Halide Code Generation Framework in Phylanx

Abstract—**Separating algorithms from their computation schedule has become a de facto solution to tackle the challenges of developing high performance code on modern heterogeneous architectures. Common approaches include Domain-specific languages (DSLs) which provide APIs familiar to domain experts, code generation frameworks that automate the generation of fast and portable code, and runtime systems that manage threads for concurrency and parallelism. In this paper, we present an extension to Phylanx distributed array processing platform to support the Halide code generation framework. The extension introduces a new category of Phylanx primitives that are generated by Halide but run on HPX threads. To accomplish this, we partially export Halide thread pool API to carry out parallelism on HPX threads. In addition, we demonstrate the accompanying performance analysis pipeline which allows fine-grain measurement, collection, and visualization of the performance data. The easy-to-use profiling pipeline facilitates tuning of Halide applications including Phylanx functions. To evaluate our work, we first run existing Halide applications on both native and HPX runtimes verifying there is no cost associated with using HPX threads. Next, we re-implement a set of Phylanx primitives using Halide to benefit from the performance and portability of the generated code.**

Index Terms—**peformance, benchmark, Halide, HPX, DSL**

I. TODO

- Change the narrative from heterogeneity to programming abstractions(DSL, code generation, AMTs)
- Adding more experiments for machine learning
- Explaining the performance behavior
- Change performance analysis from time to throughput
- Introduce Phylanx earlier in the paper

II. INTRODUCTION

In recent years there has been a massive shift towards heterogeneous computing systems as impacts of Moore's law [1] and Dennard scaling [2] have been dwindling. As a consequence, high performance code has become increasingly more complex, expensive to maintain, and less portable. A common approach to address these issues is separating the algorithm from the scheduling of the computation, i.e., the order of computations and memory accesses. This has resulted in a growing interest in developing domain specific languages (DSLs), code generation frameworks, and runtime systems which all aim at such separation, albeit at different levels of abstraction:

Domain specific languages support high-level abstractions providing APIs close to domain nomenclature. DSLs benefit from the prior knowledge on the data characteristics and computational traits of applications in a particular domain to utilize optimizations that may not be valid in general.

DSLs could be implemented as an external DSL, which is a stand-alone language with custom semantics, and syntax, or as an internal DSL, embedded with a host language [3]. External DSLs, are not restricted to a host language providing a standalone execution environment, simplifying the code and improving expressiveness and readability [4], but are very difficult to implement. Internal DSLs, on the other hand, are built on top of the host language, carrying out the advantages and drawbacks of the host language. One major disadvantage of internal DSLs is that building Intermediate Representation (IR) of the program is not possible in an internal DSL, which further limits the applicable optimizations to dynamic optimizations and prevents them from generating their custom code [5]. This affects both parallel performance and code generation for heterogeneous devices [5].

DSLs provide a platform for domain programmers to automatically generate high performance code for different hardware architectures, by leaving the implementation details to the compile rather than the programmer. However, in order to achieve heterogeneous parallelism through DSLs, they have to be able to identify the parallelism in the application, and also to generate optimized parallel code for different platforms [6]. This requires a significant effort, along with a deep understanding of hardware architecture, parallelism, and scheduling from the developer [6], while maintenance is still a challenge and requires the language designers to adapt the implementation to newer architectures as they become available.

Code generation frameworks. The complexity of new architectures has increased the already exorbitant cost of developing and maintaining handwritten high-performance code. Meanwhile, the state-of-the-art frameworks, like Halide, are capable of automatically generating code for multiple architectures while being far less error-prone. The code generation frameworks are great tools for scheduling computations, and managing the associated data for achieving performances on par to highly-tuned handwritten code. It is worth mentioning, though, that gaining performance to higher extents requires a fair amount of knowledge about the architecture, and application requirements. Also, such frameworks usually target a particular runtime and may cause performance degradation in other environments.

Runtime systems including asynchronous runtime systems (AMTs) carry out the execution model of the program. Runtime systems facilitate parallelism and concurrency at thread level, and provide functionalities to dynamically adjust execution for the best performance. Their scope of effectiveness, however, is limited to operations rather than larger tasks such as algorithms, or applications.

While each of the above can independently improve the performance, using all together may have adverse effects on overall performance—e.g., Halide's native runtime and HPX competing for resources could be a source of performance degradation. In this work we have taken an overarching solution that avails the combined benefits of all these approaches in a single environment by:

- introducing the custom HPX runtime for Halide.
- introducing a new set of Phylanx primitive (functions) to allow Halide object files to be plugged in and called from Phylanx.
- demonstrating benefits of HPX performance analysis pipeline in Halide applcations.

In the following sections we first give an overview of several existing solutions in the related work II. Next, section IV briefly discusses the underlying technologies and how they are utilized in the software package followed up by the performance portability sections V explaining how Halide is enabled in Phylanx. In the experiments section V-B, we evaluate the effectiveness of the HPX backend and interoperability of Phyalnx and Halide. We conclude by lessons we learned from the study, and outline the future work.

III. Related Work

In this section we briefly discuss some of the similar work that has been done in the fields of DSL, code generation frameworks, and AMT runtime systems.

A. Domain Specific Languages (DSLs)

The Delite Compiler Framework and Runtime [6], based on Scala language, was developed in order to produce heterogeneous, and parallel domain specific languages. Delite uses a hybrid approach to balance between internal and external DSL implementations [6]. They utilize the concept of Language Virtualization [7], in which a host language that can be virtualized allows you to use their frontend while providing metaprogramming tools in order to generate an IR [5]. The Delite compiler creates an IR from the DSL, applies the relevant optimizations, automatically generates codes for different compute kernels, and forms the Delite execution graph (DEG), which is then scheduled to be executed with an execution plan [5]. To make it even easier for the DSL developers, they provide frequently used parallel patterns with code generators that can be easily mapped to their problems. The OptiML [8] machine learning DSL is developed based on Delite Compiler Framework and Runtime.

STELLA (STEncil Loop LAnguage) [9] is a DSL for solving partial differential equations on structured grids mainly used in weather and climate simulations. They utilize C++ template meta-programming to generate optimized loop nests from the DSL, and make specialized code generation for different architecture platforms possible through backend selection [9]. STELLA is implemented based on the standard C++ template metaprogramming framework [9].

Osuna et al. [10] developed Dawn as a DSL compiler toolchain for climate and weather applications. They use GTClang [11] as the DSL frontend, which is integrated in C++ through Clang compiler. In this DSL, first the model is considered sequentially, and a parallel representation of the model is created afterwards based on the identified data dependencies [10].

B. Code Generation Frameworks

The developers of CHiLL, a framework for Composing High-Level Loop Transformations [12], based their framework on empirical optimizations. They automatically generate different optimized versions of the code and run it to on the target platform with a set of relevant input in order to find the best-performing variant [12]. Their framework is capable of performing complex code transformations such as imperfect loop nest transformations [12]. Tiwari et al. [13] extend their work by using Active Harmony [14] in order to facilitate parallel search.

Baghdadi et al. [15] developed TIRAMISU, a polyhedral framework to generate high performance code on different hardware architectures. Tiramisu generates an intermediate representation from the original code using polyhedral model , applies the relevant transformations, and generates high performance code for different targets [15].

C. AMT Runtime Systems

Charm++ [16] is a parallel programming system in which the computation is divided into "*migratable*" objects called *chare*s and are left to the runtime to decide when and where to execute them. The execution model of Charm++ is based on *message-driven* execution which facilitates overlapping communication with computation [17].

Legion [18] is developed as a data-centric model in which the runtime identifies data locality and dependencies between the tasks, and performs the necessary movements and transformations to achieve high performance.

IV. Enabling Technologies

Phylanx benefits from many of existing open source libraries with the aim of improving the user experience, particularly, for performance analysis. In this section we discuss the enabling technologies that extend the usability of Phylanx.

A. HPX

Phylanx expression trees are run on the HPX thread pool [19]. HPX is an open-source (licensed under Boost Software License) C++ high-performance asynchronous many-task (AMT) runtime system for parallelism and concurrency. HPX provides a uniform API for parallel and distributed computation allowing threads to run both locally, and remotely– the latter per active messages called *parcels* [20]. Here we highlight a few HPX facilities used in Phylanx to carry out parallel operations and improve concurrency.

hpx::async In order to boost opportunities for global parallelism and concurrency, Phylanx follows the asynchronous programming pattern using HPX's *async* syntax. Any Phylanx program is a tree of asynchronous functions evaluated by HPX threading system. This allow the non-blocking async functions start executing as soon as their input is ready and not blocked by the slower statements that may appear before them in the program. Once the evaluation starts, as long as there are resources available on the system, any number of functions can run in parallel, resulting in improved system throughput.

hpx::future Similar to C++ *std::future* class template, provides a placeholder for the result of an asynchronous operation. Each Phylanx function returns a *future* object. The value of the object can be queried explicitly by the user through the *eval* method or may be evaluated implicitly once its value is needed by another function depending on it. *future* is essential for non-blocking evaluation of HPX execution tree.

hpx::for_loop HPX's Parallel Algorithms module provides a catalog of C++20 standard conforming algorithms including the *for_loop* which implements functionality over a range specified by integral or iterator bounds. The *for_loop* iteratively applies the input operation following the execution policy set by the user. The HPX Halide runtime (section V) relies on this construct to execute parallel loops. This approach provides performance better or on par with popular multiprocessing libraries such as OpenMP [21] without requiring any directives by the user. In addition, the profiling information will be readily available and visualizable through APEX (section IV-C) and Traveler (section IV-D).

B. Halide

HPX abstracts away many complexities of lower-level APIs for parallelism and concurrency. However, benefiting from the modern architectures to the fullest extent also requires global organization of the computation and the associated data movements. This issue is more pronounced on heterogeneous systems like the state-of-the-art HPC resources. Phylanx relies on the automatic code generation capabilities provided by Halide to overcome complications posed when developing and maintaining high-performance applications.

Halide separates program schedule, i.e., managing the intermediate storage and the order of computation, from the algorithm. Halide allows programmers to define the algorithm with a range of possible organization constraints and generates code with performance equal or better than hand-tuned code. Halide is also capable of generating code for multiple architectures including CPU and GPU from the single source.

C. APEX

APEX [22] (Autonomic Performance Environment for eXascale) is a performance measurement library for distributed, asynchronous multitasking runtime systems. APEX collects data though inspectors using the dependency chain in HPX's execution tree to produce traces–instead of the call stack.

The synchronous module of APEX uses an event API and event listeners. APEX can collect performance measurements both synchronously, and asynchronously. APEX's synchronous module will start, stop, yield, or resume timers whenever an event occurs. These timers can also capture hardware metrics using the PAPI [23] library. The asynchronous measurement involves periodic, or on-demand interrogation of OS, hardware, or runtime states and counters.

APEX also supports performance profiling of runtime tasks. The profile data contains the number of times each task was executed, and the total time spent executing that type of task. APEX is integrated with the Open Trace Format 2 (OTF2) [24] library capturing full event traces including event identification and start/stop times. In HPX applications, all tasks are uniquely identified by their GUID (globally unique identifier) and the GUID of their parent task. These GUIDs are captured as part of the OTF2 trace output.

D. Traveler

We use Traveler to visualize performance data collected by APEX. Traveler [25] is a visualization platform for parallel performance data. Traveler is built on web technologies and all visualized data is readily available through the web browser. Traveler provides interactive access to performance data at multiple levels of abstraction supporting charts such as time series, histograms, source code, and Gantt charts.

Additionally, Traveler supports aggregated execution graphs as more commonly analyzed to understand AMT execution. In the context of Phylanx, Traveler is capable of visualizing three kinds of data: (1) OTF2 trace data including task traces (optionally annotated with PAPI counters), and extra dependency information through APEX, (2) execution graph data generated by Phylanx, and (3) raw source code.

V. Performance Portability in Phylanx

Phylanx is an asynchronous array processing platform built on top of the HPX runtime system. Previous works have shown the performance-portablity of Phylanx in both shared-memory [26], and distributed [27]–[29] settings. Phylanx has also been tested on container technologies such as Singularity [30] and Docker [31], and run on Agave/Tapis [32] science gateway through the JetLag [28], [33] interactive environment.

So far, however, Phylanx has relied on two of the three performance optimizing solutions discussed I above, namely, the domain knowledge, and the HPX runtime system. This has been made possible through: (1) the low-level PhySL representation, and (2) the Python frontend.

Internally, all the algorithms and operations of the Phylanx platform are implemented in PhySL (Phylanx Specialization Language). PhySL runs on HPX thread pool to exploit fine-grain parallelism, and concurrency. It benefits from constraint-based synchronization, through *asyn* and *future* constructs, in order to maximize throughput and also improves opportunities for overlapping computation and communication. The frontend, on the other hand, seamlessly manages data between Python and PhySL making low-level functionalities available in python via the *decorator* design pattern.

In this paper we introduce the new approach of automated code generation, through Halide to bring performance portability into Phylanx. This was done in two steps: first, we developed a custom HPX runtime for Halide (section V-A), and next, implemented an interface to Phylanx enabling interoperability between Phylanx and Halide object files (section V-B).

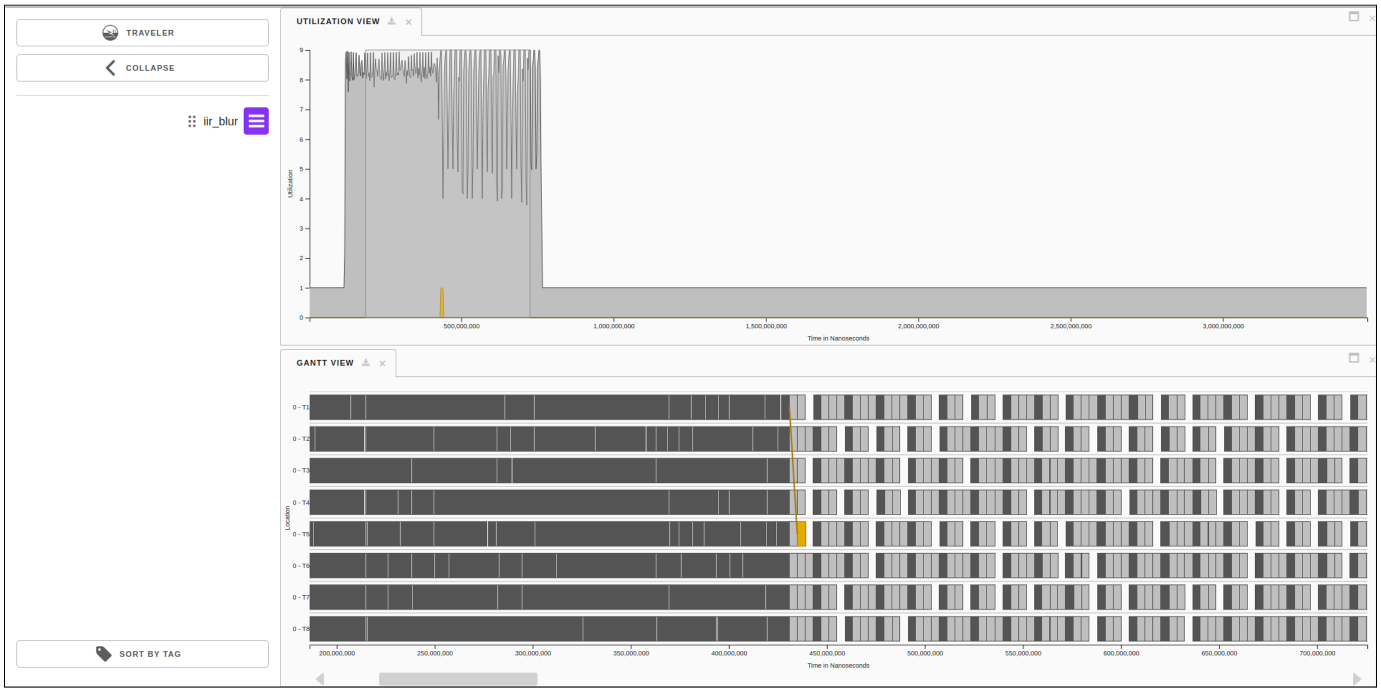


Fig. 1: The utilization view (on top) and Gantt, time series, view (bottom) generated by Traveler for iir_blur Halide application. Traveler work seamlessly with HPX runtime and requires no changes to the code. The yellow line represents dependency between two tasks.

A. *HPX runtime for Halide*

Halide's native runtime provides a highly tuned thread pool to optimize for the types of contention patterns that its pipelines encounter. However, to avoid interference with HPX running Phylanx tasks, we exported Halide's thread pool to HPX.

Halide has a minimal runtime requirements, solely requiring a memory allocation, and threading implementation. It allows individual pieces of the runtime to be overridden either through weak linking (on supported platforms), or explicitly by calling functions that overwrite set of function pointers. In order to guarantee support for both Unix-like and Windows platforms, we have overwritten the parallel loop function pointers to replace the default thread pool implementation, using HPX *for_loop* with the parallel execution policy. The HPX runtime is available to Halide in both just-in-time (JIT) and ahead-of-time (AoT) modes. Listing 1 shows an example of how one can use HPX's parallel loop construct in the JIT context. Similarly, for the case of ahead of time compilation, we have partially exported Halide's thread pool API, overriding Halide's default runtime. After compilation, the object file, contining the overridden functions, can be linked against any Halide application to have HPX threads carry out the tasks.

B. *Halide integration in Phylanx*

The goal is for Phylanx to use Halide to facilitate developing new *primitives* (Phylanx functions). To that end, we have developed a new kind of Phylanx plugin that allows calling the functions in generated Halide shared objects from Phylanx applications. This requires tasks of the two platforms to run in parallel, or concurrently and exchange data:

Runtime compatibility. Although it is possible to use JIT compilation, Phylanx relies on Halide's generators (ahead-of-time compilation scheme) to avoid runtime overheads. This approach allows us to compile Halide objects during Phylanx compilation and requires no changes to the Halide code. The only requirements is for the Halide libraries to be linked against the HPX runtime (section V-B). Having HPX carrying out tasks in both Phylanx and Halide guarantees there will be no interference between the two platforms competing over resources.

```
int hpx_parallel_loop(void *ctx, int (*f)(void
     *, int, uint8_t *),
    int min, int extent, uint8_t *closure)
{
    hpx::for_loop(hpx::execution::par,
        min, min + extent,
        [&](int i) { f(ctx, i, closure); });
    return 0;
}

int main(int argc, char **argv) {
    // construct the `brighten' algorithms in
       Halide ...
    Func brighten;
    // override the default parallel loop
    brighten.set_custom_do_par_for(
            &hpx_parallel_loop);
    // ...
    // call the function as usual
    output =
        brighten.realize({input.width(),
                          input.height(),
                          input.channels()});
    return 0;
}
```

Listing 1: Example of using `hpx::for_loop` to carry out Halide's parallel loops with JIT compilation.

Data Management. Phylanx data objects are built on top of blaze [34] and blaze_tensor [35] while Halide works with *halide buffers*. Fortunately, both support a general and convenient approach for exchanging data between the libraries by exposing a buffer view. The view provides direct access to the the underlying raw data in each object. The Halide plugin implements the interface between the two by passing the pointers to the data, allowing copy-free interoperability. The similar scheme is used for exchanging the data between Phylanx and NumPy [36] arrays, so data can be seamlessly accessed in any of the platforms with being copied.

```
from phylanx import Phylanx
import cv2
import numpy

@Phylanx
def py_harris(img):
    return harris(img)

img = cv2.imread("rgba.png")
data = numpy.asarray(img)

new_img = py_harris(data)

cv2.imwrite('result_hpx.png', new_img)
```

Listing 2: An example of calling Phylanx's harris function in Python

Listing 2 is an example of a primitives implemented through the Halide plugin. The implementation of the `harris` function is taken from Halide repository[1] and wrapped as a primitive through Phylanx's Halide plugin. This example demonstrates how the data can be seamlessly shared by NumPy, Phyalnx, and Halide.

Finally, it is worth noting that, since the whole pipeline runs on HPX, all the benefits of APEX and Traveler are at the user's disposal in the entire platform.

VI. Results

In this section we show the empirical results gathered from running a number of Halide applications using HPX thread pool in the context of Phylanx package.

A. System Setup

All the experiments were conducted on the Rostam [37] cluster at he Center for Computation and Technology at LSU.

[1]https://github.com/halide/Halide/tree/master/apps/harris

TABLE I: Specifications of the Maedusa and Kamand nodes on the Rostam cluster at CCT.

Node	CPU	RAM	Number of Cores
Medusa	Intel Skylake	96 GB	40
Kamand	AMD Rome	512 GB	128

Newly upgraded Rostam 2.0 consists of 53 nodes ranging from state-of-the-art Skylake and Rome architectures with accelerators to Raspberry Pi nodes. The results in this work are produced on Kamand and Medusa nodes. Tables I provide the specifications of each node.

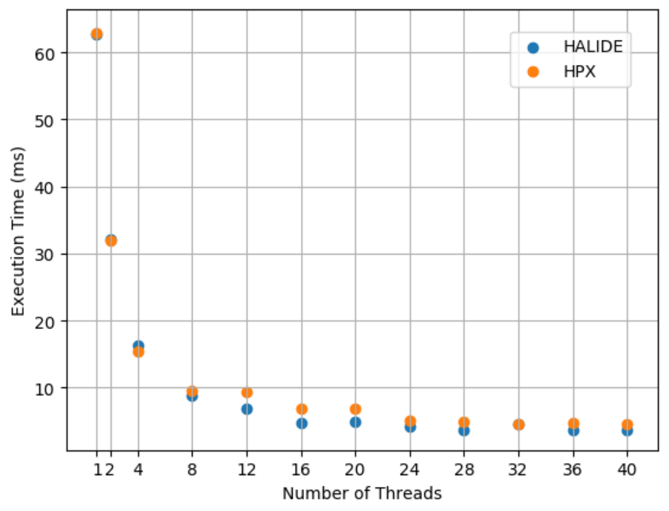
We use the latest version of all software libraries at the time of the experiment. Table II summarizes all the versions and commits used for the experiments.

B. Experiments

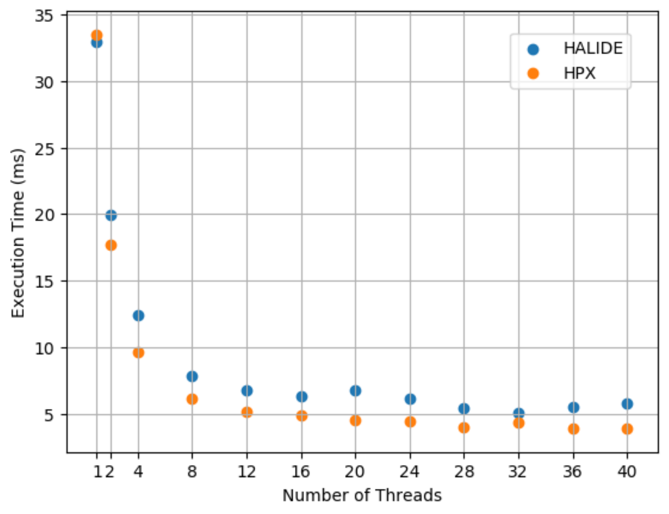
In order to evaluate the effectiveness of the HPX runtime for Halide, we selected existing applications available on Halide's GitHub repository and compared the performances with that of the native runtime. We carried out he experiments on two of the latest architectures SkyLake (Intel) and Rome (AMD) with hyper-threading disabled in all cases. We observed matching scaling patterns on both architectures across all our tests. Figures 2, and 3 show the trend in three common applications. Although in some cases HPX is slower for smaller number of threads, that gap between the execution times closes as the number of threads increases.
Listing 2 is an example of Phylanx primitive, `harris`, that is called from python, demonstrating the compatibility of the two platforms.

TABLE II: Specifications of the libraries used in the experiments.

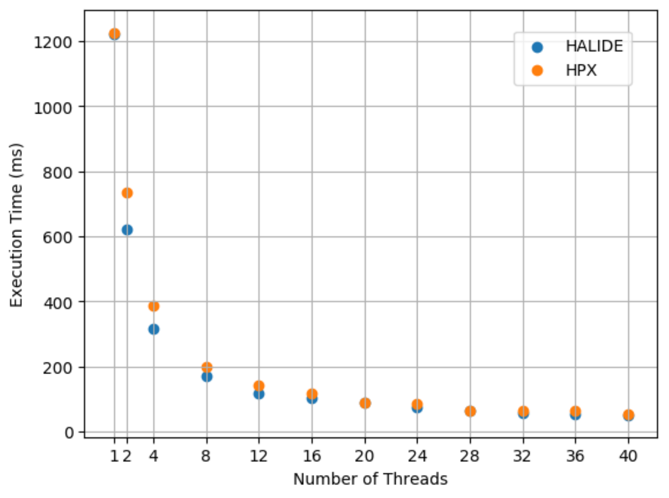
Library	Version	Commit
HPX	1.8.0	0db6fc565c
Blaze	3.9.0	89ee9476df
Phylanx	0.1	295b5f82cc
Halide	12.0.0	085e11e0dc



(a)

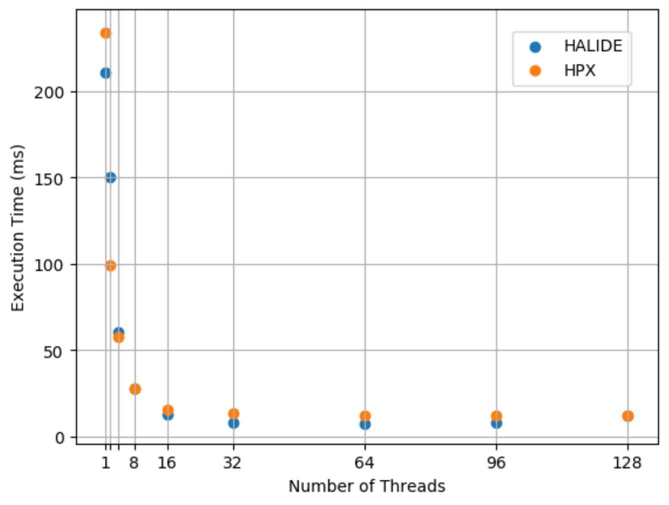


(b)

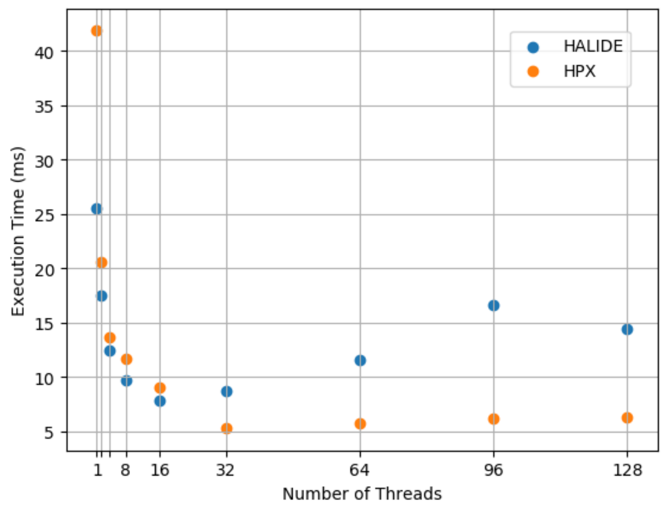


(c)

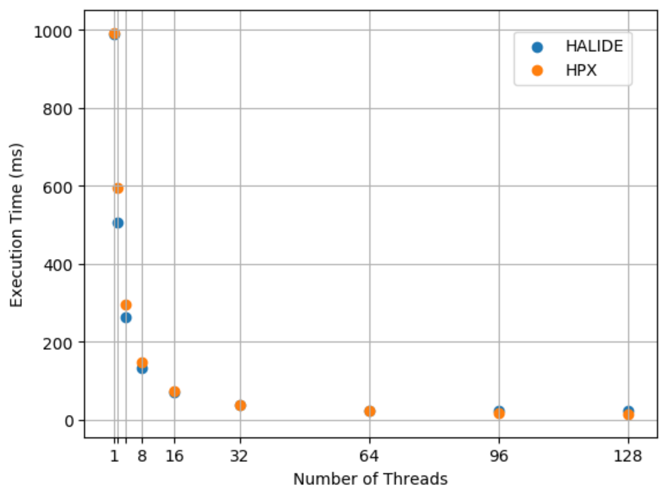
Fig. 2: (a) Convolution layers, (b) IIR Blur, and (c) Non-Local Means run on Intel SkyLake. Both Halide's native runtime and HPX scale similarly.



(a)



(b)



(c)

Fig. 3: (a) Convolution layers, (b) IIR Blur, and (c) Non-Local Means running on AMD Rome architecture. Showing comparable performance for both Halide's native runtime and HPX.

VII. Conclusion

Separation of algorithms from the scheduling of their computation has been shown effective in removing challenges of programming on heterogeneous systems and the associated portability issues. There are several established approaches, including DSLs, code generation frameworks, and runtime systems to provide such abstractions. In this work, we extended the Phylanx array processing platform to enable code generation using Halide, complimenting the HPX runtime, to better support performance portability. In addition, the HPX-APEX-Traveler pipeline provides excellent tools for analyzing Halide code. The pipeline facilitates measuring and visualizing the performance data without requiring any changes in the application code. To the best of our knowledge, HPX runtime for Halide is the first of its kind to outperform Halide's native runtime. As the future work, we are planning to export the remaining thread pool APIs and also add GPU support.

Acknowledgment

The authors are grateful for the support of this work by the LSU Center for Computation & Technology and by the DTIC project: Phylanx Engine Enhancement and Visualizations Development (Contract Number: FA8075-14-D-0002/0007).

References

[1] Robert R Schaller. Moore's law: past, present and future. *IEEE spectrum*, 34(6):52–59, 1997.

[2] Robert H Dennard, Fritz H Gaensslen, Hwa-Nien Yu, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[3] Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):1–25, 2014.

[4] Ryan D Kelker. *Clojure for domain-specific languages*. Packt Publishing Ltd, 2013.

[5] HyoukJoong Lee, Kevin Brown, Arvind Sujeeth, Hassan Chafi, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Implementing domain-specific languages for heterogeneous parallel computing. *Ieee Micro*, 31(5):42–53, 2011.

[6] Kevin J Brown, Arvind K Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A heterogeneous parallel framework for domain-specific languages. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 89–100. IEEE, 2011.

[7] Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. Language virtualization for heterogeneous parallel computing. *ACM sigplan notices*, 45(10):835–847, 2010.

[8] Arvind K Sujeeth, HyoukJoong Lee, Kevin J Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand R Atreya, Martin Odersky, and Kunle Olukotun. Optiml: an implicitly parallel domain-specific language for machine learning. In *ICML*, 2011.

[9] Tobias Gysi, Carlos Osuna, Oliver Fuhrer, Mauro Bianco, and Thomas C Schulthess. Stella: A domain-specific tool for structured grid methods in weather and climate models. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pages 1–12, 2015.

[10] Carlos Osuna, Tobias Wicky, Fabian Thuering, Torsten Hoefler, and Oliver Fuhrer. Dawn: a high-level domain-specific language compiler toolchain for weather and climate applications. *Supercomputing Frontiers and Innovations*, 7(2):79–97, 2020.

[11] C Osuna, F Thuering, T Wicky, J Dahm, et al. Meteoswiss-apn/dawn: 0.0. 2 (2020).

[12] Chun Chen, Jacqueline Chame, and Mary Hall. Chill: A framework for composing high-level loop transformations. Technical report, Citeseer, 2008.

[13] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12. IEEE, 2009.

[14] Cristian Tapus, I-Hsin Chung, and Jeffrey K Hollingsworth. Active harmony: Towards automated performance tuning. In *SC'02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 44–44. IEEE, 2002.

[15] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–205. IEEE, 2019.

[16] Laxmikant V Kale and Sanjeev Krishnan. Charm++ a portable concurrent object oriented system based on c++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 91–108, 1993.

[17] Laxmikant V Kale and Abhinav Bhatele. *Parallel science and engineering applications: The Charm++ approach*. CRC Press, 2019.

[18] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

[19] Hartmut Kaiser, Patrick Diehl, Adrian S Lemoine, Bryce Adelstein Lelbach, Parsa Amini, Agustín Berge, John Biddiscombe, Steven R Brandt, Nikunj Gupta, Thomas Heller, et al. Hpx-the c++ standard library for parallelism and concurrency. *Journal of Open Source Software*, 5(53):2352, 2020.

[20] Bibek Wagle, Samuel Kellar, Adrian Serio, and Hartmut Kaiser. Methodology for adaptive active message coalescing in task based runtime systems. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1133–1140. IEEE, 2018.

[21] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[22] Kevin A Huck, Allan Porterfield, Nick Chaimov, Hartmut Kaiser, Allen D Malony, Thomas Sterling, and Rob Fowler. An autonomic performance environment for exascale. *Supercomputing frontiers and innovations*, 2(3):49–66, 2015.

[23] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, volume 710. Citeseer, 1999.

[24] Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E Nagel, and Felix Wolf. Open trace format 2: The next generation of scalable trace formats and support libraries. In *Applications, Tools and Techniques on the Road to Exascale Computing*, pages 481–490. IOS Press, 2012.

[25] Traveler-integrated. https://github.com/hdc-arizona/traveler-integrated, 2021.

[26] R Tohid, Bibek Wagle, Shahrzad Shirzad, Patrick Diehl, Adrian Serio, Alireza Kheirkhahan, Parsa Amini, Katy Williams, Kate Isaacs, Kevin Huck, et al. Asynchronous execution of python code on task-based runtime systems. In *2018 IEEE/ACM 4th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, pages 37–45. IEEE, 2018.

[27] Bita Hasheminezhad, Shahrzad Shirzad, Nanmiao Wu, Patrick Diehl, Hannes Schulz, and Hartmut Kaiser. Towards a scalable and distributed infrastructure for deep learning applications. In *2020 IEEE/ACM Fourth Workshop on Deep Learning on Supercomputers (DLS)*, pages 20–30. IEEE, 2020.

[28] Steven R Brandt, Bita Hasheminezhad, Nanmiao Wu, Sayef Azad Sakin, Alex R Bigelow, Katherine E Isaacs, Kevin Huck, and Hartmut Kaiser. Distributed asynchronous array computing with the jetlag environment. In *2020 IEEE/ACM 9th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*, pages 49–57. IEEE, 2020.

[29] Nikunj Gupta, Steve R Brandt, Bibek Wagle, Nanmiao Wu, Alireza Kheirkhahan, Patrick Diehl, Felix W Baumann, and Hartmut Kaiser. Deploying a task-based runtime system on raspberry pi clusters. In *2020*

IEEE/ACM 5th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2), pages 11–20. IEEE, 2020.

[30] Gregory M Kurtzer, Vanessa Sochat, and Michael W Bauer. Singularity: Scientific containers for mobility of compute. *PloS one*, 12(5):e0177459, 2017.

[31] Dirk Merkel et al. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[32] Rion Dooley, Steven R Brandt, and John Fonner. The agave platform: An open, science-as-a-service platform for digital science. In *Proceedings of the Practice and Experience on Advanced Research Computing*, pages 1–8. 2018.

[33] Steven R Brandt, Alex Bigelow, Sayef Azad Sakin, Katy Williams, Katherine E Isaacs, Kevin Huck, Rod Tohid, Bibek Wagle, Shahrzad Shirzad, and Hartmut Kaiser. Jetlag: An interactive, asynchronous array computing environment. In *Practice and Experience in Advanced Research Computing*, pages 8–12. 2020.

[34] Blaze. https://bitbucket.org/blaze-lib/blaze/. Accessed: 2021-09-10.

[35] Blaze tensor. https://github.com/STEllAR-GROUP/blaze_tensor/, 2021.

[36] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in science & engineering*, 13(2):22–30, 2011.

[37] Rostam cluster, stellar group at cct. https://wiki.rostam.cct.lsu.edu/, 2021.

Appendix A
Artifact Description Appendix: [Halide Code Generation Framework in Phylanx]

A. Abstract

This description contains the information needed to launch some experiments of the SC21 paper "Halide Code Generation in Phylanx". In this appendix we explain how to compile and run the Halide Code Generation Framework within Phylanx. The results represented in the paper can be reproduced as explained.

B. Description

1) *Check-list (artifact meta information):*

- **Program:**
- **Compilation:** GCC version 10.2.0
- **Output:** execution time of the application
- **Experiment workflow:** build HPX, Phylanx, Halide and Phylanx_halide libraries
- **Publicly available?:** yes

2) How software can be obtained (if available): All utilized software are open-source and could be accessed through github.

```
git clone https://github.com:STEllAR-GROUP/hpx
    .git
git clone https://github.com/STEllAR-GROUP/
    phylanx.git
git clone https://github.com/pybind/pybind11.
    git
git clone https://bitbucket.org/blaze-lib/
    blaze.git
git clone https://github.com/STEllAR-GROUP/
    blaze_tensor.git
git clone https://github.com:halide/Halide.git
git clone https://github.com/STEllAR-GROUP/
    phylanx\_halide.git
```

3) Software dependencies: HPX depends on *boost*, *hwloc* libraries. Phylanx depends on *HPX*, *pybind*, *blaze*, and *blaze_tensor* libraries.

C. Installation

```
# Build HPX
cd /work/${USER}/p3hpc/hpx
cmake -DCMAKE_BUILD_TYPE=Release -
    DCMAKE_CXX_FLAGS="-std=c++17" -
    DHPX_WITH_THREAD_IDLE_RATES=ON -
    DHPX_WITH_MALLOC=tcmalloc -
    DHPX_WITH_EXAMPLES=OFF -DAPEX_WITH_OTF2=ON
     -DHPX_WITH_APEX=ON -DHPX_WITH_FETCH_ASIO=
    ON  -Wdev -S . -B cmake-build-release
cmake --build cmake-build-release/ --parallel
cmake --install cmake-build-release/ --prefix
    cmake-install-release
```

```
# Build Blaze
cd /work/${USER}/p3hpc/blaze
cmake -DCMAKE_BUILD_TYPE=Release -
    DCMAKE_CXX_FLAGS="-std=c++17" -
    DBLAZE_SMP_THREADS=HPX -DHPX_DIR=/work/${
    USER}/p3hpc/hpx/cmake-install-release/
    lib64/cmake/HPX  -S . -B cmake-build-
    release
cmake --build cmake-build-release/ --parallel
cmake --install cmake-build-release/ --prefix
    cmake-install-release
```

```
# Build Blaze Tensor
cd /work/${USER}/p3hpc/blaze_tensor
cmake -DCMAKE_BUILD_TYPE=Release -
    DCMAKE_CXX_FLAGS="-std=c++17" -Dblaze_DIR
    =/work/${USER}/p3hpc/blaze/cmake-install-
    release/share/blaze/cmake -DHPX_DIR=/work/
    ${USER}/p3hpc/hpx/cmake-install-release/
    lib64/cmake/HPX -S . -B cmake-build-
    release
cmake --build cmake-build-release/ --parallel
cmake --install cmake-build-release/ --prefix
    cmake-install-release
```

```
# Blaze pybind11
cd /work/${USER}/p3hpc/pybind11
cmake -DCMAKE_BUILD_TYPE=Release -
    DCMAKE_CXX_FLAGS="-std=c++17" -
    DPYTHON_EXECUTABLE:FILEPATH=python3.6 -S .
     -B cmake-build-release
cmake --build cmake-build-release/ --parallel
cmake --install cmake-build-release/ --prefix
    cmake-install-release
```

```
# Build Phylanx
cd /work/${USER}/p3hpc/phylanx
cmake -DCMAKE_BUILD_TYPE=Release -
    DCMAKE_CXX_FLAGS="-std=c++17" \
  -Dpybind11_DIR=/work/${USER}/p3hpc/pybind11/
      cmake-install-release/share/cmake/
      pybind11 \
  -Dblaze_DIR=/work/${USER}/p3hpc/blaze/cmake-
      install-release/share/blaze/cmake \
  -DBlazeTensor_DIR=/work/${USER}/p3hpc/
      blaze_tensor/cmake-install-release/share
      /blaze_tensor/cmake \
  -DPHYLANX_WITH_EXAMPLES=OFF \
  -DHPX_WITH_MALLOC=tcmalloc \
  -DHPX_DIR=/work/${USER}/p3hpc/hpx/cmake-
      install-release/lib64/cmake/HPX/ \
  -DPHYLANX_WITH_VIM_YCM=ON \
  -Wdev \
  -S . -B cmake-build-release
cmake --build cmake-build-release/ --parallel
    12
cmake --install cmake-build-release/ --prefix
    cmake-install-release
```

```
# Build Halide
cd /work/${USER}/p3hpc/Halide/
cmake -DCMAKE_BUILD_TYPE=Release -
    DCMAKE_CXX_FLAGS="-std=c++17" -
    DCMAKE_TOOLCHAIN_FILE=/work/${USER}/vcpkg/
    scripts/buildsystems/vcpkg.cmake -S . -B
    cmake-build-release
cmake --build cmake-build-release/ --parallel
cmake --install cmake-build-release/ --prefix
    cmake-install-release
```

```
# Build Halide Plugin
cd /work/${USER}/p3hpc/phylanx_halide/
cmake -DCMAKE_BUILD_TYPE=Release -
    DCMAKE_CXX_FLAGS="-std=c++17" \
  -DCMAKE_PREFIX_PATH="/work/${USER}/p3hpc/hpx
      /cmake-install-release/lib64/cmake/HPX
      /;/work/${USER}/p3hpc/Halide/cmake-
      install-release/lib64/cmake/
      HalideHelpers/;/work/${USER}/p3hpc/
      Halide/cmake-install-release/lib64/cmake
      /Halide" \
  -DPhylanx_DIR=/work/${USER}/p3hpc/phylanx/
      cmake-build-release/lib/cmake/Phylanx/ \
  -S . -B cmake-build-release
cmake --build cmake-build-release/ --parallel
cmake --install cmake-build-release/ --prefix
    cmake-install-release
```

D. *Evaluation and expected result*

For Halide code generation within Phylanx, after building the Phylanx_Halide library, the generated *.so* file should be copied to the Phylanx build directory. The *harris.py* example provided in the examples directory should be easily run as *python harris.py*.