# W&M collaboration opportunities

## PHASM: Parallel Hardware viA Surrogate Models

Nathan Brei (nbrei@jlab.org)
Xinxin "Cissy" Mei
Kishan Rajput
David Lawrence
Malachi Schram

Wednesday, August 24, 2022

EPSCI

Jefferson Lab

U.S. DEPARTMENT OF ENERGY | Office of Science

JSA

# PHASM project

- *Parallel Hardware viA Surrogate Models*
- *Running Legacy Code on Heterogeneous Hardware via Surrogate Models*
- EPSCI group: Experimental Physics Software and Computing Infrastructure
- LDRD project (~1 year old, ~2 FTE's)

### Basic idea

Make it easy to train a neural net *surrogate model* to mimic and replace an arbitrary piece of existing, complicated code. Systematize and formalize the process from analysis to deployment.

### Short term goal

Create an on-ramp for getting ML research models into production, apply industry best practices, and tighten the design feedback loop.

### Long term goal

Create a tool analogous to a debugger, which lets a user interactively experiment with introducing AI surrogate models into a legacy scientific codebase *possibly without recompiling*. Explore the limits of AI/ML surrogate models both quantitatively and intuitively.

# Background and motivation

A neural net trained to mimic an existing numerical algorithm is just another numerical algorithm. However:

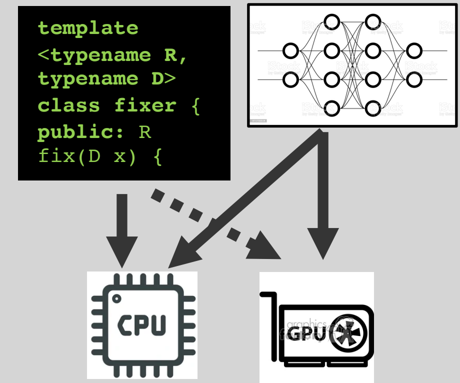
- It has <u>different time and space complexity</u>. This gives us fast simulations and inverse simulations.

- Unlike other modelling techniques, neural nets can <u>sidestep the curse of dimensionality</u>. This makes surrogate models feasible for a much wider class of algorithms than in the past.

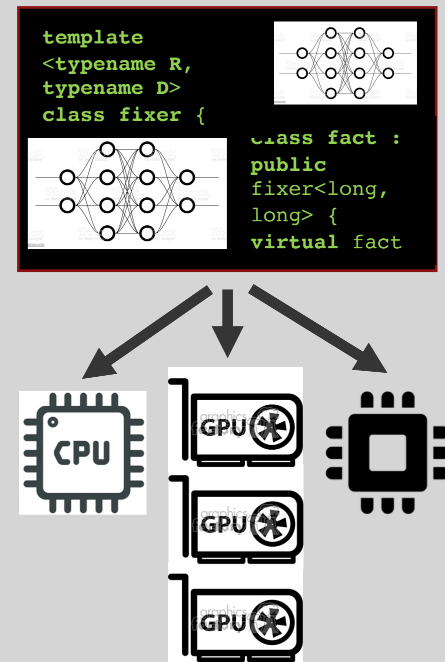
- They can have much <u>higher internal parallelism</u> than the original algorithms. This lets them run efficiently on heterogeneous hardware such as GPUs, TPUs, and FPGAs. The transformation is usually automatic.

Modern data centers are investing heavily in heterogeneous hardware, but many important codebases are unable to take advantage of it without undergoing a costly rewrite. AI surrogate models could mitigate this.

## Present situation




## Future situation



# PHASM project deliverables

1. **Surrogate API**

   Connects legacy C++ code to a machine learning model in a clean, abstract way. Focus on data binding; delegate to best-in-class tools when possible

2. **Model variable discovery tool**

   Traces a function's memory accesses to identify all inputs and outputs, not just those in the function's type signature

3. **Performance analysis playbook**

   Allows researchers to qualitatively predict the performance of a surrogate model (particularly on heterogeneous hardware) *before* designing and training it

# Surrogate API

**Idea**

- Inject a machine learning model into the middle of a legacy codebase without rearchitecting the surrounding code (Stretch goal: do this at runtime)
- Provide a high-level interface for specifying the input variables, bounds, and model parameters.
- Abstract away the work of integrating a ML framework, capturing training samples, choosing additional training samples, training the model, loading and storing the trained model, and switching between the original function and the surrogate.

**Current status**

- Uses *profunctor optics,* a concept from functional programming, to perform simple and effective two-way data transformation between arbitrary C++ types and tensors. Currently supports primitives, arrays, structures, and unions thereof. Correctly handles nested datatypes, e.g. an array containing a struct containing an array of doubles translates to a 2D tensor of doubles (without writing any loops!).

- Uses PyTorch+TorchScript as a backend.

**Next steps:**

- Dynamically loaded ML backends

- Integration with MLFlow for model lifecycle management

- Generating synthetic training data. Interesting links to reinforcement learning, fuzzing

# Surrogate API

```cpp
double f(double x, double y, double z) {
    return 3*x*x + 2*y + z;
}

// We create a surrogate model for f like so:

phasm::Surrogate f_surrogate = phasm::SurrogateBuilder()
        .set_model(std::make_shared<phasm::FeedForwardModel>())
        .local_primitive<double>("x", phasm::IN)
        .local_primitive<double>("y", phasm::IN)
        .local_primitive<double>("z", phasm::IN)
        .local_primitive<double>("returns", phasm::OUT)
        .finish();

double f_wrapper(double x, double y, double z) {
    double result = 0.0;
    f_surrogate.bind_original_function([&](){ result = f(x,y,z); })
                .bind_all_callsite_vars(&x, &y, &z)
                .call();
    return result;
}
```

# Model variable discovery tool

**Idea**

- Given a function, identify *all* of its inputs and outputs (global variables, nested structures, pointers, etc)
- Report the complexity of the input space and the true memory movement needed
- Generate code that repackages them into tensors for the machine learning model

**Theory**

- Turns an impure function of rich nested datatypes into a pure function of tensors (when possible)
- Opens the door for property-based testing, numerical sensitivity and stability analyses, etc

**Current status**

- Performs a *dynamic binary analysis* based on Intel PIN. We instrument a compiled binary to intercept and log memory accesses and allocations, analogous to valgrind memcheck.
- Uses DWARF debugging data to resolve variable names and types. Currently supports primitives, arrays, and structures/classes. Support for nested types underway.

**Next steps**

- Improve the tool's reliability
- Experiment with performing a static analysis instead, based on either LLVM/Clang or ROSE

# Performance analysis playbook

**Idea:** Qualitatively predict, for an arbitrary function, whether and when a surrogate model would run effectively on heterogeneous hardware.

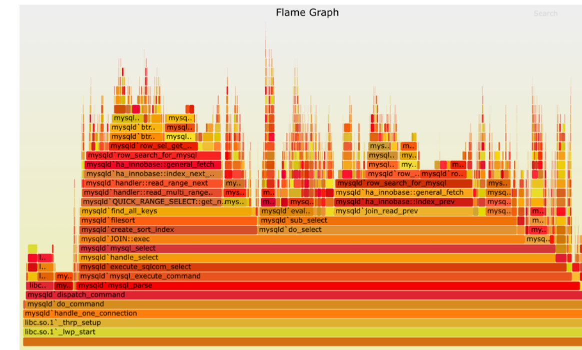
- Systematize the analysis procedure into a step-by-step playbook. Which tools, which metrics?
- Make it realistic for researchers to do the full analysis before writing code
- The goal is hardware utilization and/or throughput, not strictly a speedup
- Match theoretical predictions (e.g. Amdahl, Gustavsson, Brent, Roofline) against empirical data

## Current status

- Performed roofline analyses for various compute kernels and hardware using Intel Advisor, likwid, nvprof
- Developing a decision criterion, expressed as a mathematical formula, spiritually similar to Amdahl's Law but specialized for our use case.

## Next steps

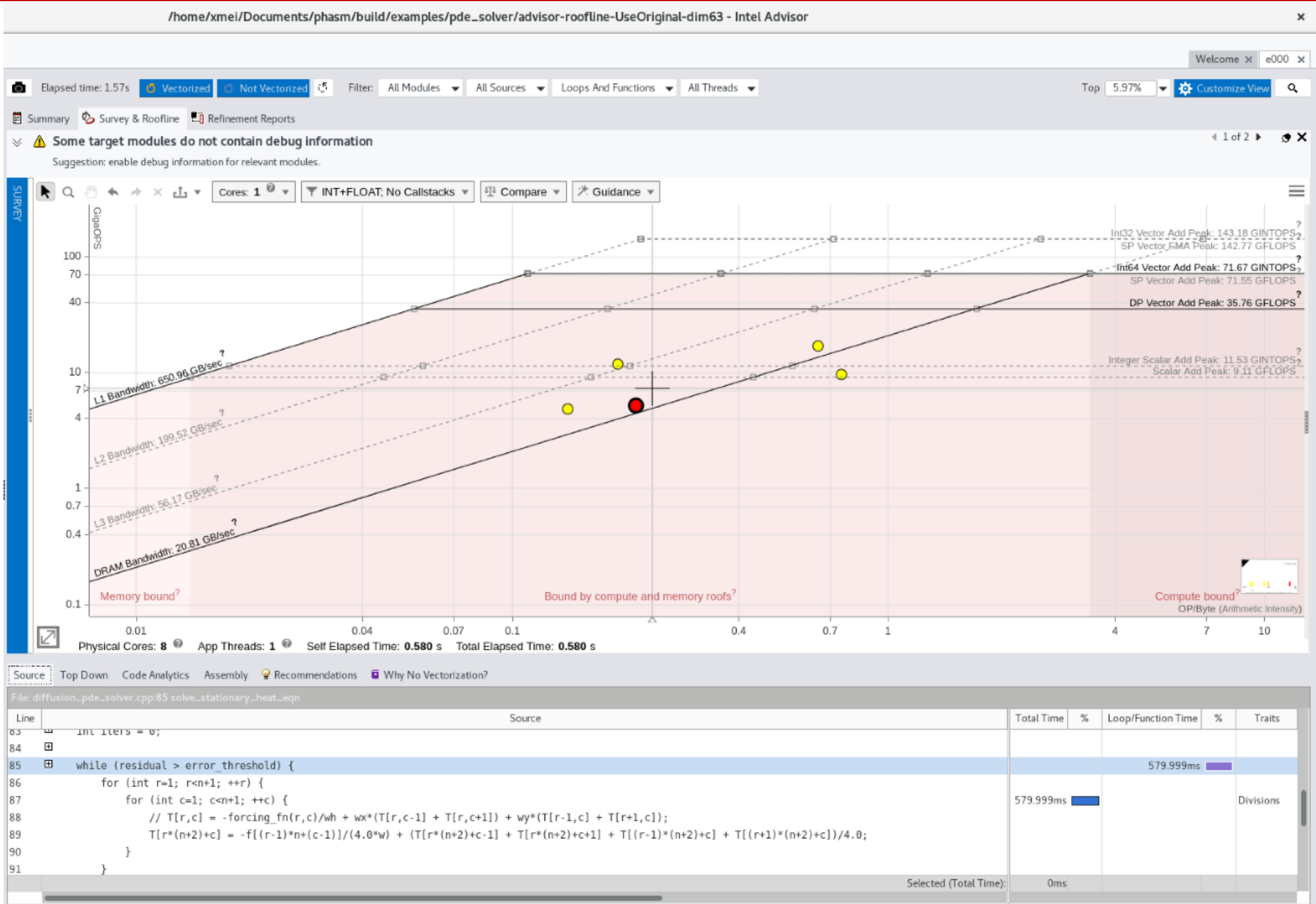
- Automatically profile a program and traverse its flame graph to identify kernels that could be promising candidates for surrogate models
- Develop an intuition and back it up with empirical data






Jefferson Lab

# Performance analysis example



# Thank you!

Questions?