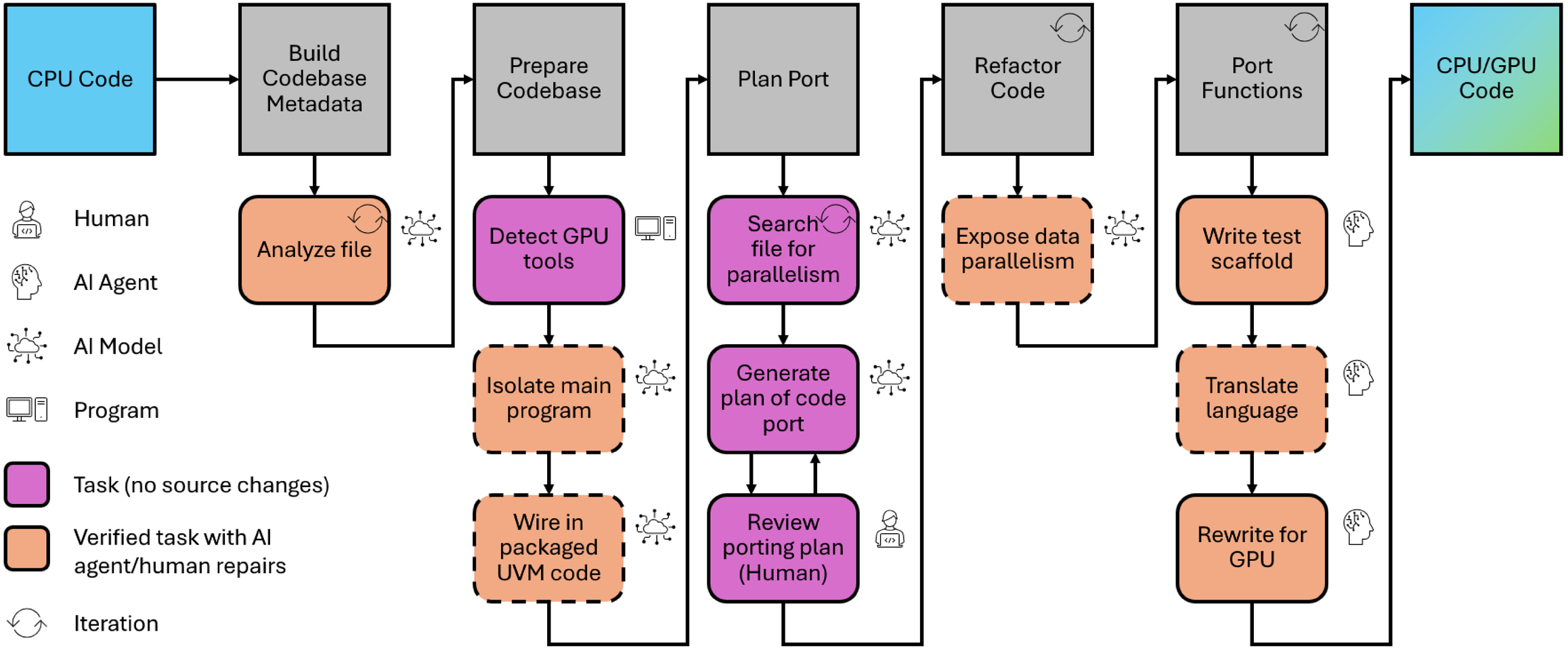
Our Testing Approach

Unit Testing, and Microbenchmarks



Our Agentic Workflow

Orchestration to scale manageable tasks to large codebases



Top-level Control

Deterministic or Agentic?

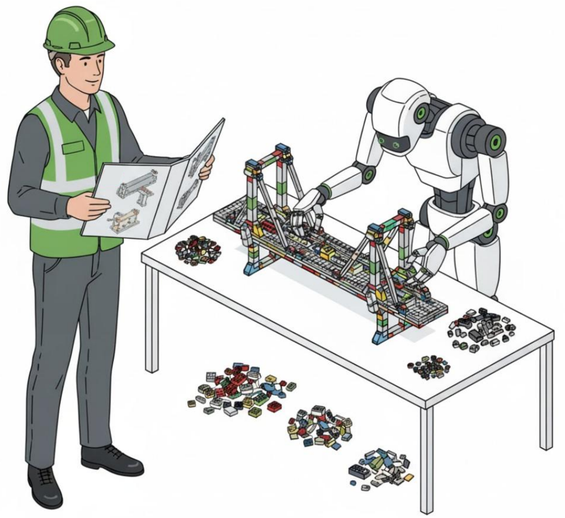
- Acceleragent was originally a CLI program written in Python that would call LLMs with a series of consistent prompts, extract data from JSON-formatted responses, and integrate into the code
  - Guaranteed to follow rules like testing modified code
  - Early 2025 LLMs could not be trusted to follow multistep processes
- This has been successful but two persistent issues
  - Long setup time getting the target code prepared just right for the Python logic
  - Fragile against unforeseen patterns common in real codes
- **May 2026, we built an Agent Skill version of Acceleragent**
  - Purely prompts and templates, to be given to a general coding agent (we use Claude Code)
  - It instruments the code and iteratively uses Nsight Systems to guide its efforts
  - This has shown great promise with latest frontier models like Claude Opus 4.8
  - **We are now pivoting to a SKILL-first approach**
    - We don’t have to re-implement an entire coding agent harness
    - We benefit from improvements to general coding agents

What Project Acceleragent Is

- A **workflow** for porting CPU codes to GPUs with AI
  - Step-by-step, single prompt-sized tasks
  - Correctness enforced
  - **Orchestration of AI responses**
- A tool to remove most of the effort of porting a code to GPUs
  - Designing a porting plan amenable to AI acceleration
  - Offloading as much rote work as possible
  - Creating tests, fixing bugs
  - Targeting 90% decrease in time to first cut solution
  - Expert can optimize for performance (kernel optimization, memory movement minimization)
- A proof of concept

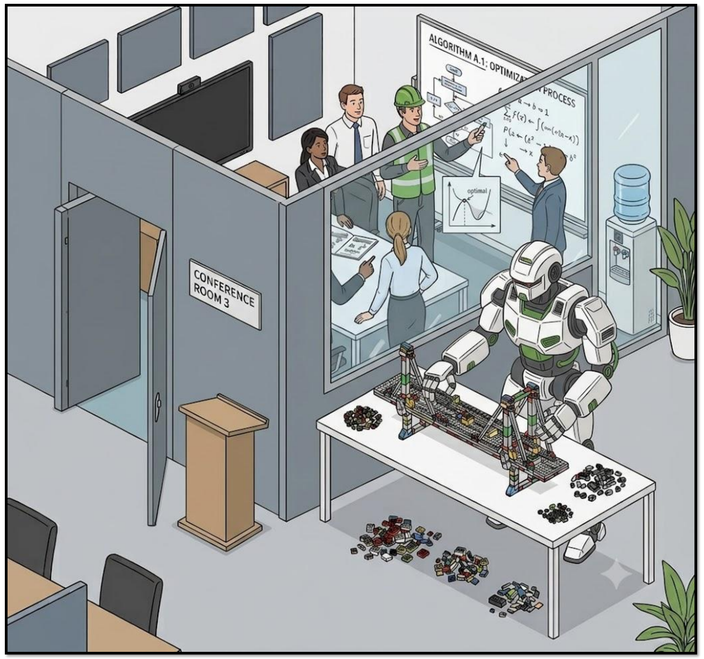
What Project Acceleragent Is Not

- A superhuman intelligence that accepts a codebase and returns the optimal GPU port with no human input
- A fine-tuned model; works with any available LLM accessible by API call, or coding agent
  - As models improve, so does Acceleragent
  - If you have a fine-tuned model, that model can be used by Acceleragent
- **A tool that can independently rearchitect an application or algorithm**
- A product



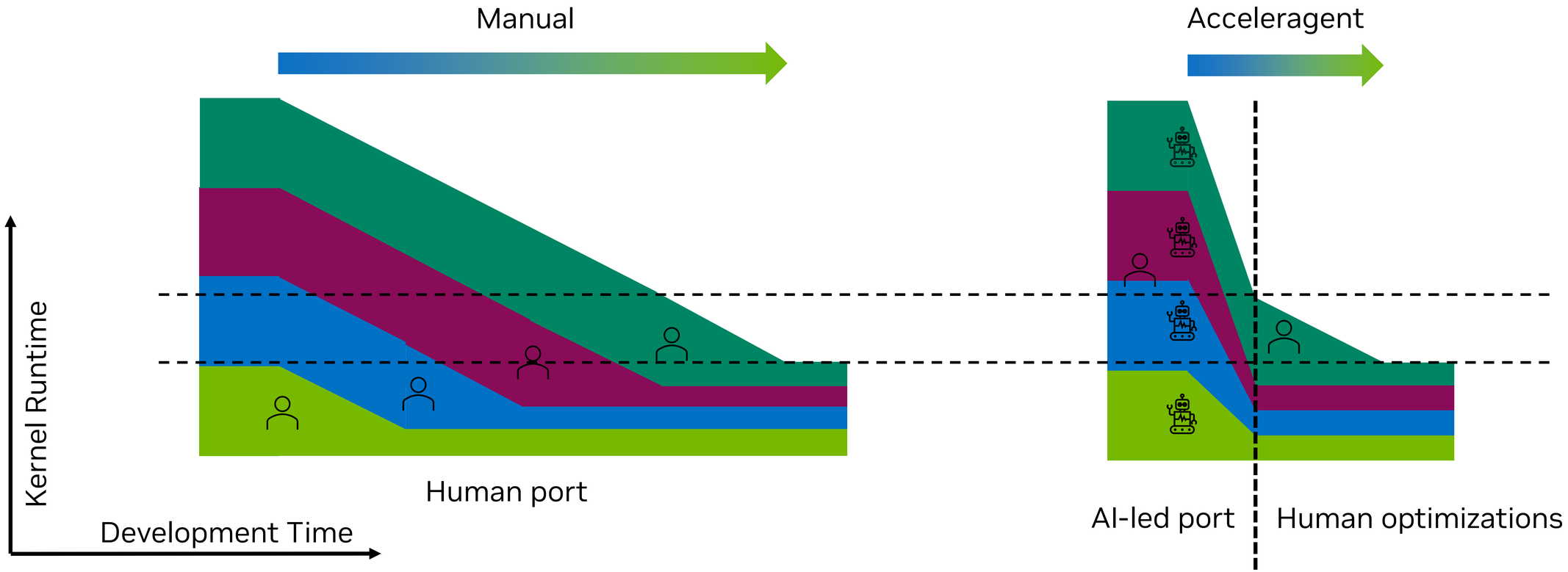
Owning and Specifying Correctness

- Human developers have complete knowledge of a project
  - Target problems, performance goals, stakeholder needs, likely future requirements
  - AI can only infer this from what it has e.g., the codebase
- Human involvement in defining correctness scripts used by AI to verify its changes is **crucial**
  - Anecdotally, AI hallucinations are particularly problematic for generating robust program-scale test scripts
    - If asked, AI will produce some test script. But maybe it just checks exit codes. Or global integrals.
- Project Acceleragent takes build and run verification scripts as inputs



Using Project Acceleragent

- Acceleragent organizes the work of a port into a proven, decomposable pattern
  - Individual prompts are aligned across the workflow
- Decreased time to solution for first pass of a full code path running on GPUs
  - Even if the original algorithm is not amenable to GPUs, this is often a useful starting point
- Human domain expertise remains important for this process
  - Reviewing plans, guiding repairs



Using Acceleragent

The Underlying LLM Matters

- We don’t require one shot LLM correctness for success
- More powerful LLMs driving Acceleragent require less human intervention
- Off the shelf models know enough about programming, GPU frameworks
- LLMs are rapidly improving (~6-month doubling timescale)

Smaller (~100B parameter) models, models without reasoning

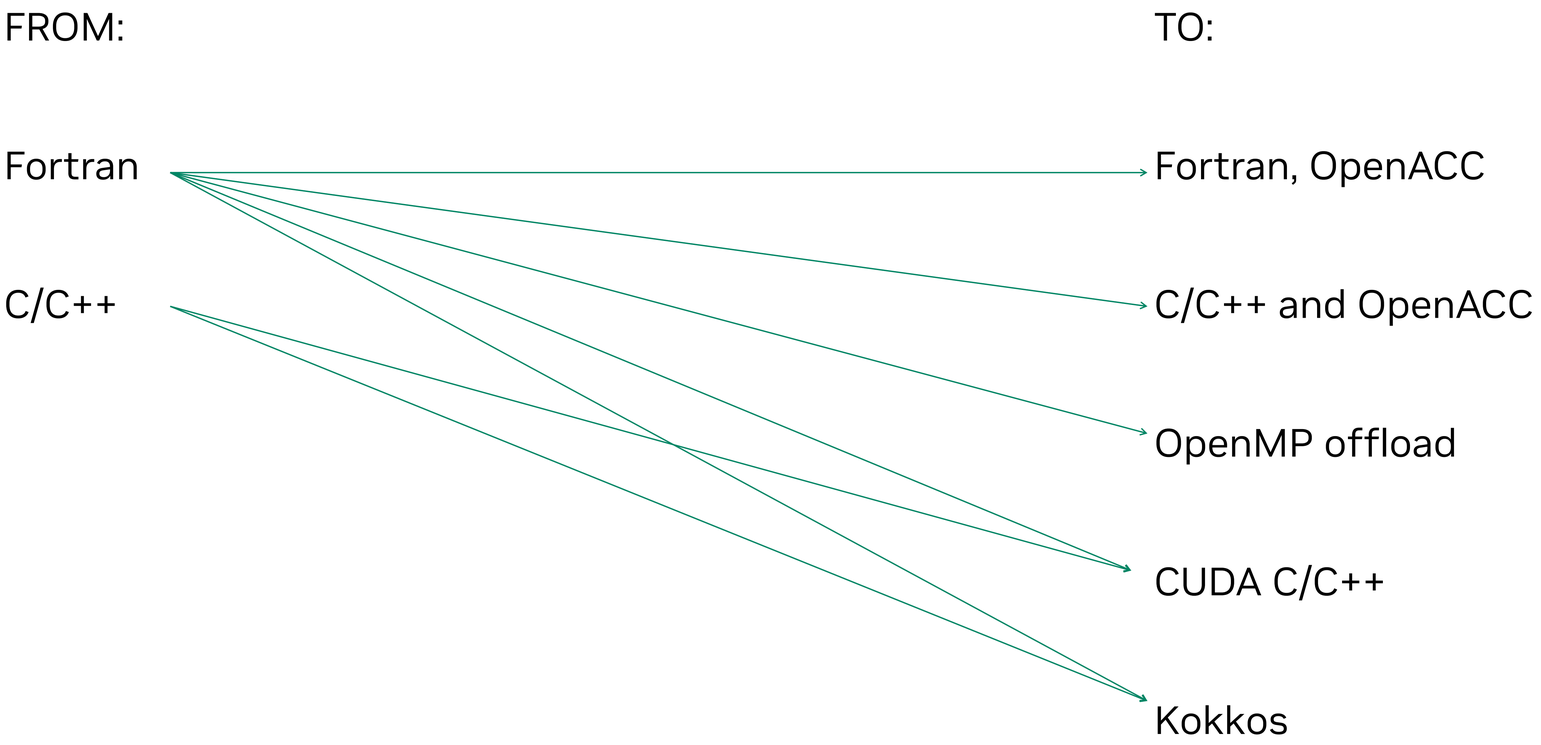
- ❌Generated code has many errors
  - Syntax errors
    - E.g., `#include` with no header file
  - Logic errors
    - E.g., incorrect multi-D array indexing
- ❌ LLM cannot repair its own generated code

Large models with reasoning (mainly closed source)

- ✅Generated code often high quality
- ✅Good knowledge of multiple GPU frameworks (CUDA, OpenACC,OpenMP, Kokkos)
- ⚠️ LLM can be expected to repair some but not all errors

Acceleragent Results

From X to Y



miniWeather Benchmark

2D weather hydrodynamics miniapp

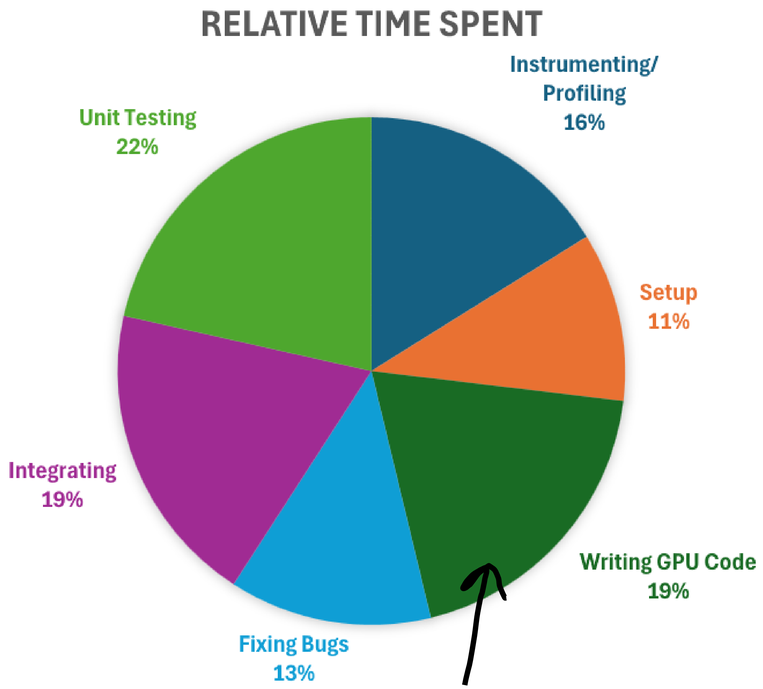
- I used Claude Code + Project Acceleragent Skill to port this benchmark to Kokkos
- I started from a version with only serial Fortran
- Then I opened Claude Code with autorun permissions and told it:
  - `Use the $acceleragent skill to port the code in this folder to Kokkos.`
- I answered initial interview questions:
  - `What orchestrator to use?` `Nsight Systems-based profiling`
  - `Scope of port?` `Entire codebase`
  - `How to manage GPU memory?` `Managed memory`
  - `What level of user engagement?` `Low`
- Then Claude Code ran with Claude Opus 4.8 for ~90 minutes…

miniWeather Benchmark

2D weather hydrodynamics miniapp

- Ported 6 functions from Fortran to C++, then C++ to Kokkos C++, created unit tests
- Refactored ~800 lines, generated ~2100 lines
- ~800M cache-read tokens, 800K output tokens, ~$110
- **2.6x speedup** on H100 GPU versus 72 Grace CPU cores (Fortran MPI + OpenMP version)

First pass



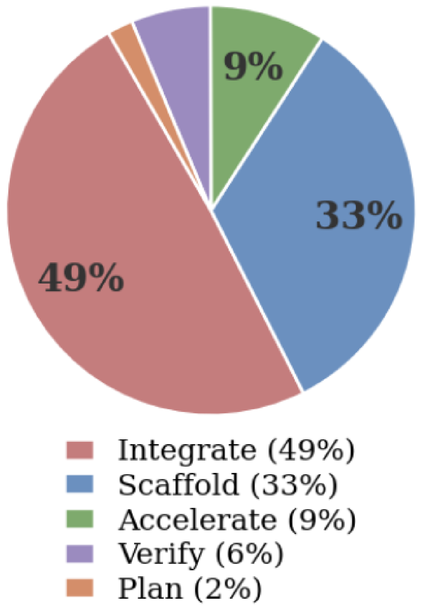
Writing GPU code is not most of the AI effort

MPAS-A

- Older port using Acceleragent CLI
- MPAS-A is a Fortran production code for climate simulation, about ~400k LOC
- Porting six key routines in dynamical core (~4k LOC) to Fortran OpenACC was entirely done by AI, ~25 million tokens
- Correctness verified by standalone tests, pointwise comparison
- On GH200, AI-ported GPU functions are ~few x slower than human ports in MPAS v8.3
  - Some unported kernels add memory motion costs

Codebase LOC	~400,000
Functions Translated	6
Source LOC (ported)	2,073
GPU Implementation LOC	2,105
Standalone Tests LOC	3,359
Input Tokens (million)	23.3
Output Tokens (million)	0.52
Agent Tool Calls	1,408
Repair Iterations	288

Tokens by Phase



Subroutine	AI GPU (s)	Human GPU (s)	AI Speedup
atm_compute_dyn_tend	10.4	1.9	0.2x
atm_compute_solve_diag.	1.9	1.2	0.6x
atm_advance_acoustic_step	2.3	1.3	0.6x
atm_recover_large_step	0.5	0.7	1.6x
atm_advance_scalars	1.1	0.1	0.1x
atm_compute_vert_imp_coefs	0.4	0.1	0.3x

Conclusions

- AI models and agents are powerful tools for code porting
- Connecting the AI to the large context and requirements of production codes is a key challenge
- Project Acceleragent goals:
  - Demonstrate and quantify value of AI applied to porting HPC codes to GPUs
  - Identify and share best practices for AI GPU porting
- First cut end-to-end GPU ports appear to allow substantial developer speedups
  - Most of the work is not writing kernels
- Optimized GPU ports still need additional effort
  - Traditional performance optimization, algorithmic changes that affect results