Applying the Midas Touch of Reproducibility to High-Performance Computing

A. C. Minor
Dept. of Computer Science
Virginia Tech
acminor@vt.edu

W. Feng
Dept. of Computer Science
Virginia Tech
wfeng@vt.edu

***Abstract*—With the serial performance of CPUs improving exponentially through the 1980s and 1990s and then plateauing by the mid-2000s, the high-performance computing community has seen parallel computing become ubiquitous, which, in turn, has led to a proliferation of parallel programming models. This diversity in hardware platform and programming model has forced programmers to port their codes from one hardware platform to another (e.g., CUDA on Nvidia GPU to OpenCL on AMD GPU) and show reproducibility via ad-hoc testing. So, to validate reproducibility between codes, we propose Midas, a system to ensure that the results of the original code match the results of the ported code by leveraging snapshots to capture the state of a system before and after the execution of a kernel.**

***Index Terms*—heterogeneous computing, parallel computing, programming models, programming languages, reproducibility, snapshots, validation.**

I. Introduction

Today, heterogeneous computing is at a crossroads. Parallel computing has become ubiquitous with multicore CPUs, manycore GPUs and TPUs, and even FPGAs becoming the norm. Unfortunately, with this proliferation of parallel computing platforms comes a variety of parallel programming models, most of which are *not* cross-platform compatible.

As the high-performance computing (HPC) community updates their facilities, this lack of cross-platform compatibility becomes an issue, leading to vendor lock-in, decreased productivity, and ad-hoc testing of ported code.[1] When organizations change vendors and, in turn, parallel computing platforms, existing code needs to be rewritten or ported to whatever language the new platform supports.

CU2CL [1] facilitated the automated porting of code from CUDA to OpenCL, but reproducibility was addressed via ad-hoc testing only [2]. Likewise, AMD created a CUDA-like environment called HIP, short for Heterogeneous Interface for Portability, to enable porting code from Nvidia to AMD platforms [3], using a tool called HIPify to convert CUDA code to HIP code [4]. In the meantime, Intel created oneAPI, which leverages SYCL and has a translator from CUDA to SYCL called SYCLomatic [5].

[1]For example, DOE Oak Ridge National Laboratory's previous fastest supercomputer in the world, Summit, consisted of Nvidia GPUs programmed with CUDA, while their latest fastest supercomputer in the world, Frontier, consists of AMD GPUs programmed with HIP or OpenCL.

Rather than focus on which language to standardize upon, we seek to address the "hidden elephant" in the room, namely the need to rigorously validate the cross-platform porting of code, which is aided by tools such as CU2CL, HIPify, and SYCLomatic. However, without rigorous testing or with only full-suite integration testing, the validation of large-scale HPC codes is suspect, particularly when results are non-deterministic, as the integration test might only test stochastic equivalence. Consequently, tests must be finer-grained, allowing for more deterministic testing and precision and benefiting the porting of large-scale HPC codes by automatically identifying where issues occur with finer-grained testing and preventing potentially unexpected results that might be smoothed over by a broad system-level test.

To address the issue of reproducibility, we propose Midas, as depicted in Fig. 1. Midas is a system to ensure that the results of the original code match the results of ported code. It accomplishes this task by leveraging snapshots to capture the state of a system before and after the execution of a kernel. These snapshots can then be used to unit test the ported code against the original code. By leveraging a snapshot-based test flow, porting and testing can be made precise.

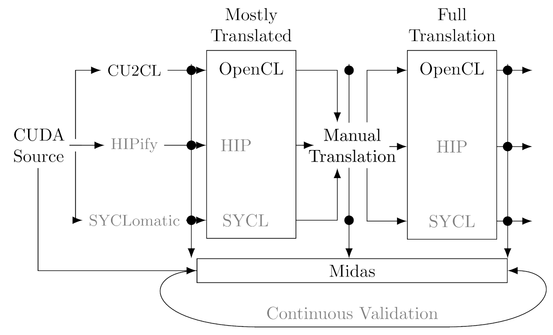

Fig. 1: Midas: Validating the Porting of GPU Source Code

II. Snapshot Testing as a Workflow

Using snapshots, we identify three types of translation encountered in porting code:

- *Porting:* Raw translation
- *Interpretation:* Feature parity issues (e.g., queues, locks, ...)
- *Refinement:* Fitting target xPU, where $x = \{C, G, T, ...\}$

First, we define *porting* as the process of translation between two different programming environments — in this paper, CUDA to OpenCL, as our case study. To facilitate the porting of large-scale HPC codes, programmers use source-to-source translation tools, like CU2CL [1]. Once translated, rigorous validation testing is needed to ensure reproducibility of the ported code, as source-to-source translation is non-trivial and has many corner cases. So, we seek to automatically probe these corner cases of language translation and validate results.

Snapshotting provides tools to validate the results of source-to-source translation. Fig. 2 shows an example of a software process for testing the porting effort.[2]

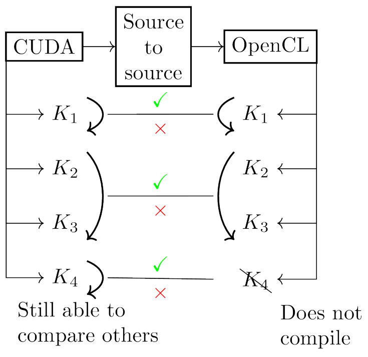

Fig. 2: Porting Workflow.
K_i represents the i^{th} kernel in the program

Second, we define *interpretation* as the creation or re-definition of GPU framework features to deal with the lack of feature parity between two programming frameworks. For example, perhaps a source-to-source translation framework can replace queues in one language with library calls to a custom queue in another language. However, due to framework differences, this might not be possible. Thus, interpretation is necessary to change the way that the code is written but to behave the same while fitting in the target framework.

Because source-to-source translators generally do not address the problem of interpretation, we rely on the existence of "stubs" to have points of insertion or interpretation for the code constructs and missing features. These stubs allow code to run without stopping and without failing to compile. However, the results will not be the same, as functionality will be missing.

Snapshotting helps validate these issues in three ways: (1) We can sidestep the issue of infinite looping and broken code due to missing features. With a snapshot, we can unit test kernels individually. Thus, we can test all working kernels and assign members to fix any apparent issues. (2) We can address the issue of filling in stubs. Some language features might be missing a library. Thus, we can fill in stubs with another library, handwritten or third-party, to match the features needed for the code in question. The kernels using these stubs can then be addressed one by one in parallel. (3) In addressing larger language differences (e.g., absence of locks), we can rework multiple kernels with confidence that our final results are the same. In the case that two or more kernels must be fused or separated, we can validate units of work instead of kernels. Fig. 3 shows the general process of interpretation workflows.

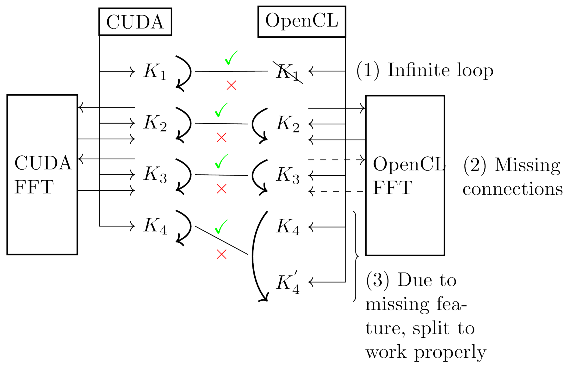

Fig. 3: Interpretation Workflow.
K_i represents the i^{th} kernel in the program

Third, we define *refinement* to be the process of adapting code to fit the programming framework or environment of the newly ported code. It can also be defined as the process of incrementally improving a code's speed or readability. This closely matches what software engineers refer to as refactoring and can be illustrated by tuning the optimal execution dimensions for a kernel or tweaking data structures to map on the cache architecture of the target hardware. With snapshot testing, not only can we continuously improve and refactor, but we can also validate our changes on a kernel-by-kernel and integration-test level. Furthermore, we can profile each individual kernel's execution time, separate from the whole-program execution time. See Fig. 4 for a refactoring workflow.

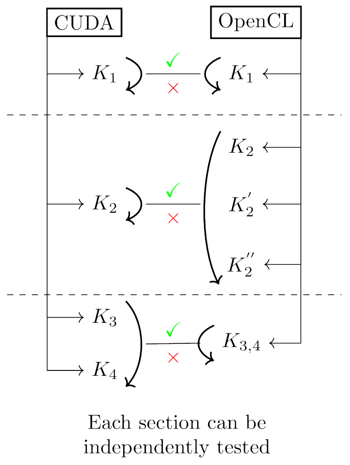

Fig. 4: Refactoring Workflow.
K_i represents the i^{th} kernel in the program

Snapshotting can be easily integrated into an existing testing framework. It is as simple as choosing the points of integration (where to validate pre-/post-conditions) and capturing the relevant data.

[2]For this and subsequent figures, CUDA and OpenCL are provided as a case-study example without loss of generality. They could be substituted with other frameworks, e.g., SYCL and HIP.

III. Midas

Midas is our framework for providing easy-to-use and performant snapshot-based testing for code porting. It consists of three tools that address each of the aforementioned translation tasks using software engineering to ease developer toil. Fig. 5 shows an image of the entire system.

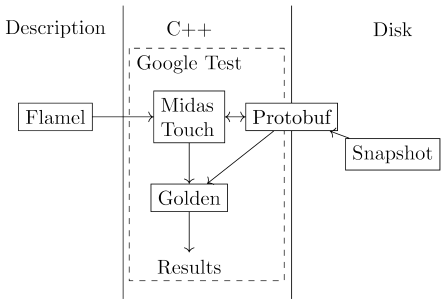

Fig. 5: Midas System Overview

Golden serves as the foundation of Midas. It reliably snapshots data to disk and compares the snapshots. Inspired by the C++ library Approval Tests [6] and the concept of "goldens," goldens are a snapshot of what your data (input or output) should be for a given function. Current snapshots and goldens can then be compared to determine if your cross-platform program has changed its behavior. Fig. 6 shows the Golden workflow, i.e., save a snapshot from a verified source to disk and then use it to validate future snapshots.

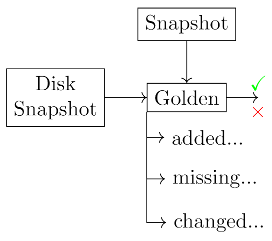

Fig. 6: Golden Workflow

Approval Tests as a framework behaves similarly; however, it has a major shortcoming for HPC workloads: it mainly works with text data, though you can write your own writer [7]. This shortcoming results in two major ramifications. First, floating-point numbers must be rendered consistently to text. Because we are comparing text, we cannot rely on epsilon-based equality schemes; instead we need a consistent truncating-based design. Second, due to the conversion of floating-point numbers to strings, we incur both a performance penalty and a memory penalty. The performance penalty is obvious. The memory penalty can be understood with the following reasoning. Suppose we want to compare a floating-point number to eight decimal-place precision. This implies up to eight characters in the text output plus the leading magnitude and the decimal position character, resulting in more than eight bytes. Now consider the floating-point representations for `float` and `double`, four bytes and eight bytes, respectively. We are guaranteed the same or better performance for eight bytes and can better handle the separation of data elements by only requiring the need for a size (as all floating-points are equal width). This analysis can be seen in Fig. 7.

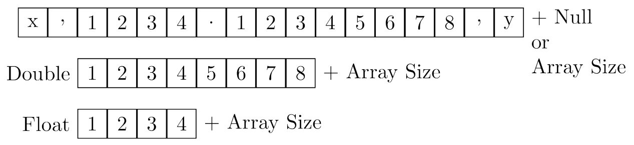

Fig. 7: Storage Size between Storage Types

To accomplish this non-text-based snapshotting scheme, we use Google Protocol Buffers (i.e., protobuf) [8] to handle the representation of snapshots. With protobuf, you focus on specifying the different members of a snapshot (type, repetition, etc.), see Listing 1. Protobuf serializes this efficiently, using a binary format [9]. By using a well-established library, we get assurance of its future sustainability and its current validity. Furthermore, protobuf has the ability to introspect and compare its messages to each other. This can be used to report missing members, added members, and non-matching members for a message [10], as noted in Fig. 6. The end user can then take this report and analyze what possible issues might be occurring (i.e., logic bugs, epsilon differences, missing fields, and so on).

```
message ReducePE {
// calculates kernel launch dims based
// off this and a macro constant
int32 natom = 1;
repeated lib.pb_float4 f4d = 2;
repeated lib.pb_float4 f4d_nonbond = 3;
repeated lib.pb_float4 f4d_bonded = 4;
int32 natomd = 5;
int32 workgroup_sized = 6;
}
```

Listing 1: Protobuf Snapshot Description for the FenZi Molecular Dynamics Code

To use Golden directly, snapshots must be handmade for each unit of work that we want to test. Because this process is tedious and error-prone, we created Midas Touch, an interface and collection of converters to handle common serialization tasks. If needed, users can write custom converters using the provided Midas Touch interfaces. Table I provides a listing of the implemented converters.

Converter	Description
Dim3	OpenCL and CUDA
Float2,3,4	OpenCL and CUDA
Complex	CUDA, OpenCL complex handled by Float2
Int2,4	OpenCL and CUDA
Primitive	OpenCL and CUDA, raw ints, floats, structs, etc.
Vector	OpenCL and CUDA
MallocInteropArray	OpenCL and CUDA, uses tracking logic to determine array size

TABLE I: Midas Touch Converters

Using Midas Touch, we can focus directly on what we are serializing instead of how we are serializing it. For example,

```
using
  Converters::MallocInteropArrayConverter;
using Converters::PrimitiveConverter;
auto snapshot = some_protobuf_snapshot();
MallocInteropArrayConverter.Serialize(
  someFloatArray,
  [&](const auto &x) {
    PrimitiveConverter.Serialize(
      &x, [&](const auto &x) {
          snapshot.add_some_float_array(x);
      },
      make_options<MemoryOptions::Host>());
  },
  make_options<MemoryOptions::Device>());
VectorConverter.Deserialize(
  &someFloatArray,
  snapshot.some_float_array(),
  make_options<MemoryOptions::Device>(
    FilledConverter(
        PrimitiveConverter,
        make_options<
          MemoryOptions::Host>())));
```

Listing 2: Example Midas Touch Source Code

suppose we need to serialize an array of floats. Instead of writing a for-loop along with the associated code to copy data to and from the GPU properly, we can use Midas Touch and simply say "dump this array from GPU to host." This is highly convenient when used with our allocation-snooping allocator override. This allows source code to have the size of GPU allocations snooped, using a simple C preprocessor macro. Thus, when the data is serialized to disk, we do not have to specify the size. This dump to array logic, along with the corresponding deserialization code, can be seen in Listing 2.

Midas Touch is a generic framework that only handles the serialization and deserialization parts of a snapshot. This gives programmers the freedom to design their serialization code in the best way that fits their system. For example, we made explicit serialization and deserialization functions that could then be used with other code to only serialize the subsequent calls of a kernel. Looking at our previous discussion on splitting and merging snapshots, the benefit of this generic interface is apparent. For example, we can take two snapshots' serialization functions and combine them into one to simulate a merged kernel. Examples of the freedom in Midas Touch's interface can be seen in Fig. 8.

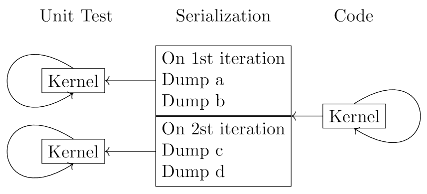

Fig. 8: Midas Touch: Freedom of Snapshotting

With Midas Touch being verbose, so as to provide a complete interface, we propose a third tool called Flamel that enables the programmer to focus on what is being serialized.

Flamel is a high-level description language to generate Midas Touch code. Flamel abstracts the descriptions for fields of a snapshot to a TOML interface. TOML [11] is a data-description language, similar to JSON. In our usage, we specify metadata for a snapshot and do not worry about the Midas Touch code that is generated for that snapshot. In the end, we get a description of a snapshot that can be used to generate C++ functions for serialization and deserialization. This abstraction allows for future replacement of the underlying library code. For example, we might want to test Python code and could implement code to generate the serialization code in Python from our Flamel TOML descriptions. Furthermore, we can store our descriptions in a database to analyze how our snapshots change over time, instead of how our code changes over time. Ultimately, researchers and others can more easily generate snapshot code for their given problem, while providing for greater introspection into the changes that occur in a snapshot over time. See Fig. 9 for how Flamel could integrate into such a system.

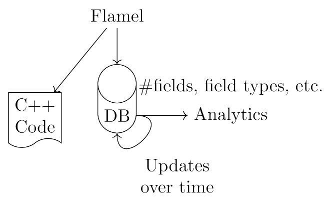

Fig. 9: Potential Flamel Workflow

IV. Results

With Midas, users can incorporate a snapshot-based testing workflow into their original testing workflow or into a new testing workflow. In our case, we created a rigorous testing framework using our snapshot-based testing and GoogleTest [12].

As a case study, we use FenZi, a molecular dynamics code written in CUDA [13] and then translated to OpenCL [2] via CU2CL [1]. While we could not illustrate Midas's incremental porting benefits, as FenZi was already ported, we can show the validation of the porting done at a fine granularity. Due to the way that FenZi executes, the precise solution to the problem is non-deterministic; thus, we could only validate the code stochastically. However, analyzing the source code revealed that virtually all the kernels could be tested deterministically. Thus, using Midas at this point in the porting process, we abstracted away the whole-program non-determinism to show that the individual kernels themselves behave similarly, without needing to be concerned with how the outputs might change between runs.

Using Midas, we validated the kernels in the FenZi code base. Furthermore, the CUDA source code and OpenCL source code used varying numbers of parameters (e.g., up to 39 parameters for the kernel BondedForce) and slightly different interfaces (e.g., switching interfaces for memory accesses between the two ports (e.g., using or not using texture memory). Through this validation process, we detected one minor error in the ported code (but not in the GPU kernels) related to accidentally using previous global state. This error only showed up in isolation and was benign in the overall program runs. In addition, we found a minuscule difference in the epsilon results for a few kernels. For reproducibility, we have provided our tests and our code at `https://github.com/vtsynergy/midas`. A listing of the kernels tested can be seen in Table II.

Kernels	
BCMultiply	BondedForce
CellBuild	CellClean
CellUpdate	ChargeSpreadMedium
ChargeSpreadSmall	CheckCellOccupancy
CheckLatticeNum	CheckNonbondNum
ConjugatedGradient	CoordsUpdate
HalfKickGPU	LatticeBuild
nbbuild_exclbitvec	NonBondForce
PMEForceLarge	PMEForceMedium
ReduceCOM	ReducePE
RestraintForce	SolveBondConstraints
SolveVelocityConstraints	UpdateCoords

TABLE II: Tested Kernels

V. Challenges

In examining our snapshot testing for FenZi, we observed several challenges.

Non-determinism can change snapshots between runs. If the change was due to a bug across platform ports, we can validate the said change and update the snapshot or port; but if it is due to non-determinism, then the snapshot will continuously change, whether a bug was introduced or fixed. For instance, one source of non-determinism in parallel HPC code is atomic operations across threads.

While we have encountered several non-deterministic kernels, they have either used atomic operations that perform a commutative operation on a result (e.g., reduction-style workflow, as shown in Fig. 10) or that perform a reversible operation. (By reversible, we mean that the operation can be made deterministic by removing and/or imposing a theoretically valid ordering.) Thus, we have been able to avoid the need to skip or to only partially test the non-deterministic kernels.

Fig. 11 shows an example of a reversible kernel. For this kernel, we are processing a set of three-dimensional (3D) arrays. If an array cell has been previously updated, we move to the next array in the stack and process the same cell location. This forms a non-deterministic ordering of values along atomic time. Furthermore, for our kernel, the generated values are deterministic in value and contain the thread identification (i.e., id) that processed them. So, we have a non-deterministic ordering of deterministic values with thread id that can be used to impose an output ordering. That is, we sort array cells over time from smallest to largest thread id, which represents a possible execution ordering. Thus, we can assert equality of any two results via this deterministic transformation.

In addition, our system allows for small-scale testing which can minimize the size of untested non-deterministic parts of the code. For example, suppose that an individual kernel is non-deterministic, but a collection of kernels run together converges toward a single value. We can snapshot and compare at the convergence point. This convergent point could be an entire run of the system, or it could be subsets of the run, as shown in Fig. 12. On the other hand, suppose that no convergence exists in the output but only a small subset of kernels is non-deterministic. We can still test a large number of the deterministic kernels accurately and fully while leaving the few non-deterministic kernels for other analyses, as shown in Fig. 12.

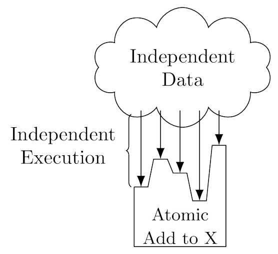

Fig. 10: Commutative Reduction Example

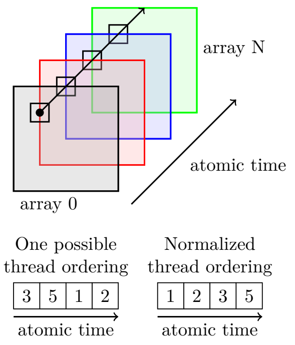

Fig. 11: Reversible Non-Determinism Example

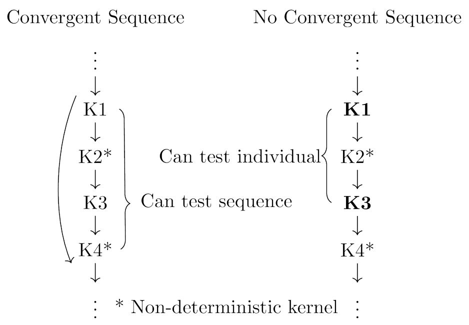

Fig. 12: Non-deterministic Testing Workflow. K_i represents the i^{th} kernel in the program

VI. Future Work

Midas is a comprehensive reproducibility system, made for adapting to today's gamut of parallel programming models and underlying parallel hardware. In the future, we seek to further improve Midas and the porting workflow presented here. For example, experimentally, we would like to have two teams port a code base between two GPU platforms, i.e., have one team use Midas while the other team does not. The port would use current tools and technologies for porting to demonstrate Midas's benefits in a modern workflow. The teams should be analyzed for completeness, bugs, speed of development, and ease of use.

Further future work includes extending Midas to support SYCL, HIP, and other GPU languages and platforms. This would demonstrate Midas's ability to address current and future challenges in a repidly-changing landscape of HPC systems. Along with this goal, we would like to see a port of Midas that works with FPGAs.

From a software engineering perspective, We want snapshot testing to be a platform usable by anyone. However, additional work needs to be invested in determining the best interfaces for doing so. Currently, we envision a study of the Flamel snapshot description language and code generation. What is the minimal subset needed? What extension points are needed to fit current and future demands? What is the best code generation platform to permit for the ease of adding custom code generation?

Finally, based on our inspiration from Approval Tests, if a snapshot does not match, a `diff` would be generated, which can be used to see the differences between the golden snapshot and the current snapshot [14]. In GPU codes, the amount of data generated can be rather large. Furthermore, it is possible that a small change can result in a large number of differences. Thus, a text `diff` of all the number differences might be a wasted effort to analyze. We hope for more studies to be conducted on different visualizations of these `diffs`. For example, a simple `diff` visualization might be to plot the golden and current snapshot data and compare the results that way. Alternatively, we could plot the frequency plot of the data or conduct an analysis of the statistics of the data we are analyzing. Which diff types are most useful is going to be domain-specific and an investigation, as such, would need to look at different, particular domains for ideas.

VII. Conclusion

Snapshot testing addresses raw translation, interpretation, and refinement during the porting process. Our Midas artifact is a comprehensive snapshot-testing framework for testing compute kernels efficiently, accurately, and easily. It is generic and can be integrated into existing workflows. To test our design, we validated FenZi CUDA [13] to FenZi OpenCL [2] showing that the code was indeed operating the same.

We would like to acknowledge similar work. Abramson and Sosic in "A Debugging and Testing Tool for Supporting Software Evolution" [15] designed a system for verifying ports across platforms including serialization issues such as big vs little-endian. Their work used debugging infrastructure, networks, and visualization to verify and find bugs in cross-platform high performance computing code. They also supported a version of serialization for testing. They did not address GPU systems and their use of debugging infrastructure could be problematic with more advance optimizations, including code reordering.

In this new world of transitioning HPC resources and source-to-source translators, we hope our contribution of an understanding of how snapshot-based testing can contribute to GPU porting efforts along with a case study for FenZi CUDA and FenZi OpenCL helps lead the way in validating ported code. Non-determinism in GPU computing is a difficult topic to address, we showed that it is possible in some circumstances to continue to test in spite of non-deterministic code.

References

[1] G. Martinez, M. Gardner, and W. Feng, "CU2CL: A CUDA-to-OpenCL Translator for Multi- and Many-Core Architectures," in *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, 2011, pp. 300–307.

[2] P. Sathre, M. Gardner, and W. Feng, "On the Portability of CPU-Accelerated Applications via Automated Source-to-Source Translation," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, ser. HPC Asia 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 1–8. [Online]. Available: https://doi.org/10.1145/3293320.3293338

[3] HIP Programming Guide v4.5. https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-GUIDE.html.

[4] HIPIFY: Tools to translate CUDA source code into portable HIP C++ automatically. https://github.com/ROCm-Developer-Tools/HIPIFY.

[5] SYCLomatic. https://github.com/oneapi-src/SYCLomatic.

[6] Approval Tests. https://github.com/approvals/ApprovalTests.cpp.

[7] ApprovalTests.cpp: Writers. https://approvaltestscpp.readthedocs.io/en/latest/generated_docs/Writers.html.

[8] Protocol Buffers - Google's data interchange format. https://developers.google.com/protocol-buffers.

[9] (2022, 05) Protocol Buffers - Encoding. https://developers.google.com/protocol-buffers/docs/encoding.

[10] (2021, 05) message_differencer.h. https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.util.message_differencer.

[11] T. Preston-Werner, P. Gedam, et al. TOML: Tom's Obvious Minimal Language. https://github.com/toml-lang/toml.

[12] GoogleTest User's Guide. https://google.github.io/googletest/.

[13] N. Ganesan, M. Taufer, B. Bauer, and S. Patel, "FENZI: GPU-Enabled Molecular Dynamics Simulations of Large Membrane Regions Based on the CHARMM Force Field and PME," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, 2011, pp. 472–480.

[14] L. Falco. (2011, 12) Using Reporters in Approval Tests. https://blog.approvaltests.org/2011/12/using-reporters-in-approval-tests.html.

[15] D. Abramson and R. Sosic, "A Debugging and Testing Tool for Supporting Software Evolution," *Autom. Softw. Eng.*, vol. 3, pp. 369–390, 08 1996.