A Metric for HPC Programming Model Productivity

Wei-Chen Lin
School of Computer Science
University of Bristol
Bristol, UK
wl14928@bristol.ac.uk

Tom Deakin
School of Computer Science
University of Bristol
Bristol, UK
tom.deakin@bristol.ac.uk

Simon McIntosh-Smith
School of Computer Science
University of Bristol
Bristol, UK
S.McIntosh-Smith@bristol.ac.uk

***Abstract*—There has been a healthy growth of heterogeneous programming models that cover different paradigms in the HPC space. Selecting an appropriate programming model for new projects is challenging: how does one select a model that is both productive and performant? The same applies for existing projects aiming to leverage heterogeneous offload capabilities.**

While characterisation of programming model performance has been abundant and comprehensive, productivity metrics are often reduced to basic measures like Source Line of Code (SLOC). This study introduces a novel model divergence measure to objectively evaluate productivity. We cover common aspects of productivity, including syntax, semantics, and optimisation overhead.

We present a productivity analysis framework supporting GCC and Clang, covering models for C/C++ and Fortran. We evaluate our metric using this framework on select mini-apps, including some from SPEChpc, and propose a combined productivity and performance probability visualisation for a comprehensive picture.

***Index Terms*—Performance portability, productivity, StdPar, CUDA, HIP, Kokkos, OpenMP, OpenMP target, SYCL, TBB, Fortran, benchmarking, semantics**

I. Introduction

A parallel programming model is an encoding of constraints that helps programmers to express parallelism in their code. To illustrate, consider a commonly used encoding of doing parallel work using thread objects (e.g., `pthread` objects in Linux). The thread objects constrain and model how threads are mapped to physical CPU threads. Another way to encode this abstraction is parallel combinators such as map and reduce, as seen in C++ PSTL (StdPar). The interfaces of the combinator functions constrain what kind of input and output is possible. These are many ways to encode a similar constraint, and these form the basis of a programming model.

With advances in both heterogeneous computing and traditional CPUs from various vendors, the need for portable parallel programming models has increased. Many of the programming models we see today are capable of expressing similar constraints but with different design philosophies.

For example, directive models such as OpenMP use declarative annotations to minimise the changes required to the existing codebase when introducing parallelism. On the other hand, imperative models like SYCL provide an idiomatic C++ API, where the programmer is responsible for transforming their code to exploit parallelism. First-party models, such as CUDA and HIP, use a C++ dialect to encode accelerator-specific constraints (e.g., locality of data), thus giving the programmer maximum control in expressing parallelism. Finally, we also see library-based abstractions like Kokkos and RAJA, each providing an opinionated API for the supported backends.

Each model presents its own strengths and trade-offs, catering to different programming preferences and requirements. While the expressed parallelism may be similar, the productivity property may differ significantly depending on the design choices of the model, whether technical or philosophical. Understanding how productivity is influenced by these design choices is critical in helping programmers select the most productive model for the task at hand. Beyond just selecting a programming model, understanding the productivity aspect also helps inform potential optimisations to the model itself to improve productivity.

The current state of the art in objective metrics for measuring the productivity of a programming model frequently uses methods like measuring the SLOC (Source Lines of Code). For example, a programmer might measure the SLOC increase from a serial version of their code to a port that uses a heterogeneous programming model. The degree of SLOC increase is then used to determine whether the performance improvement, or lack thereof, is justified. Similarly, the verbosity of models can be measured by comparing implementations of the same application using different programming models, although care must be taken that they are intrinsically equally skillfully written.

Beyond objective measures, productivity may also be summarised subjectively based on the overall experience of using the model. Overall, productivity measures derived from the codebase itself are an area that has not been widely explored.

A. Contribution

We introduce a novel model divergence metric to capture the semantic information of the codebase. This new metric offers a higher level of insight than the widely used SLOC measure. Specifically, this paper presents:

- A novel tree-based method of summarising a codebase while retaining semantic information for both C/C++ and Fortran.
- A method for evaluating the divergence of semantic-bearing trees.

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00160

- An open source unified software framework [1] that provides an end-to-end workflow to collect and analyse semantic-bearing trees.
- Evaluation of the new metric against the state of the art using select HPC benchmark codebases chosen from SPEChpc.
- Methods of combining this metric with the performance portability metric [1].
- Insight on the semantic and perceived complexities of different programming models.

This paper focuses solely on the codebase and does not directly address the human side of productivity, including programmer performance, behaviour, and experience. These aspects of programmer productivity have the potential to be subjective depending on a programmer's background knowledge. For example, a Haskell programmer might not implement a stencil algorithm over a structured grid in the same way a C++ programmer would. Similarly, programmer experience—or the element of frustration, as coined by Harrell et al. [2]—may be more pronounced for a programmer more familiar with the functional programming paradigm.

II. Related work

Despite the recent rise of multiple heterogeneous and parallel programming models in the HPC space, objective measures beyond SLOC for productivity are sparse.

In Youssef's survey of k-means cluster algorithms on the GPU using SYCL, a reduction of the SLOC is associated with an increase of productivity [3]. Asahi on the other hand, has attributed the productivity of heterogeneous code directly to the programming model itself, where the act of using one, such as Kokkos, is a productivity improvement [4].

Classically, there has been a constant focus on the human factor.

Stephen et al. [2] have used survey projects to investigate how productivity and performance portability can be measured effectively. Wienke et al. proposed a productivity model where the aim is to predict HPC software development cost [5]. His method includes the use of SLOC as a factor, along with many other metrics on software development workflow.

Pennycook et al. [6] introduced a related code divergence metric that measures lines that are different between two ports of an application. Their method involves a Jaccard difference of two codebases using regions that differ textually after the execution of the preprocessor. The result is a divergence value in the range of 0 to 1, which is then used to generate a dendrogram and a navigation chart when paired with performance portability metrics. Our work is inspired by this approach but with enhancements to add the semantic dimension that the compiler already uses internally for compilation.

III. Model divergence metric

This section describes our novel programming model divergence metric, how it measures productivity, and how it is related to other kinds of productivity metrics such as Source Lines of Code (SLOC).

In general, measuring the productivity of a programmer while using a certain programming model will always have elements of subjectivity due to human involvement. This is different from objective measures such as the performance portability metric (Ᵽ) [1], which can be measured by executing and comparing benchmarks results. We explore the extent to which one can adopt a similar workflow to Ᵽ, attempting to derive relative metrics without human subjectivity in the loop.

First, we define a serial codebase S with no parallelism as a baseline representing maximum productivity (indeed, parallel programming is hard [7]). This serial codebase uses the standard control constructs of the implementation language, with no special consideration for memory allocation or address space.

Then, we introduce codebase P which is a port of S that expresses parallelism using the programming model M. P is ported idiomatically using M and exhibits the expected speedup from the baseline codebase S; this assumption captures that the port is effective at leveraging parallel performance. The divergence between S and P represents additional porting effort for the programmer, which reduces the productivity.

Note that we do not consider productivity in the sense of application usability, nor run time, here, such that a user of the parallel code might expect output in human timescales such as, before the deadline, overnight, etc.

A frequently used measure in current literature when presented with different ports of S is to count the increase of SLOC against either S or an existing P. For example, Lin et al. previously compared the SLOC needed for the StdPar port against the CUDA versions of mini-apps as a measure of programming effort [8]. However, the SLOC measure does not capture any semantic information from either codebase and could be misleading due to factors such as linebreak preferences and macros. Utilising tools to formalise coding styles may also be problematic when programming models have idiomatic styles that might diverge from usual conventions in the underlying programming language. A slightly more involved method might use Logical Lines of Code (LLOC) or compute the edit distance (e.g., Levenshtein distance or `diff`), or number of locations to edit irrespective of the size of the edit (the authors recall this was suggested in workshop discussion by S. Hammond) between codebases. However, these methods are prone to anchoring issues where the ordering on both sides of the comparison is important.

A. A Tree-Based metric

To capture the semantic information and avoid ordering issues, we first propose a method where a codebase can be reduced into multiple *semantic-bearing trees* for comparison.

From the compiler's perspective, the source program is already represented as trees at all levels of the compilation. We take advantage of this by extracting the Abstract Syntax

[1] https://github.com/UoB-HPC/SilverVale

Tree (AST) at both the frontend and the backend of the compilation process. Let T_{sem} represent information extracted from the frontend AST (e.g., ClangAST). T_{sem} retains symbolic relations, expansions (i.e., templates or generic types), and any other constructs that influence the code generation. Crucially, we keep a reference to the source location for all nodes in this tree (i.e., line and file name).

For the backend, let T_{ir} represent the platform-independent Intermediate Representation (IR) AST (e.g., LLVM IR) before machine code generation. This tree retains the semantics of the low-level but platform-independent machine operation. To keep T_{ir} comparable, the IR used must be stripped of architecture-specific information. Like T_{sem}, we retain all source location references.

Where possible, we also collect the Concrete Syntax Tree (CST) either from the compiler directly or through a separate lexer. CST is sometimes referred as the parse tree as it captures all syntactical tokens required to fully reconstruct the source. For example, CSTs usually retains low semantic-value tokens such as commas in source text, even when the comma separated text is already parsed into a list of tokens. We collect both the AST and CST so that we can compare how much boilerplate syntactic token is present in the source that subsequently gets discarded at the frontend.

Let T_{src} represent information extracted from the CST after normalisation. The normalisation process removes noise such as space, comments, and control tokens. T_{src} contains a tokenised view of the source with nodes that represent syntactic elements. This is conceptually similar to what syntax highlighters provide. For example, the CST cannot discriminate between function calls and functional-style casts in C++; both are considered a call token. Languages that include a preprocessing phase will yield two T_{src}: one before and one after the preprocessor.

The back reference from the semantic trees to the source code is important and serves multiple purposes. First, this information is necessary for reconstructing the dependency tree between all source units. This process enables the calculation of secondary metrics such as module coupling [9] and overall tree complexity. Additionally, back references are useful for annotating or pruning specific parts of the tree based on source location. For example, we use runtime coverage data to eliminate parts of the tree that were never executed. The coverage data, provided by most compilers, reference the original source location which is used to mask out parts of the tree.

B. Tree distance

Now that we have defined a way to summarise a codebase in semantic-bearing trees, we introduce a method for comparing these trees to form a relative divergence metric. Similar to string edit distance, to compare distances between trees we use the Tree Edit Distance (TED) algorithm. TED is defined as the minimal amount of deleting, inserting, and relabelling of tree nodes required to transform one tree to another [10]. For example, consider the comparison of two simple ClangASTs shown in Fig. 1, the dotted lines show nodes that are common between two trees. The remaining five nodes are either relabelled, inserted, or deleted.

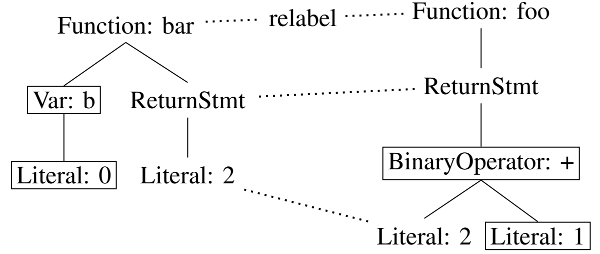

Fig. 1. Two ASTs with a TED distance of five: four outlined nodes are inserted or deleted with one relabelled node on the top.

We denote $d_{TED}(T_1, T_2)$ as the TED between n-ary trees T_1 and T_2. TED is applicable for any n-ary tree structure, and the algorithm supports specifying different weights for different operations. For simplicity, we use the unit weight of one for all nodes and operations. A future study may associate different weights depending on operations and node types; adding new code may have a different productivity impact than removing existing code.

To avoid excessive relabelling of nodes that have programmer-introduced names, we normalise names by retaining only the token type. For example, all variable, function, and class names are removed. For TED, this has the effect of preserving the overall semantic structure and control flow graph such that a subtree with the closest structure will have the minimal distance.

C. Tree-Based Model Divergence (TBMD)

TABLE I
CODEBASE SUMMARISATION METRICS

Metric	Measure	Domain	Variants
$SLOC$	Absolute	Perceived, Language agnostic	+preprocessor +coverage
$LLOC$	Absolute	Perceived, Language agnostic	+preprocessor +coverage
$Source$	Relative (Edit distance)	Perceived, Language agnostic	+preprocessor +coverage
T_{src}	Relative (TED)	Perceived	+preprocessor +coverage
T_{sem}	Relative (TED)	Semantic	+inlining +coverage
T_{ir}	Relative (TED)	Semantic	+coverage
Performance	Relative (Φ)	Runtime	N/A

As our metric is novel, we compare it with existing metrics to highlight its strength and potential advances over the state of the art. Table I outlines the measure, domain, and possible variants when used on a codebase. An empirical evaluation is done for all metrics listed later on in Section V.

Given a codebase C containing the set of source files $F_C = \{F_1, F_2, \ldots, F_n\}$. We consider a unit of comparison as the source file itself, and all its module dependencies (i.e., headers), denoted $unit_C(x)$ in Eq. (1) for some file x in C. This is crucial for correctly handing languages like C++ where the header may contain large chunks of the program logic. While this definition also includes artefacts such as system headers, those can simply be masked out during the analysis phase.

$$unit_C(x) = dep(x) \cup x \tag{1}$$

Using this unit definition, we discuss the concrete metric listed in Table I. This table is sorted by the kind of semantic information given. Starting from the top, with the classic $SLOC$, measured in the number of lines break characters in the source file. The metric, together with all perceived metrics discussed ($SLOC$, $LLOC$, $Source$), is applied after normalisation of white spaces and comments. This is inline with the SLOC definition proposed by Nguyen et al. [11]. Whitespace normalisation removes consecutive whitespace characters while preserving all other tokens. In the same vein, comments are removed using ranges marked by a CST.

In all cases, we make special provisions for language that store semantic-bearing information in unusual places. For example, OpenMP pragmas are identified and retained even after preprocessing and normalisation steps. Languages that use special comment tokens for directives are also handled.

Like $SLOC$, we use the LLOC definition from Nguyen et al. [11]. $LLOC$ considers certain tokens as a logical line—for example, a for-loop header in C++ would be counted as a single line regardless of linebreak. This requires lexical understanding of the source, which is again available from the CST. $SLOC$ and $LLOC$ are both absolute measures, meaning that a single value can be computed from one codebase. Eq. (2) and Eq. (3) defines the $SLOC$ and $LLOC$ respectively on codebase C as the sum of the metric on all units in the codebase.

$$SLOC(C) = \sum_{x \in F_C} SLOC(unit_C(x)) \tag{2}$$

$$LLOC(C) = \sum_{x \in F_C} LLOC(unit_C(x)) \tag{3}$$

$Source$ is a relative measure where it compares unit pairs from two codebases textually, as defined in Eq. (4).

$$d_{LCS}(C_1, C_2) = \sum_{\{F_{C1}, F_{C2}\} \in match(C_1, C_2)} LCS(F_{C1}, F_{C2}) \tag{4}$$

This is the sum of the longest common subsequence (LCS) on all unit pairs between the two codebases, where each file of the unit is normalised like $LLOC$. The $match$ function determines which two units should be paired together, indicating that the files implement equivalent parts in their respective code bases. This function is specific to the codebase being compared, but logically, it should pair units with the same purpose.

In principle, $match$ is not required as the entire codebase can be treated as a single large tree with units at the first level. In practice, this adds significant runtime overhead when solving for TED on codebases beyond a few thousand lines, so providing $match$ where possible accelerates the workflow.

Finally, metrics T_{src}, T_{sem}, and T_{ir}, are tree-based relative measures where we compare semantic tree pairs from two codebases. T_{src} covers the perceived semantics from each unit, as viewed by a programmer in a syntax-highlighted text editor. Then, T_{sem} covers the semantic information as interpreted by the compiler. We end with T_{ir} covering low-level machine semantics as interpreted by the abstract hardware.

$$T_{src}(F) = T_{sem}(F) = T_{ir}(F) = \bigcup_{x \in F_C} unit_C(x) \tag{5}$$

For all these cases, we use the sum of the TED distance of all tree pairs defined in Eq. (6) and the tree constructor defined in Eq. (5).

$$d(C_1, C_2) = \sum_{\{F_{C1}, F_{C2}\} \in match(C_1, C_2)} d_{TED}(T(F_{C1}), T(F_{C2})) \tag{6}$$

A value of zero indicates the codebase compared is identical. These three tree metrics form the basis of the novel divergence metric; we collectively call this the Tree-Based Model Divergence, or the *TBMD* metric.

We define a bound for TBMD that denotes the maximum divergence of two codebases such that no semantic similarity exists, shown in Eq. (7).

$$dmax(C_1, C_2) = \sum_{\{F_{C1}, F_{C2}\} \in match(C_1, C_2)} |T(F_{C2})| \tag{7}$$

Let $|T|$ represent the total number of tree nodes in tree T. $dmax(C_1, C_2)$ gives the maximum distance $d(C_1, C_2)$ can yield before C_2 is considered an entirely different codebase from C_1. This is different from a divergence upper-bound, which we do not define. Intuitively, the maximum divergence is determined by the amount of change necessary to remove all nodes from one codebase and then fully reintroducing nodes from another. Maximum divergence is primarily useful for normalising TBMD for it to be comparable across divergences of multiple models.

IV. SilverVale analysis framework

This section describes the software framework we implemented to capture the required tree information discussed in Section III. For a comprehensive study, we design our framework to cover two primary language categories widely used in the HPC space: C/C++ and Fortran.

We designed compiler plugins to implement the extraction of semantic-bearing trees discussed in the previous section. For C/C++, we devlop against the open-source Clang and GCC compilers. These two compilers cover many HPC compiler uses and provide a friendly plugin system for developers.

For Fortran, we use GFortran from GCC. There are other equally popular compilers for Fortran, but many are proprietary and do not provide a plugin system with the required capabilities. At the time of writing, the Clang-based `flang-new` compiler did not provide a robust OpenMP implementation that can compile common Fortran HPC codebases.

Modern codebases typically involve multiple source files and may have complex configuration steps that generate additional source files and set macro arguments. We design our framework to handle this robustly by using Compilation Databases (DBs). These DBs are a single `compile_commands.json` file that records the compiler invocations used to compile the codebase to completion. Build tools such as CMake and Meson natively support generating Compilation DBs, and third-part tools for GNU Make are also available.[2]

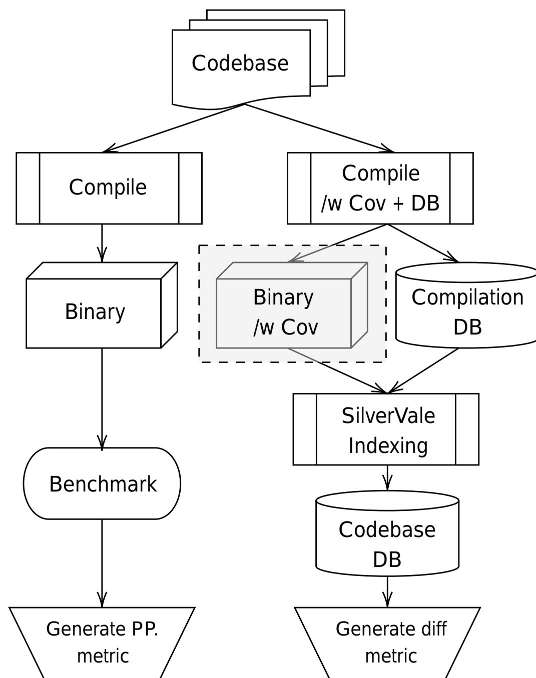

Fig. 2. Divergence metric workflow, optional coverage data in grey

Fig. 2 shows where our framework fits into an end-to-end workflow. The SilverVale framework ingests a Compilation DB file from a codebase that has been successfully compiled previously. The user can optionally provide a binary that has been compiled with coverage options alongside the generated coverage data. At this stage, SilverVale generates a Codebase DB where it indexes all compiler invocations in the Compilation DB. The result is a portable set of semantic-bearing trees and metadata files all stored in a Zstd compressed MessagePack format.

The following section discusses the challenges and overall workflow for extracting semantic-bearing trees from both Clang and GCC.

[2] https://github.com/rizsotto/Bear

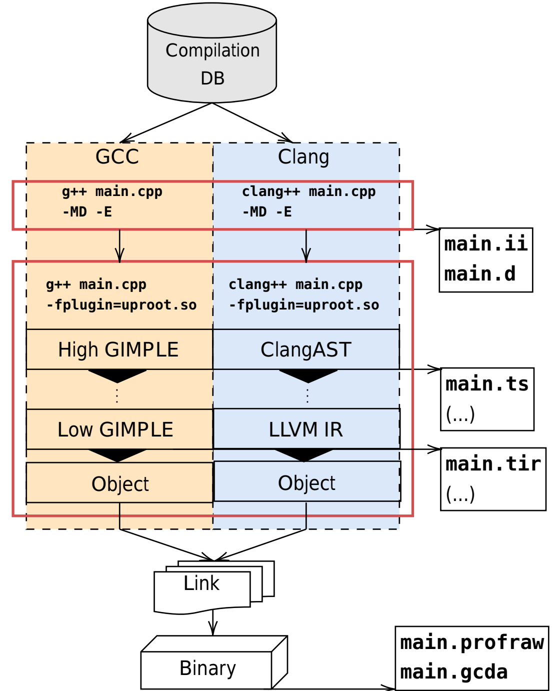

Fig. 3. SilverVale internal tree capture process

A. Compiler: Clang

The Clang compiler supports user plugins and provides a set of modular and easy-to-use AST traversing APIs for the underlying ClangAST. ClangAST is the primary AST used for all C/C++ dialects, including CUDA, HIP, and directives such as OpenMP. Internally, C++20 modules and precompiled headers in Clang are implemented using ClangASTs that are serialised to files.

At the frontend, the representation of the source code in ClangAST is straightforward, but a lot of non-semantic bearing nodes are also required to capture nuances of the C++ language. For example, implicit and value category casts are prevalent and visible in most statements. To generate T_{sem}, we discard all non-semantic nodes and record only the node type, literal, and operator names. Two T_{sem} are generated: T_{sem+i} which inlines all function invocations that originated from the same source at the tree level (i.e., system headers or libraries are excluded), and the one that is as written (T_{sem}, unmodified). T_{sem+i} captures the case where the codebase itself attempts to abstract over a parallel programming model.

To generate T_{ir} from the backend, we directly invoke Clang again but adjust the compile command to emit LLVM

bitcode. This only works well for host code; for offloading models, including HIP, CUDA, and OpenMP target, the output format is a Clang offload bundle[3] that contains bitcode for multiple targets. In later versions of Clang, depending on the dialect, the compiler driver may embed the bitcode directly in the host bitcode file under a special section annotated with `@llvm.embedded.object`. For both cases, we extract the embedded bitcode and parse it into a semantic tree. Like the frontend tree, we discard all symbol names but retain instruction names, functions, basic blocks, and globals. Fig. 3 gives a high-level flowchart of the artefacts generated at each of the stages in the compilation process.

B. Compiler: GCC

Like Clang, GCC (including GFortran) supports compiler plugins, although the exposed API is significantly more complex and is only sparsely documented. GCC lowers (translates) C/C++ source code to GIMPLE, GCC's internal tree representation designed for optimisation passes. Other languages like Fortran first lower to a different IR, GENERIC, which is then converted to GIMPLE as a later step. GIMPLE is functionally similar to ClangAST but represents the node using a tuple instead of an arbitrary tree. This tuple structure is not comparable to ClangAST in any meaningful way, so cross-compiler comparison is not possible.

Unlike Clang, GCC uses GIMPLE for both the frontend, and the backend before target-specific code generation. Early on in the frontend, GCC uses *High GIMPLE* to represent the semantics of the source language. Towards the backend, *Low GIMPLE* is used which contains a reduced set of instructions, with control structures rewritten as `goto` statements to go between basic blocks. After this, target-specific code generation begins, which converts *Low GIMPLE* to RTL for allocating machine registers.

We generate both T_{sem} and T_{ir} from the two GIMPLE variants by rewriting all tuple representations back to a tree-like structure while retaining semantics, as shown in Fig. 3. Unlike Clang, generating the inlined tree T_{sem+i} requires significant effort due to how lambda expressions are represented, so we have omitted this for GCC. We also do not consider offload scenarios for GCC at this time, and is something we plan to investigate in the future.

C. Concrete Syntax Tree

Modern compilers do not usually have a clearly demarcated phase where a parse tree solely describes the lexical properties of the source; some semantic information will be embedded for convenience. For both Clang and GCC, the plugin API does not expose a CST of any kind. To obtain a T_{src}, we use the tree-sitter[4] library to generate parse trees and filter out anonymous tokens. The tree-sitter library is a proven syntax and symbol parsing library that GitHub internally uses for static code analysis [12]. The library has a wide range of official language support, including C/C++, variants for CUDA/HIP, and Fortran.

Using a CST that is decoupled from the compiler has an additional advantage where the resulting T_{src} are comparable within the same language, and not just with the same compiler. Recall that T_{sem} is not comparable across compilers due to semantic differences.

D. Coverage

SilverVale implements coverage support and accepts profile data generated using Clang's source-based code coverage option or GCC's GCoverage. If the codebase under analysis is compiled with coverage flags, SilverVale can be instructed to use these profile data as part of the index step. Internally, the coverage data is converted to a line-based mask that can be toggled for any tree structure or source file.

E. TED and string distance

The time and space complexity of TED is high: the naive recursive implementation exhibits exponential time complexity [13]. However, due to the versatility of the TED algorithm in various domains that handle structured data, the newest method, APTED, proposed by Pawlik et al. lowers the time complexity to just $O(n^2)$ [14]. We use the open-source implementation[5] of APTED implemented by Pawlik et al. themselves [15].

For comparing source files, we use the well-established string sequence distance algorithm proposed by Wu et al. [16]. This is the algorithm used internally by the Linux utility `diff`. Like APTED, we have integrated a high-quality and open-source implementation[6] created by Tatsuhiko Kubo [17].

To reduce the complexity of implementing SilverVale, we implemented a C++ header-only combinator library called Aspartame[7]. Aspartame is similar to C++20's range library but provides a much richer set of operations inspired by the Haskell and Scala programming language.

V. Evaluation

The following subsections will evaluate our proposed TBMD metric. We will use a set of mini-apps that are implemented in multiple programming models to assess whether our metric provides any more insight than SLOC.

We use four mini-apps with different runtime characteristics, as listed in Table II. BabelStream is the McCalpin STREAM benchmark but implemented in heterogeneous models for both CPUs and GPUs [18]. We include the recently added Fortran version [19] to evaluate SilverVale's Fortran support. Similar in spirit to BabelStream, miniBUDE is a molecular docking mini-app implemented in multiple programming models [20], representative of the larger scientific code, BUDE.

For larger applications, CloverLeaf is a CFD application using a structured grid to solve PDEs. Likewise, TeaLeaf is a heat equation solver that uses the Conjugate Gradient

[3]https://clang.llvm.org/docs/ClangOffloadBundler.html
[4]https://github.com/tree-sitter/tree-sitter
[5]https://github.com/DatabaseGroup/tree-similarity/
[6]https://github.com/cubicdaiya/dtl
[7]https://github.com/tom91136/Aspartame

TABLE II
CODEBASE SUMMARISATION METRICS

Mini-app	Type	Models
BabelStream Fortran	Memory BW	Sequential, Array, DoConcurrent OpenMP, OpenMP Taskloop OpenACC, OpenACC Array
BabelStream C++	Memory BW	Serial, CUDA, HIP OpenMP,OpenMP target, Kokkos, StdPar, TBB, SYCL (USM/Accessors)
miniBUDE	Compute	Serial, CUDA, HIP OpenMP,OpenMP target, Kokkos, StdPar, TBB, SYCL (USM/Accessors)
CloverLeaf	Memory BW Structured grid	Serial, CUDA, HIP OpenMP,OpenMP target, Kokkos, StdPar, TBB, SYCL (USM/Accessors)
TeaLeaf	Memory BW Structured grid	Serial, CUDA, HIP OpenMP,OpenMP target, Kokkos, StdPar, TBB, SYCL (USM/Accessors)

method. Both of these mini-apps are part of Sandia's Mantevo benchmark suite, and the base OpenMP version has recently been added to the SPEChpc suite [21].

It is imperative that we start our evaluation on smaller mini-apps. The purpose of mini-apps is to have proxies that mirror the real application but without the additional complexity associated with the full application. As such, any discrepancies from expectation can be easily investigated. Although mini-apps do not necessarily capture some of the complexities around source coding practices or size of the code base, by using production-ready tools to ingest the source (GCC, Clang, tree-sitter, etc.), we have engineered our approach to work for such applications, with successful preliminary tests.

When comparing models, we consider some variants of a model as distinct. Namely, OpenMP and OpenMP target, and SYCL with or without USM. While OpenMP target is just an extra set of directives for accelerators, more often than not, the expression of parallelism and data movement is different from host OpenMP. For SYCL, the USM model removes a significant amount of the boilerplate but shifts the complexity of memory management to the runtime. We would like our metric to capture these productivity variables.

For all comparisons, we use Clang 17 as the primary compiler and also GCC version 13 for Fortran codes. We use Clang's built-in support for the CUDA and HIP model. For SYCL, we use the edge commit (August 2024) of the Intel LLVM fork as SYCL support in upstream Clang is still at the RFC (Request For Comment) stage.

Note that any boilerplate code shared between all models will not have any impact on the metric as those simply evaluate to a divergence of zero as the trees will be identical. If a specific model has deviation from the shared boilerplate, then this will also be automatically captured.

A. Semantic retention: C/C++

In this section, we use TeaLeaf to highlight the semantic-retention properties of TBMD. We select TeaLeaf because the amount of code expressed in any given programming model is balanced in terms of shared and specialised model code. This is not the case for miniBUDE or BabelStream, where the code has a higher ratio of boilerplate to actual algorithm as the computational kernels are relatively short in SLOC.

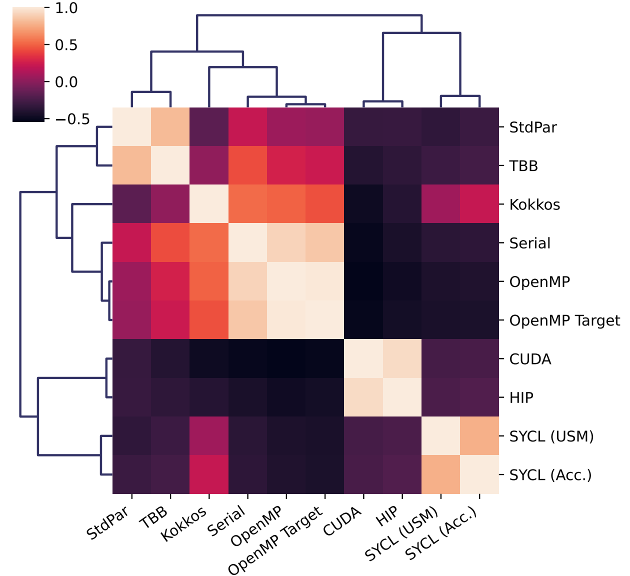

Fig. 4. TeaLeaf model clustering, using T_{sem}

We setup TeaLeaf to be compiled with the Clang compiler and generate a compilation database. We then execute our SilverVale tool over TeaLeaf, on a total of ten models, including two variants. We run the comparison step over the cartesian product of all models to yield a correlation matrix. Fig. 4 shows the clustering of all models. We generate the associated dendrogram around the map using complete linkage and Euclidean distance between points. In brief, the proximity to the closest branch of any two groups represents the degree of similarity; clusters connected with a short distance before a branch are more similar.

We observe a clear clustering of model variants and models that are related in terms of design philosophy. For example, both variants of SYCL, and OpenMP, are grouped into a cluster, and the HIP model is grouped with CUDA. The serial model appears to be close to the OpenMP variants. This validates the design philosophy where OpenMP can mean minimal changes are required to your code.

We now explore the clustering of both existing metrics together with TBMD in Fig. 5. Overall, $SLOC$ and $LLOC$ did not group related models together, and the clustering appears random, confirming our experiences of their limited use as productivity metrics.

Comparing $Source$, T_{src}, and T_{sem}, we start to see an almost identical clustering. Here Kokkos has proximity to SYCL, but it is later grouped with OpenMP and serial models when using T_{src} and T_{sem}. As investigation on the tree showed that while Kokkos may be textually similar to SYCL in certain locations, the difference is outmatched by the number of semantic-bearing elements the SYCL model introduces. Concretely, the core SYCL API surface is heavily templated with non-visible but semantic-beaning elements such as default

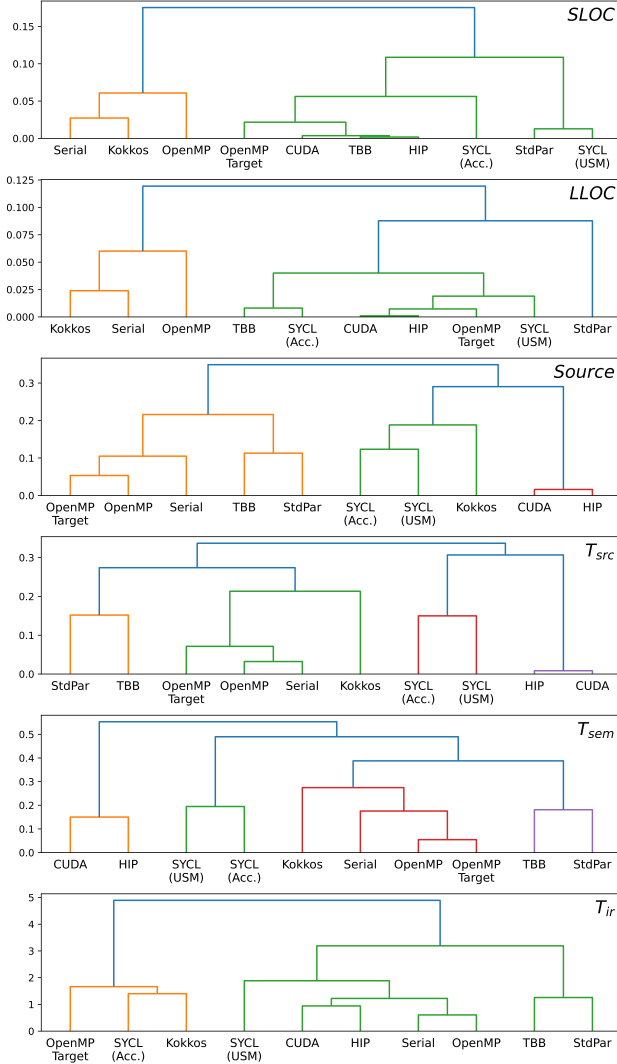

Fig. 5. TeaLeaf model clustering dendrogram, using $LLOC$, $SLOC$, $Source$, T_{src}, T_{sem}, and T_{ir}

values of parameters or even templates. In other words, SYCL tries to hide semantic complexities using the C++ syntax and managed to achieve a lower divergence at the source level.

The relation of TBB and StdPar is interesting. In $Source$, T_{src}, and T_{sem} of Fig. 5, TBB and StdPar are grouped in the same cluster. In essence, the two models look similar and exhibit similar semantics in the case of TeaLeaf. Indeed, StdPar on CPUs are frequently implemented using TBB, as Lin et al. have explored previously [8], and the similarity in API design may have led to this decision. In fact, STL directly influences the API surface of TBB: Reinders et al. make many references about this inspiration in their textbook on TBB [22].

At the IR level, some host models maintain clustering while other offloading models are grouped in a different cluster. For offload models, we observe IR elements that are not part of the program, but runtime support code that help setup the device. Since BabelStream contains only five short kernels, we do not see any meaningful clustering for T_{ir} except for host-only models.

B. Semantic retention: Fortran

We now apply the same semantic analysis for the Fortran version of BabelStream. This is the only mini-app where we find a wide range of programming models to compare against for the Fortran language. Note that both TeaLeaf and CloverLeaf have a version in Fortran using OpenMP and MPI, but due to time constraints, we do not evaluate them in this paper, instead using the C++ versions which have wider coverage of programming models. Fig. 6 shows the clustered dendrogram for BabelStream Fortran using the metrics $LLOC$, $SLOC$, $Source$, T_{src}, T_{sem}, and T_{ir}.

Just as we found with C/C++ previously, $SLOC$ and $LLOC$ contain no semantic information, so the clustering appears random. $Source$, T_{src}, and T_{sem} all clustered in a similar manner where OpenACC and the rest of the models are in distinct groups. On inspection of T_{sem} and T_{ir}, we found that the OpenACC model, including the array variant, did not introduce extra tokens related to parallelism. This outcome is consistent with the single-threaded performance as evaluated by the OpenACC port's authors [19], where they note a possible quality of implementation issue in GCC.

Overall, all the models at T_{sem} are more similar when compared to the C++ version of BabelStream.

C. Metric model relation

Since we support different variants of each metric, such as with and without preprocessor or coverage, we now take a deeper look at all the metrics together. The coverage modifier is obtained by recompiling the application under analysis with the appropriate coverage flags and then running the application with a reduced problem set. The resulting coverage data is then fed into to the indexing step of SilverVale tool to use as a mask for tree-based representations.

When comparing the serial code (model) to itself, as seen in the left most column of Fig. 7 and Fig. 8, we see a correct divergence of 0 for all metrics.

SYCL, when using the CPP modifier ($Source{+}pp$), exhibits extreme divergence from the serial model. This appears to be an artefact of the two-pass compilation used by Intel DPCPP, the effect of this compilation model is discussed in depth by Aksel et al. while implementing the oneAPI specification [23]. The complex host-device macro expansion yields a nearly 20 MB header when `<CL/sycl.hpp>` is included. Additionally, the compiler attempts to insert several integration headers as part of the process, which may have introduced unnecessary source files.

The directive-based OpenMP has a consistently higher T_{sem} divergence when compared to T_{src} or other perceived metrics. We found that Clang has OpenMP-specific AST tokens that capture OpenMP semantics. In essence, the subtree containing an OpenMP token is handled at the compiler level: the

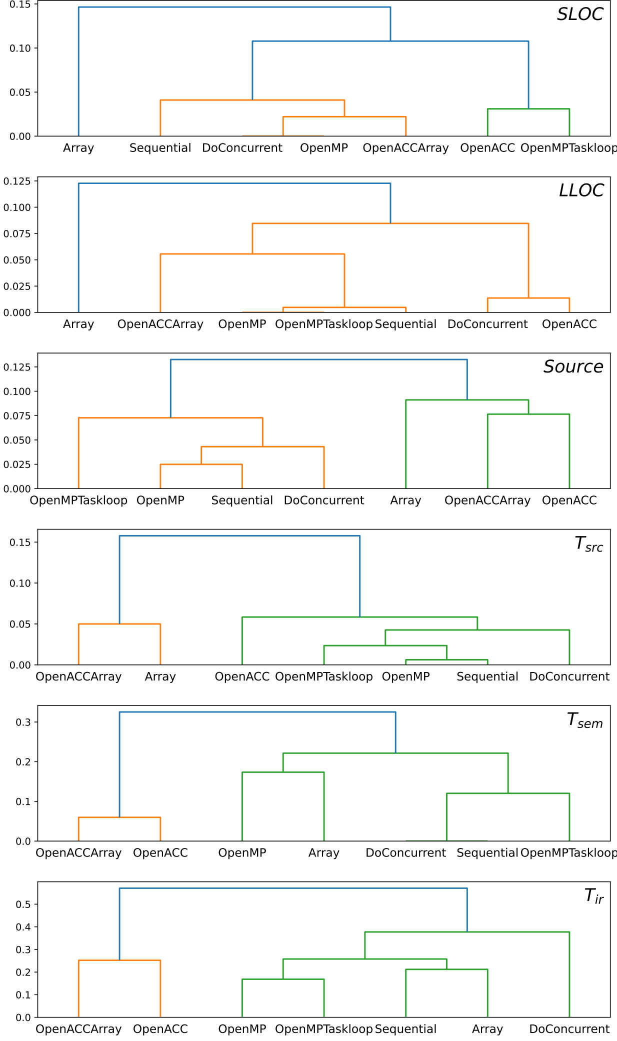

Fig. 6. BabelStream Fortran model clustering dendrogram, using $LLOC$, $SLOC$, $Source$, T_{src}, T_{sem}, and T_{ir}

semantic meaning is ascribed in a way that is opaque in the source. In other words, OpenMP pragamas provide additional semantics beyond those of the base language. And because of this, no other metric was able to capture this information. Intuitively, using OpenMP may require a few lines of directive and look simple, the actual transformation that happens is involved and may not be any less complex than, say Kokkos. We found GCC to also have OpenMP tokens in the AST. Unfortunately, due to time constraints, we did not conduct a full experiment to investigate this further.

The inlined semantic tree T_{sem+i} inlines any function calls back into the tree. As expected, for library-based or language-based models, we see a huge jump in divergence as foreign code is brought in to the tree. For OpenMP, and to a lesser degree CUDA, both show very little change for T_{sem+i}. These two models rely on the compiler to introduce semantics, so

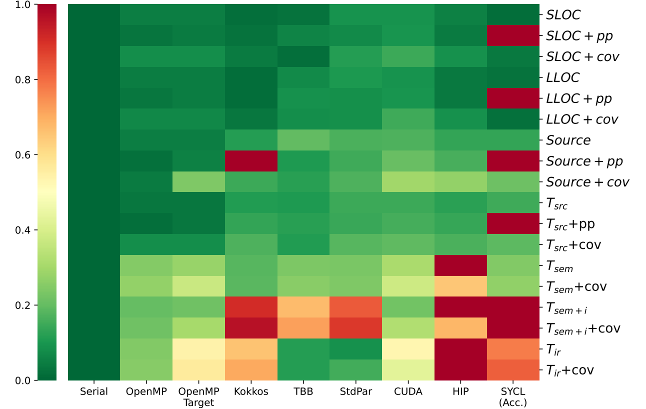

Fig. 7. miniBUDE models, where divergence from serial is plotted from in a heatmap from 0 to 1

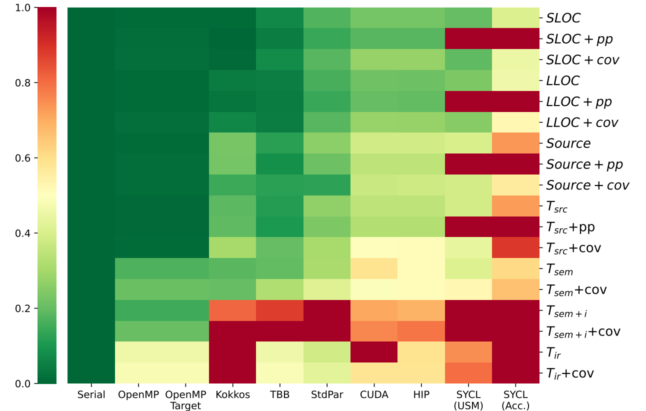

Fig. 8. CloverLeaf models, where divergence from serial is plotted from in a heatmap from 0 to 1

nothing gets inlined. HIP, on the other hand, requires non-trivial runtime headers, so we still see a higher divergence compared to T_{sem+i}.

T_{ir} seems to misbehave for offload models. Initial investigation suggests that the obtained IR contains multiple layers of driver code that is unrelated to the core algorithm. This is repeated for each file, thus artificially increasing the divergence. The coverage modifier has limited success in recovering this metric as Clang does not offer a way to instrument device code to generate coverage.

D. Code Migration

Programming model migration commonly occurs for HPC applications. As a motivating example, we consider the scenario where an existing code may have a CUDA implementation at first because NVIDIA GPUs were the first platform available when the code was developed. As other systems with GPUs from other vendors were available, the base CUDA version may need to be ported to run on other platforms. Other similar scenarios can be described.

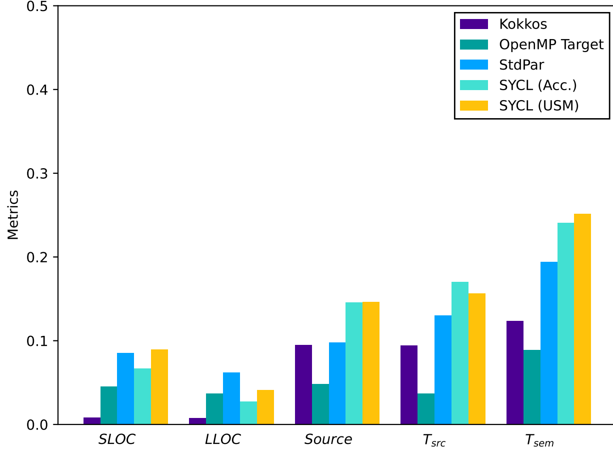

Fig. 9. Model divergence from the serial model of TeaLeaf

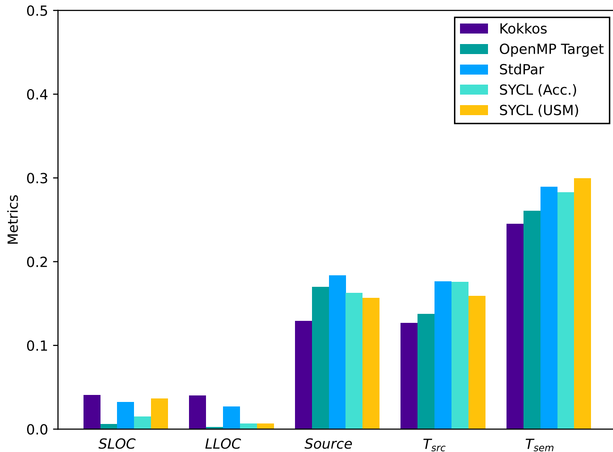

Fig. 10. Model divergence from the CUDA model of TeaLeaf

In this section, we consider the different cost of divergence when porting a model from CUDA to other offload models, instead of starting from a serial model. We use the same TeaLeaf data obtained from Fig. 5 to conduct this case study. Fig. 9 shows the divergence of offload TeaLeaf models when starting from a serial codebase. Likewise, Fig. 10 shows the divergence starting from a CUDA codebase.

The divergence when starting from serial is lower when compared to starting from CUDA. This is most obviously seen with the T_{sem} metric and not the perceived metrics. This suggests that CUDA already encoded a set of semantics that differ from that of other models. Intuitively, it is common for CUDA programs to include platform-specific logics such as warp-size, or even use features that depend on independent thread scheduling. Such features or constraints are not always implemented or presented on another platform's hardware. This impedance is reflected in the higher T_{sem} for porting from CUDA.

The OpenMP target model stands out as having the lowest divergence overall when ported from serial, and the second lowest for T_{sem} and T_{src} when ported from CUDA. We conjecture that there may even be a desirable path where first porting to a model with less divergence, and then porting from that to the intended model, leads to an overall improvement in productivity. We previously noted that OpenMP describes the parallel semantics in a comparatively concise way in terms of code divergence, and so this may provide a stepping stone to first state the parallel semantics of the code using a non-verbose model.

VI. Combined Performance Portability and Productivity

Our model divergence metric presented is only concerned with the productivity of the codebase artefact. This section proposes a simple yet effective way of enhancing performance portability with the model divergence metric.

First, we run TeaLeaf and CloverLeaf on a diverse set of hardware platforms shown in Table III.

TABLE III
Platform details for Ᵽ benchmarks

Vendor	Name	Abbr.	Platform Topology
Intel	Xeon Platinum 8468	SPR	8 nodes (32C*2)
AMD	EPYC 7713	Milan	8 nodes (64C*2)
AWS	Graviton 3e	G3e	8 nodes (64C*1)
NVIDIA	Tesla H100 (SXM 80GB)	H100	2 nodes (4 GPUs)
AMD	Instinct MI250X	MI250X	2 nodes (4 GPUs)
Intel	Data Center GPU Max 1550	PVC	1 node (4 GPUs*)

The benchmarks are run using all available compilers for each platform. Where more than one compiler exists for each model, we compile our benchmark with each and only use the best performing result. For CloverLeaf, we use the BM64 deck at 300 iterations over 4 MPI ranks. For TeaLeaf, we use the BM5 deck at 4 steps over 4 MPI ranks. Fig. 11 and Fig. 12 shows our performance results in the classic cascade plot, as proposed by Sewall et al. [24].

Interpreting the Ᵽ bar chart on the right might tell us that these models are all quite similar. If we are interested in specific platforms, we may look at the cascade plot and observe where the performance drops. However, it is hard to draw a concrete recommendation without the productivity dimension.

To this end, we plot Ᵽ against our model divergence metric to form a navigation chart, as shown in Fig. 13, and Fig. 14. This chart is similar to and inspired by the navigation chart proposed by Pennycook et al. [6]. We chose to include only two of our tree metrics, T_{sem} and T_{src}, to avoid overcrowding the chart. As the two metrics represent the same model, we draw a line to create the association.

In general, the ideal model is located in the top right quadrant, where it shares proximity to the serial model and has good performance portability. Models that are not portable (e.g., a Ᵽ of zero) are still plotted on the chart as the

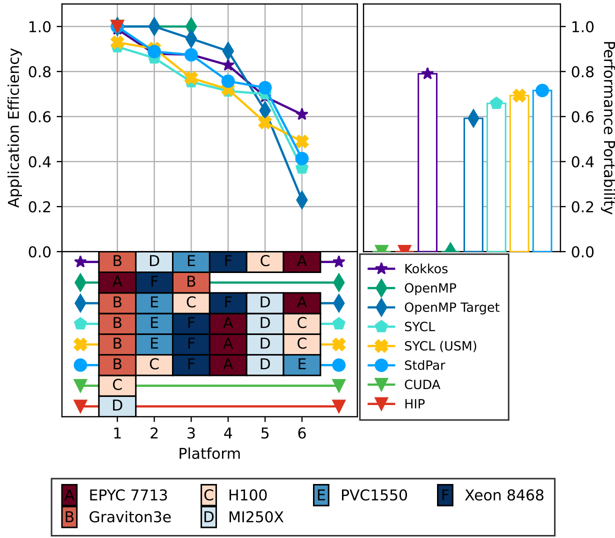

Fig. 11. TeaLeaf cascade plot, showing performance portability on six platforms

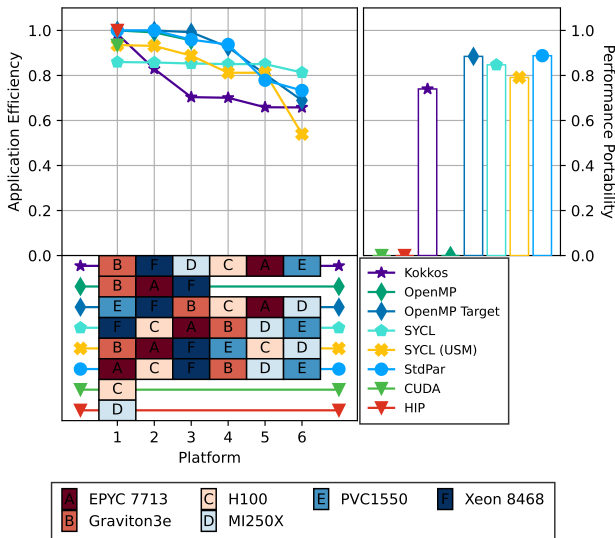

Fig. 12. CloverLeaf cascade plot, showing performance portability on six platforms

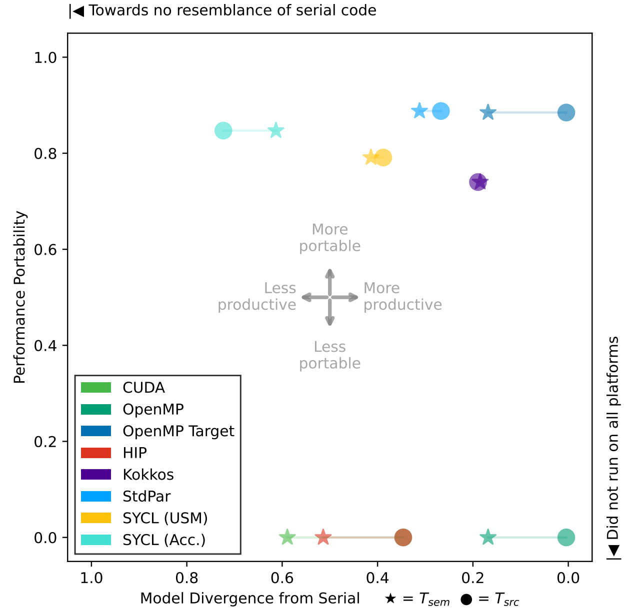

Fig. 13. CloverLeaf navigation chart of Ƥ and TBMD

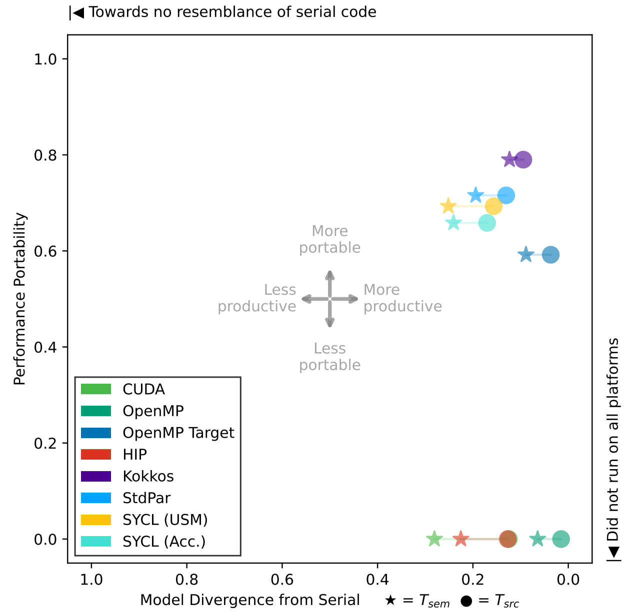

Fig. 14. TeaLeaf navigation chart of Ƥ and TBMD

divergence is unaffected by Ƥ. Overall, the navigation chart visualises the tradeoff of performance and productivity in a self-contained way. For example, with CloverLeaf, the accessor variant of SYCL is not a particularly productive model compared to the rest. However, it exhibited slightly better performance than the SYCL USM variant, possibly due to explicit memory movements that accessors encode.

The distance between T_{sem} and T_{src}, as illustrated using connected $\star$ and $\bullet$, gives interesting insight on the perceived and semantic similarity to the serial version. Consider the OpenMP and Kokkos model in Fig. 13, where the value of T_{sem} for OpenMP target is close to T_{sem} and T_{src} of Kokkos. We can interpret this as: the OpenMP model encodes similar levels of semantic complexity to Kokkos while accomplishing this with near zero cost at the source (T_{src}) level.

Another interesting example is the SYCL accessor model variant in Fig. 13: unlike other models where the source is usually perceived as more similar to serial (T_{src} is closer

to zero), the excessive accessor for SYCL buffers made the source appear much more complex than it is semantically.

Finally, note that the patterns of T_{src} and T_{sem} are application-specific, but the ordering is similar between Fig. 13 and Fig. 14.

VII. Future work

The SilverVale framework is designed with interfaces that enable support of new programming languages in mind. However, to limit scope for this study, we only explored Fortran and C/C++.

In the future, we would like to add support for established languages such as Python, and also emerging ones like Rust and Julia. For managed languages (i.e. languages that require a runtime environment for JIT or GC) like Julia and Python, there are additional complexities when approaching the S_{ir} level. Exploring how we can capture these appropriately will be an interesting challenge.

We would also like to expand the study to include full production applications that have different ports. For example, the computation kernels of GROMACS and Blender both have implementation in multiple programming models. At its current stage, SilverVale is focused primarily on correctness. One reason why production-ready applications were not included in this paper is due to the unoptimised memory usage while executing TED. For example, we were only able to do a short and incomplete divergence run of GROMACS's SYCL and CUDA port but had to exclude OpenMP due to limited memory on our workstations. In principle, the memory used during TED can and should be compressed on the fly.

Our future work will focus on improving the performance of SilverVale so that more comprehensive and meaningful experiments can be conducted.

VIII. Conclusion

In this paper, we proposed a new productivity metric, TBMD, based on the tree divergence from a serial model of the codebase. We designed a novel tree representation to summarise a codebase while retaining semantic information. To compare trees, we used Tree Edit Distance (TED) for the basis of divergence. And finally, we implemented an end-to-end framework, SilverVale[8], to carry out analysis of codebases systematically.

Our evaluation of TBMD has revealed several interesting properties. For example, we found that OpenMP directives in both Clang and GCC are represented as unique AST tokens that posses semantic information above the laws of the host language. We were also able to show that code migration is highly dependent on the original model used. This was done with an example where migrating from CUDA to other offload models may be less productive than porting from a serial one. In general, we find declarative models such as OpenMP and StdPar tend to have a lower divergence from serial when compared to the rest. In essence, declarative

[8]https://github.com/UoB-HPC/SilverVale

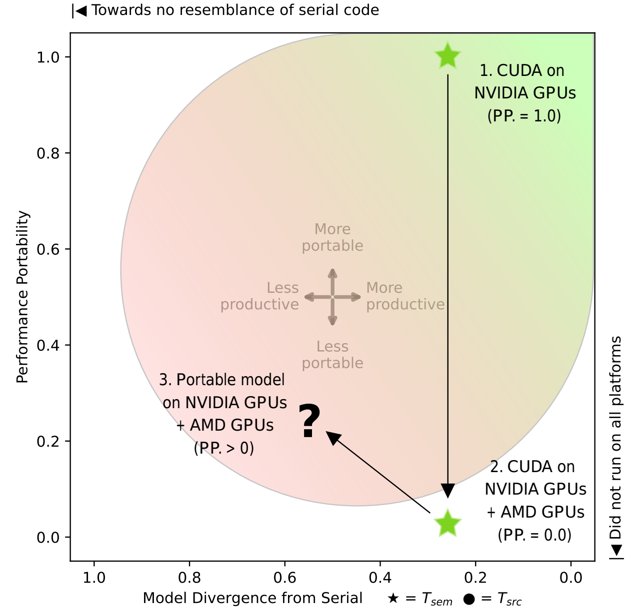

Fig. 15. Navigation chart on for picking the right model, starting from an unportable one

parallelism provides an encoding that helps preserve intent; the constraints are unobtrusive and deals less damage to the original serial semantic. This result suggests a workflow where porting serial code to a declarative model first and then using backend interop features (i.e., escape hatch) to fill performance gaps may be a viable option.

Throughout the result analysis, we find our TBMD metric generally agrees with anecdotal reports from developers that have contributed to the mini-apps. The perceived and semantic divergence result themselves aligns well with intuition. When we compare the semantic (T_{sem}) and perceived (T_{src}) metrics, we observe valuable insight such as potential model bloat where the perceived divergence from serial is higher than that of semantic divergence. In the same vein, we also observe declarative models like OpenMP sharing similar semantic complexity to imperative models but with very little perceived complexity. Based on the results analysing five mini-apps, we believe collecting and comparing T_{sem} and T_{src} is the most appropriate application TBMD.

Finally, we combine our metric with performance portability and propose an enhanced navigation chart. We conducted heterogeneous benchmarks using CloverLeaf and TeaLeaf and presented a navigation chart result highlighting tradeoffs of the eight different programming models and variants.

Fig. 15 presents a common HPC codebase scenario where vendor diversification has led to a need for more portability. The figure starts with the data point 1 on the top right, where a codebase is initially designed to use CUDA exclusively because NVIDIA GPUs is the only GPGPU platform at the time. Such a codebase has a Φ of one because only a single platform existed. As time moves on, AMD introduces

competitive GPUs to the market, and so we add a second platform which causes the Φ to drop to zero: we are at data point 2 where the original CUDA codebase is not directly portable to the HIP platform. The navigation chart, when augmented with past results, can be used to select which model may be more suitable for the codebase, thus helping us land on a better location for data point 3. We hope this method of visualisation provides an easy way to help evaluate tradeoffs between productivity and portability for both new and existing projects.

ACKNOWLEDGEMENT

We would like to thank John Pennycook for his valuable input and enlightening discussion on the topic related to model divergence. Additionally, we would also like to thank Jamie Wills, Johannes Doerfert, and Aksel Alpay, for their positive and constructive feedback on the overall design of the metric. Some benchmark results for performance portability is collected with the assistance of Gonzalo Brito and Aksel Alpay.

This work used the Isambard (http://gw4.ac.uk/isambard/) UK National Tier-2 HPC Service operated by GW4 and the UK Met Office, and funded by EPSRC (EP/P020224/1). This work was performed using resources provided by the Cambridge Service for Data Driven Discovery (CSD3) operated by the University of Cambridge Research Computing Service (www.csd3.cam.ac.uk), provided by Dell EMC and Intel using Tier-2 funding from the Engineering and Physical Sciences Research Council (capital grant EP/T022159/1), and DiRAC funding from the Science and Technology Facilities Council (www.dirac.ac.uk). DiRAC is part of the National e-Infrastructure. The University of Bristol is an Intel oneAPI Center of Excellence, which helped support this work. This project has received funding from the European Union's HE research and innovation programme under grant agreement No 101092877. Some experiments in this paper used resources from NVIDIA's Selene and pre-EOS Supercomputers. We are extremely grateful to AWS for supporting access to Graviton 3e.

REFERENCES

[1] S. J. Pennycook, J. D. Sewall, and V. W. Lee, "A Metric for Performance Portability," 2016. [Online]. Available: https://arxiv.org/abs/1611.07409

[2] S. L. Harrell, J. Kitson, R. Bird, S. J. Pennycook, J. Sewall, D. Jacobsen, D. N. Asanza, A. Hsu, H. C. Carrillo, H. Kim, and R. Robey, "Effective Performance Portability," in *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2018, pp. 24–36.

[3] Y. Faqir-Rhazoui and C. García, "Exploring the performance and portability of the k-means algorithm on SYCL across CPU and GPU architectures," *J. Supercomput.*, vol. 79, no. 16, pp. 18 480–18 506, Nov. 2023.

[4] Y. Asahi, T. Padioleau, G. Latu, J. Bigot, V. Grandgirard, and K. Obrejan, "Performance portable vlasov code with c++ parallel algorithm," in *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2022, pp. 68–80.

[5] S. Wienke, "Productivity and software development effort estimation in high-performance computing," Ph.D. dissertation, Dissertation, RWTH Aachen University, 2017, 2017.

[6] S. J. Pennycook, J. D. Sewall, D. W. Jacobsen, T. Deakin, and S. McIntosh-Smith, "Navigating Performance, Portability, and Productivity," *Computing in Science & Engineering*, vol. 23, no. 5, pp. 28–38, 2021.

[7] P. E. McKenney, "Is Parallel Programming Hard, And, If So, What Can You Do About It? (Release v2023.06.11a)," 2023. [Online]. Available: https://arxiv.org/abs/1701.00854

[8] W.-C. Lin, T. Deakin, and S. McIntosh-Smith, "Evaluating ISO C++ Parallel Algorithms on Heterogeneous HPC Systems," in *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2022, pp. 36–47.

[9] A. J. Offutt, M. J. Harrold, and P. Kolte, "A software metric system for module coupling," *Journal of Systems and Software*, vol. 20, no. 3, pp. 295–308, 1993.

[10] P. Bille, "A survey on tree edit distance and related problems," *Theoretical computer science*, vol. 337, no. 1-3, pp. 217–239, 2005.

[11] V. Nguyen, S. Deeds-Rubin, T. Tan, and B. Boehm, "A SLOC counting standard," in *Cocomo ii forum*, vol. 2007. Citeseer, 2007, pp. 1–16.

[12] T. Clem and P. Thomson, "Static Analysis at GitHub: An experience report," *Queue*, vol. 19, no. 4, p. 4267, sep 2021. [Online]. Available: https://doi.org/10.1145/3487019.3487022

[13] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," *SIAM journal on computing*, vol. 18, no. 6, pp. 1245–1262, 1989.

[14] M. Pawlik and N. Augsten, "Tree edit distance: Robust and memory-efficient," *Information Systems*, vol. 56, pp. 157–173, 2016.

[15] ——, "tree-similarity," https://github.com/DatabaseGroup/tree-similarity/, 2024.

[16] S. Wu, U. Manber, G. Myers, and W. Miller, "An O (NP) sequence comparison algorithm," *Information Processing Letters*, vol. 35, no. 6, pp. 317–323, 1990.

[17] T. Kubo, "dtl," https://github.com/cubicdaiya/dtl, 2024.

[18] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, "Evaluating attainable memory bandwidth of parallel programming models via BabelStream," *International Journal of Computational Science and Engineering*, vol. 17, no. 3, pp. 247–262, 2018.

[19] J. R. Hammond, T. Deakin, J. Cownie, and S. McIntosh-Smith, "Benchmarking Fortran DO CONCURRENT on CPUs and GPUs Using BabelStream," in *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2022, pp. 82–99.

[20] A. Poenaru, W.-C. Lin, and S. McIntosh-Smith, "A Performance Analysis of Modern Parallel Programming Models Using a Compute-Bound Application," in *High Performance Computing*, B. L. Chamberlain, A.-L. Varbanescu, H. Ltaief, and P. Luszczek, Eds. Cham: Springer International Publishing, 2021, pp. 332–350.

[21] J. Li, A. Bobyr, S. Boehm, W. Brantley, H. Brunst, A. Cavelan, S. Chandrasekaran, J. Cheng, F. M. Ciorba, M. Colgrove, T. Curtis, C. Daley, M. Ferrato, M. G. de Souza, N. Hagerty, R. Henschel, G. Juckeland, J. Kelling, K. Li, R. Lieberman, K. McMahon, E. Melnichenko, M. A. Neggaz, H. Ono, C. Ponder, D. Raddatz, S. Schueller, R. Searles, F. Vasilev, V. M. Vergara, B. Wang, B. Wesarg, S. Wienke, and M. Zavala, "SPEChpc 2021 Benchmark Suites for Modern HPC Systems," in *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1516. [Online]. Available: https://doi.org/10.1145/3491204.3527498

[22] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for multi-core processor parallelism*. OReilly, 2010.

[23] A. Alpay, B. Soproni, H. Wünsche, and V. Heuveline, "Exploring the possibility of a hipsycl-based implementation of oneapi," in *Proceedings of the 10th International Workshop on OpenCL*, ser. IWOCL '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3529538.3530005

[24] J. Sewall, S. J. Pennycook, D. Jacobsen, T. Deakin, and S. McIntosh-Smith, "Interpreting and visualizing performance portability metrics," in *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, Nov. 2020.

Appendix

Artifact Description

We ran BabelStream, miniBUDE, TeaLeaf, and CloverLeaf on a wide range of hardware platforms listed in Table III. We created a SilverVale Codebase DB using all supported models of these mini-apps.

Artifacts Available: The source code for the mini-apps used in all experiments is available at:

- BabelStream - https://github.com/UoB-HPC/BabelStream
- miniBUDE - https://github.com/UoB-HPC/miniBUDE
- TeaLeaf - https://github.com/UoB-HPC/TeaLeaf
- CloverLeaf - https://github.com/UoB-HPC/CloverLeaf

The SilverVale software introduced in this paper is open source software, available at https://github.com/UoB-HPC/SilverVale. The project homepage contains comprehensive documentation on the usage of the tool.

Experimental setup: SeeTable III for a list of hardware platforms used. We performed the indexing step of codebases using SilverVale on all supported models of BabelStream, miniBUDE, TeaLeaf, and CloverLeaf. The complete experiment setup is available at https://github.com/UoB-HPC/SilverVale/p3hpc.

Artefact Evaluation

Performed verification and validation studies: Each mini-app contains built-in verification for correctness. At runtime, SilverVale compares the base model against itself; non-zero results will indicate an error in the implementation. SilverVale also contains unit and integration tests.

Validated the accuracy and precision of timings: N/A

Used manufactured solutions or spectral properties: N/A

Quantified the sensitivity of your results to initial conditions and/or parameters of the computational environment: N/A

Describe controls, statistics, or other steps taken to make the measurements and analyses robust to variability and unknowns in the system: N/A