PHASM: A toolkit for creating AI surrogate models within legacy codebases

Nathan Brei, Xinxin Mei, Kishansingh Rajput, David Lawrence
Thomas Jefferson National Laboratory, 12000 Jefferson Avenue, Newport News, VA, 23606
E-mail: nbrei@jlab.org

Abstract. PHASM ("Parallel Hardware viA Surrogate Models") is a software toolkit currently under development for creating AI-based surrogate models of scientific code. AI-based surrogate models are widely used for creating fast and inverse simulations. The project anticipates an additional future use case: adapting legacy code to modern hardware. Data centers are investing in heterogeneous hardware such as GPUs and FPGAs. Meanwhile, many important codebases are unable to take advantage of this hardware's superior parallelism without undergoing a costly rewrite. An alternative is to train a neural net surrogate model to mimic the computationally intensive functions in the code, and deploy the surrogate on the exotic hardware instead. PHASM addresses three specific challenges: (1) systematically discovering which functions can be effectively replaced with a surrogate, (2) automatically identifying, for a given function, the true space of inputs and outputs including those not apparent from the type signature, and (3) integrating a machine learning model into a legacy codebase cleanly and with a high level of abstraction. In the first year of development, a proof of concept has been developed for each challenge. A surrogate API makes it easy to bring PyTorch models into the C++ ecosystem and use profunctor optics to establish a two-way data binding between C++ datatypes and tensors. A model variable discovery tool performs a dynamic binary analysis using Intel PIN in order to identify a target function's model variable space, including types, shapes, and ranges, and generate the optics code necessary to bind the model to the function. Future work may include exploring the limits of surrogate models for functions of increasing size and complexity, and adaptively generating synthetic training data based on uncertainty estimates.

1. Background

A neural net trained to mimic an existing numerical algorithm is fundamentally just another numerical algorithm. However, this class of algorithms has advantageous properties. Firstly, it might have different time and space complexity, which prove immediately useful for fast and inverse simulations. Secondly, in contrast to other modelling techniques, neural nets can act as universal function approximators while sidestepping the curse of dimensionality [1]. This makes surrogate models feasible for functions with much higher-dimensional input and output spaces. Thirdly, a neural net surrogate model might have a much higher level of internal parallelism compared to the original algorithm [2]. This permits them to run efficiently on heterogeneous hardware such as GPUs, TPUs, or FPGAs. Once designed and trained, it is straightforward to deploy a model on different hardware types. Modern data centers are investing heavily in heterogeneous hardware, but many important codebases are unable to take advantage of this hardware without undergoing a costly rewrite. An alternative is to train a neural net surrogate

model to mimic the computationally intensive functions in the code, and deploy the surrogate on the exotic hardware instead.

While promising, there are several challenges with the general surrogate modelling approach. Obtaining an acceptable level of accuracy is essential, as is having reliable uncertainty quantification. Another consideration is that the input distributions are liable to drift over time as a model is run in production, so that model versioning and continuous re-training are necessary. Identifying the scope of a model can also be challenging in a real-world codebase, due to layers of abstraction, a lack of separation of concerns, use of highly stateful software patterns, and the organizing of data into object hierarchies. Finally, there are practical difficulties linking modern machine learning tools into a legacy codebase when both require highly customized (yet very different) build systems and environments.

2. Project goals and deliverables

PHASM (Parallel Hardware viA Surrogate Models) is a toolkit for creating AI surrogate models within legacy scientific codebases. It is a one-year-old LDRD project at Jefferson Lab. PHASM's overarching goal is to systematize and formalize the process of developing AI/ML surrogate models. It creates a smoother on-ramp for designing, testing, and bringing ML research models into production. It aims to bridge the gap between the legacy code and the best-in-class industry frameworks, which enforces best practices and tightens the design feedback loop. It introduces new tools to address specific pain points that arise with legacy code, such as predicting the surrogate model's performance upfront, identifying the model variables, converting arbitrary datatypes to and from tensors, and linking in a machine learning framework. If the field of scientific machine learning (SciML) continues to accelerate, these tools could prove particularly useful for parallelizing legacy codebases.

There are three main project deliverables: a performance analysis 'playbook', a model variable discovery tool, and an API for injecting surrogate models into a legacy codebase. The playbook tackles the question of whether a model makes sense to begin with: it provides instructions for obtaining and interpreting hardware benchmarks and profiler data to predict what bottlenecks the model will experience, and what overall performance will result. The model variable discovery tool analyzes the binary to figure out the true inputs and outputs for a given target function, returning a description of the data along with code for accessing it. Finally, the surrogate API connects legacy code to a machine learning model in a clean, abstract way, with a particular focus on the bidirectional data bindings between C/C++ datatypes and tensors. It enables the user to wrap and intercept a target function, so that they can decide out-of-band (a) whether to use the surrogate model or original function, and (b) whether to capture the inputs and outputs and dump them to file, or use them for continuous training. These deliverables shall be discussed in turn.

3. Performance analysis playbook

The performance analysis playbook lets the user predict, for an arbitrary target function, whether a surrogate model would run effectively on the hardware. The playbook systematizes the analysis procedure into an explicit checklist that details tools to use, settings to configure, and metrics to take into account. The analysis is currently scoped to event-level parallelism and high-throughput computing (HTC), as opposed to a spatial domain decomposition and high-performance computing (HPC). It assumes the goal is to increase throughput and hardware utilization, rather than to decrease latency.

There are several steps in this analysis. First is a benchmarking step to characterize the hardware bottlenecks as per the Roofline model [3]. Next is profiling, to measure the compute intensity and memory movement of the target function. Following that is predicting the compute intensity of the machine learning model, which, together with the memory movement, indicates

which bottleneck will be active. The active bottleneck in turn determines the maximum performance of the model on the hardware. The change in latency and throughput for each event can then be predicted via Amdahl's and Gustavsson's laws.

Future work includes empirically validating the accuracy of this performance prediction on real-world problems, and on identifying opportune target functions by examining a profiler's flame graph [4].

4. Model variable discovery tool

The model variable discovery tool (MVDT) automatically identifies, for a given function, the true space of inputs and outputs. These are not obvious from the type signature due to complications such as nested structures, pointers, and global variables. The model variable discovery tool traces all memory operations within the target function and maps out the essential data. It generates code for data binding, which is used by the surrogate API. It also reports the size and shape of the input and output tensors, which factors into the performance analysis. A further goal is to identify impure constructs that are particularly difficult to finesse away, such as I/O streams and random number generators. These features help the user eliminate bad candidates for surrogate models before going through the effort of designing and training a model. Since the MVDT is meant to assist a human who would otherwise have to manually comb through a large amount of unfamiliar code and write the bindings by hand, it is permissible for its analysis to be incomplete as long as it clearly reports which code regions it couldn't understand.

The current proof-of-concept tool traces memory movement using Intel PIN [5] to perform a dynamic binary analysis, similar in spirit to Valgrind memcheck. PIN intercepts execution of the binary and adds instrumentation code to track memory operations. The types and symbols corresponding to each address are recovered from the DWARF debugging information. Although versatile, this design has a number of practical limitations. Firstly, it is unable to observe branches until the program execution reaches them. This means that many executions are needed, and some branches may never be reached. Secondly, the DWARF data is complex and fragile, even with optimization turned off. Finally, even with perfect DWARF data, recovering the symbol and type information for a given address is computationally intensive.

The next step is to try a static analysis using clang instead, which should address all of the above issues [6]. A static analysis can observe all execution paths at once, and because the clang static analyzer has full access to the program's abstract syntax tree, it can retrieve the desired symbolic information cheaply. In the long term, if the model variable discovery tool becomes reliable enough, it would make it possible to use PHASM's surrogate API to intercept arbitrary functions at runtime, without having to recompile.

5. Surrogate API

The key goal for the surrogate API is to insert a machine learning model into the middle of a legacy codebase without rearchitecting the surrounding code. It provides a high-level interface for specifying the model, its hyperparameters, its input variables, and their bounds. Meanwhile it abstracts away the work of integrating the machine learning 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.

Figure 1 shows the most basic usage of PHASM. The legacy codebase implements a function `f`, which accepts two doubles and returns a double. This function is ideal for surrogate modelling because it is pure, continuous, differentiable, and has low dimensional inputs and outputs. The first step is to hide this function from its callees by wrapping it in a namespace (1). Next we define a surrogate model via the builder interface (2). In the general case this should be defined statically, because the original function has static duration. In (3), we declare that we will be using a simple feedforward network with 1 hidden layer of 10 nodes. Next we declare the model's

```
namespace wrapped {                                                    // (1)
double f(double x, double y) {
    return 3*x*x + 2*y;
}}

auto f_surrogate = phasm::SurrogateBuilder()                           // (2)
    .set_model(std::make_shared<phasm::FeedForwardModel>(1,10))        // (3)
    .local_primitive<double>("x", phasm::IN)                           // (4)
    .local_primitive<double>("y", phasm::IN)
    .local_primitive<double>("z", phasm::OUT)
    .finish();

double f(double x, double y) {                                         // (5)
    double z = 0.0;
    f_surrogate
        .bind_original_function([&](){z = wrapped::f(x,y);})           // (6)
        .bind_all_callsite_vars(&x, &y, &z)                            // (7)
        .call();                                                       // (8)
    return z;
}
```

Figure 1. The simplest usage of the PHASM surrogate API

inputs and outputs (4). Note that PHASM's model abstraction supports adding adapter layers so that a generic foundation model can be matched with the precise input and output shapes, if necessary.

We now define the wrapper function `f`, with the exact same signature as its wrapped predecessor (5). We set up two bindings: Firstly, so that the surrogate can call the original function, we pass it a zero-argument lambda in (6) that captures the local callsite variables `x,y,z`. Secondly, so that the surrogate can transform the local callsite variables into tensors, we pass it pointers to the local callsite variables in the same order they were declared. Finally, we call the surrogate model via `call` (8). This may call the original function and capture training data, or it may call the model directly, or it may train the model – its behavior is configured out-of-band via the `PHASM_CALL_MODE` environment variable.

The example from Figure 1 illustrates the best-case scenario for surrogate modelling, where the target function arguments match the model variables exactly. In the general case, however, the target function might reference global or thread-local variables, chase pointers inside nested structures, iterate over an array or collection, or handle missing values such as nulls or unions. To accommodate these, the surrogate API maintains a clean separation between "callsite variables" (defined to be pointers to arbitrary C++ types that are in scope at the target function call site) and "model variables", which are named tensors of primitives.

To express the relationship between these two sets of variables in a declarative way, PHASM maintains a tree of profunctor optics. Profunctor optics are small composable 'getter and setter' objects that are generalized to support nested structures ('lenses'), missing values ('prisms'), one-to-one type conversions ('isos'), and multiple values ('traversals') [7]. They were developed by the functional programming community in order to cleanly read and update nested immutable data structures. Chaining together a sequence of optics yields a new, specialized optic. PHASM's optics tree has roots corresponding to callsite variables, branches that each handle one level of data nesting, and leaf nodes corresponding to model variables. To construct this tree in a clean, intuitive, and correct way, PHASM provides a builder interface with a typed cursor so that it only accepts valid options for the datatype at that nesting level.

Figure 2 illustrates typical usage of PHASM's surrogate API with complex data, and Figure 3

```
struct Vec { double x, y; };                                                    // (1)
double dist = 0;

namespace wrapped {
void manhattan_dist(Vec* coords) {
    for (int i=0; i<4; ++i) { dist += s[i]->x + s[i]->y; }                      // (2)
}}

auto surrogate = phasm::SurrogateBuilder()
    .set_model(std::make_shared<phasm::TorchScriptModel>("model.pt"))
    .global_primitive<int>("dist", &dist, phasm::INOUT)                         // (3)
    .local<Vec>("coords")                                                       // (4)
        .array<Vec>(4)                                                          // (5)
            .accessor<double>([](Vec *s) { return &(s->x); })                   // (6)
                .primitive<double>("x", phasm::IN)                              // (7)
            .accessor<double>([](Vec *s) { return &(s->y); })
                .primitive<double>("y", phasm::IN)
            .end()
        .end()
    .finish();

void manhattan_dist(Vec* coords) {
    surrogate.bind_original_function([&](){wrapped::manhattan_dist(coords);})
             .bind_all_callsite_vars(coords)
             .call();
}
```

Figure 2. Usage of the PHASM surrogate API with fixed-length arrays, structs, and globals.

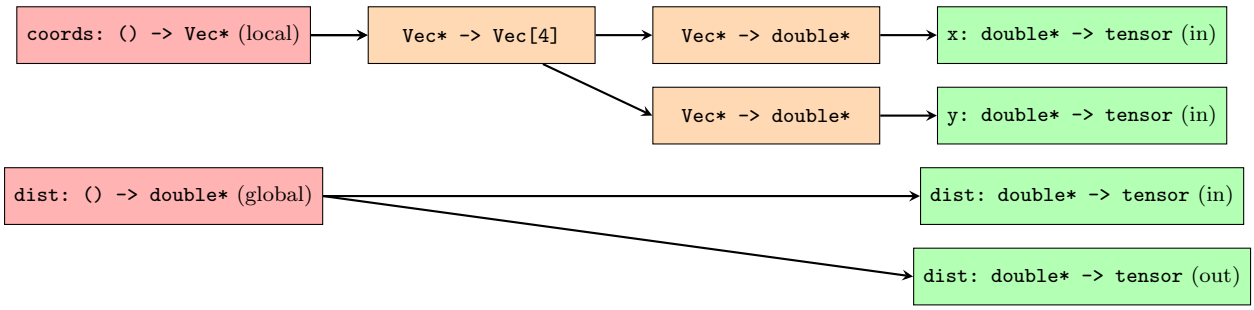

Figure 3. Optics tree corresponding to Figure 2.

shows the corresponding optics tree that is constructed internally. The target function accepts a fixed-size array of `struct Vec` (1), iterates over each entry, summing the `x` and `y` fields, and writing the result to a global variable (2). We start by declaring the global primitive `dist` as both an input and an output (3). Under the hood, one callsite variable is constructed, which is connected directly to two model variables, one for input and one for output. This is important because the original function is impure, yet its tensor equivalent remains pure, which is essential for reliably training the model.

Next we declare a local callsite variable, `coords` (4). The builder opens a cursor to type `Vec*`, constraining any child optics to have that as their input type. Since the `coords` pointer is actually a decayed array, we apply an ‘array‘ traversal (5). Next we define `accessor` lenses that reach into each `Vec*` in that array and pull out the `x` and `y` values (6). Since we have reached

actual model variables, we add a leaf to our optics tree using the `primitive` iso (7). This iso is the base case of a recurrence: it copies the pointed-to data to a fresh tensor object (optionally with a different dtype) and passes this tensor back to its parent optic. Each parent optic manipulates its children's tensors as needed: for instance, the `array` traversal performs a tensor stack operation. By the time the recurrence unwinds to the roots, the tensors corresponding to each model variable will have been fully constructed, and tensors `x` and `y` will have shape $(4,)$ automatically.

Presently, the surrogate API supports C-style data structures: structs, arrays (fixed and variable-length), unions, and nullable pointers. C++ poses additional challenges, such as handling objects that lack a default constructor or maintain an invariant, in particular STL collections. Other plans for the surrogate API include putting the machine learning framework behind a dynamically loaded plugin interface for easier integration with legacy build systems, and communicating with MLFlow for model version management.

6. Long-term goals

The long-term goal for PHASM is to create an interactive tool for designing and testing surrogate models within a legacy scientific codebase, without even having to recompile. This would be analogous to a debugger, as the user could pause a running program and interactively choose options such as rewriting the binary to replace an arbitrary function with a surrogate model, tuning the model's hyperparameters, capturing training data and bounds, and profiling both the original function and its surrogate. This would make it possible to empirically explore the limits of surrogate models for functions of increasing size and complexity.

PHASM is also a convenient platform for further research into SciML uncertainty quantification and continuous learning. If the surrogate model returns an uncertainty estimate below some preset threshold, PHASM could defer back to the original function instead, and capture the result as training data. Furthermore, because PHASM keeps track of the bounds of its input space, it could immediately redirect out-of-bounds inputs to use the original function. Finally, because the mapping between C/C++ types and tensors is fully bidirectional, PHASM could generate new inputs as tensors, transform them back into C++ datatypes, evaluate the original function, and capture the outputs as training data. Thus, PHASM could understand what region of the input space its model is valid for, and then expand this region autonomously. While PHASM could sample the input space by using well-understood techniques such as Latin Hypercubes, it could also generate input samples by leveraging static analysis or fuzzing. The ability to move back and forth between an algorithm's code representation and its neural net representation yields intriguing possibilities.

7. References

[1] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[2] S Furletov, F Barbosa, L Belfore, C Dickover, C Fanelli, Y Furletova, D Lawrence, and D Romanov. ML on FPGA for Event Selection. In *Experimental Applications of Artificial Intelligence 2 for the Electron Ion Collider*, page 8, September 2021.

[3] Nan Ding and Samuel Williams. An Instruction Roofline Model for GPUs. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 7–18, Denver, CO, USA, November 2019. IEEE.

[4] Brendan Gregg. Visualizing Performance with Flame Graphs, 2017.

[5] Tevi Devor and Sion Berkowits. Pin: Intel's Dynamic Binary Instrumentation Engine.

[6] Bence Babati, Gábor Horváth, Viktor Májer, and Norbert Pataki. Static Analysis Toolset with Clang. In *Proceedings of the 10th International Conference on Applied Informatics*, pages 23–29. Eszterházy Károly University, 2018.

[7] Matthew Pickering, Jeremy Gibbons, and Nicolas Wu. Profunctor Optics: Modular Data Accessors. *CoRR*, abs/1703.10857, 2017. arXiv: 1703.10857.