PHASM: Parallel Hardware viA Surrogate Models

ACAT 2022

Nathan Brei, Xinxin Mei, David Lawrence

Jefferson Lab

October 26, 2022

What is PHASM?

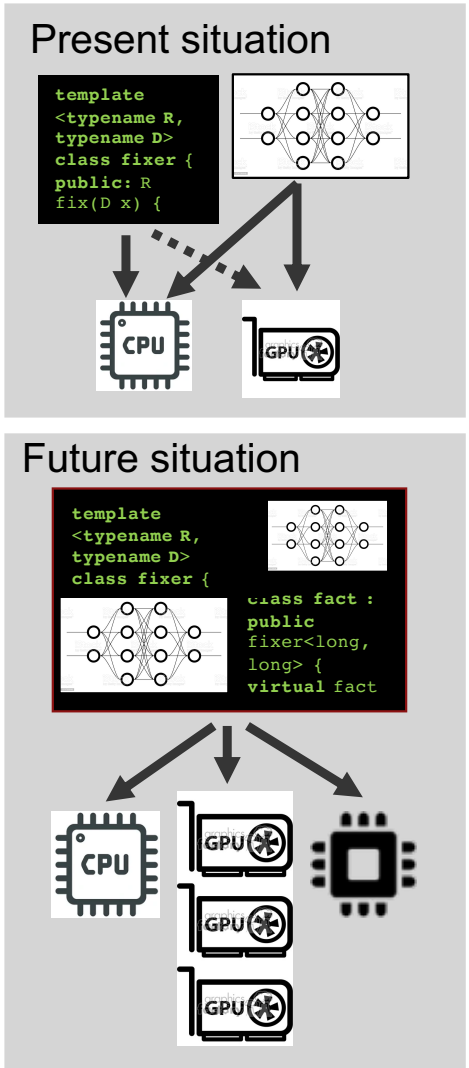
- LDRD project at Jefferson Lab
- 1 year old, 2-3 people
- Proof of concept

Basic Idea

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

Perspective shift

A neural net surrogate model of an algorithm is a *transformation* of that algorithm. Eventually, classical numerical methods and their data-driven analogues will be understood under a unified theory.



Advantages

- Different time and space complexity: fast simulations, triggers
- Invertible: detector optimization
- Inherent parallelism: offloading onto a GPU or FPGA
- Sidesteps curse of dimensionality: large input/output spaces
- Access to full 'truth': can generate training data as needed

Challenges

- Model accuracy and uncertainty quantification
- Drifting input distributions
- Deciding the boundaries of what gets modelled

'Killer app'

Getting legacy codebases to run effectively on heterogeneous hardware without rewriting everything in CUDA or SYCL

Strategic goals

1. Create an interactive tool for designing and testing surrogate models empirically and rapidly, without recompiling. Analogous to a debugger: Pause execution of the program and present the user with options such as: profile a function, rewrite the binary to replace a function with a surrogate, tune the surrogate's hyperparameters, capture training data, etc.
2. Pieces of this tool should be usable on their own and should help productionize ML models around the lab. We want to make it easy to integrate best-in-class ML tools into established codebases.
3. This opens up new research opportunities into mixing neural nets with numerical methods, and embedding domain knowledge and UQ. Example: Synthetic generation of training data using static analysis/fuzzing.

Surrogate API

- Integrates best-in-class ML tools into a legacy codebase in a highly abstract way (PyTorch, soon also MLFlow, Julia)

Model variable discovery tool

- Automatically identifies, for a given function, the true space of inputs and outputs including those not apparent from the type signature. Generates bindings from an impure function of (possibly nested) C++ types to a pure function of tensors.

Performance analysis playbook

- Systematically determines whether it makes sense to apply a surrogate model to a given function before going further. This will be extended to automatically identify compute kernels that might make good surrogacy candidates.

Exposes

- Model variables, including name, shape, type, and bounds
- Model hyperparameters (coming soon!)
- Generating training samples (coming soon!)

Abstracts away

- The details of the model design (delegated to PyTorch et al)
- Capturing training samples
- Switching between the original function and the surrogate
- Model lifecycle management (delegated to MLFlow) (coming soon!)
- Linking the ML framework (coming soon!)

Surrogate API: Simple example

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

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>("retval", phasm::OUT)
        .finish();

double f_wrapper(double x, double y, double z) {
    double res = 0.0;
    f_surrogate.bind_original_function([&](){res = f(x,y,z);})
               .bind_all_callsite_vars(&x, &y, &z)
               .call();  // $PHASM_CALL_MODE controls this
}
```

Optics

- *Profunctor optics* are used to perform simple and effective two-way data transformation between C++ types and tensors.
- Profunctor optics are composable 'getter and setter' objects that are generalized to support type conversions, missing, and multiple values.
- 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!
- Currently PHASM supports primitives, fixed-length arrays, variable-length arrays, structures, and unions.
- In progress are STL collections and objects with invariants

Surrogate API: Nested data example

```
struct MyStruct { double x, y; };
double sum = 0;
void sum_all_x(MyStruct* s) {
    for (int i=0; i<4; ++i) { sum += s[i]->x; }
}
phasm::Surrogate sum_all_x_surrogate = phasm::SurrogateBuilder()
    .set_model(std::make_shared<phasm::TorchScriptModel>("model.pt"))
    .local<MyStruct*>("s")
        .array<MyStruct>(4)
        .accessor<double>([](MyStruct *s) { return &(s->x); })
        .primitive<double>("x", phasm::IN)
        .end()
    .global_primitive<int>("z", &z, phasm::INOUT)
    .finish();

void sum_all_x_wrapper(MyStruct* s) {
    sum_all_x_surrogate.bind_original_function([&](){f(s);})
                .bind_all_callsite_vars(s)
                .call();  // $PHASM_CALL_MODE controls this
}
```

Model variable discovery tool

- A key obstacle to rapid development of surrogate models is that scientific code is almost never written as pure functions with primitive arguments.
- We need a tool that takes a function and identifies all of its inputs and outputs (global variables, nested structures, pointers, etc).
- For each input and output it reports the shape, size, and type of the memory being moved, and generates optics to convert the data to and from tensors.
- This will probably never work in the fully automated case (consider IO streams, RNGs, cyclic graphs), but it can always fall back to directing the user to the offending line of code and asking for manual intervention.
- It has applications far beyond surrogate models: property-based testing, numerical sensitivity/stability analyses, etc.

Current status

The proof-of-concept traces memory movement using Intel PIN to perform a dynamic binary analysis, similar in spirit to Valgrind memcheck. Types and symbols are recovered by traversing the DWARF debugging data. This works somewhat, but has a number of problems in practice:

- Inability to 'look' down branches not taken
- Fragility of DWARF data even with optimization turned off
- Recovering the symbol and type information is computationally intensive

Next steps

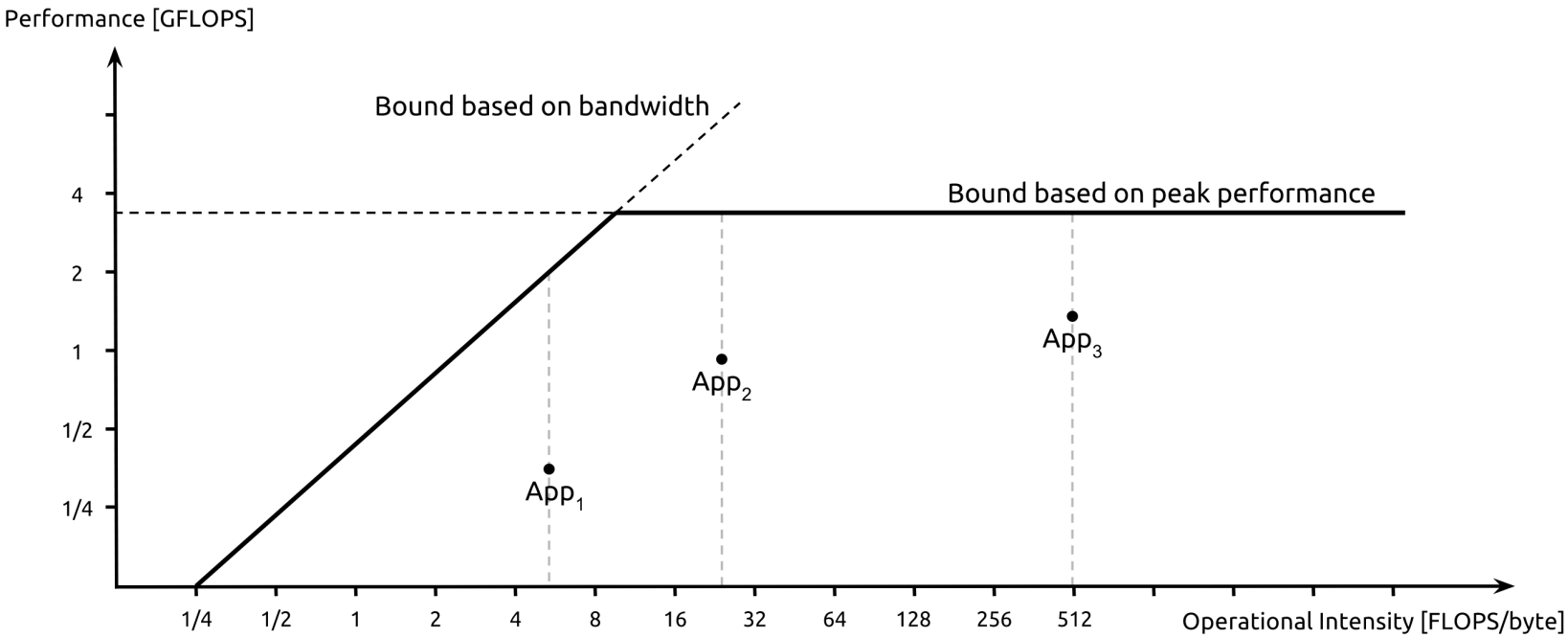
- Use either clang/llvm or ROSE to perform a static analysis instead, with full access to the AST.

Idea

Qualitatively predict, for an arbitrary function, whether and when a surrogate model would run effectively on some hardware.

- Systematize the analysis procedure into a step-by-step playbook. Which tools, which metrics, which settings?
- Make it realistic for researchers to do the analysis before writing code
- The goal is to increase hardware utilization and/or throughput, not necessarily obtain a speedup
- Account for hardware-specific memory bottlenecks and model-specific compute intensity
- Match theoretical predictions (e.g. Amdahl, Gustavsson, Roofline) against empirical data

Performance analysis: Roofline model



Thank you!

Any questions?