FFTX-IRIS: Towards Performance Portability and Heterogeneity for SPIRAL Generated Code

Sanil Rao
Carnegie Mellon University
sanilr@andrew.cmu.edu

Mohammad Alaul Haque Monil
Oak Ridge National Laboratory
monilm@ornl.gov

Het Mankad
Carnegie Mellon University
hmankad@andrew.cmu.edu

Jeffrey S. Vetter
Oak Ridge National Laboratory
vetter@ornl.gov

Franz Franchetti
Carnegie Mellon University
franzf@andrew.cmu.edu

ABSTRACT

FFTX-IRIS is a dynamic system to efficiently utilize novel heterogeneous platforms. This system links two next-generation frameworks, FFTX and IRIS, to navigate the complexity of different hardware architectures. FFTX provides a runtime code generation framework for high-performance Fast Fourier Transform kernels. IRIS runtime provides portability and multi-device heterogeneity, allowing computation on any available compute resource. Together, FFTX-IRIS enables code generation, seamless portability, and performance without user involvement. We show the design of the FFTX-IRIS system along with an evaluation of various small FFT benchmarks. We also demonstrate multi-device heterogeneity of FFTX-IRIS with a larger stencil application.

KEYWORDS

Performance portability; Heterogeneity; FFT; SPIRAL; IRIS

ACM Reference Format:
Sanil Rao, Mohammad Alaul Haque Monil, Het Mankad, Jeffrey S. Vetter, and Franz Franchetti. 2023. FFTX-IRIS: Towards Performance Portability and Heterogeneity for SPIRAL Generated Code. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3624062.3624242

1 INTRODUCTION

Modern applications' computational demands are increasing exponentially. After Dennard scaling ended, heterogeneous systems emerged as go-to solutions to meeting the ever-increasing computation need. As a result, heterogeneous systems are now ubiquitous in all contemporary state-of-the-art high-performance computing facilities. In a heterogeneous computing paradigm, multi-core and many-core processors and accelerators from different manufacturers co-exist in a single node. While adopting such heterogeneity keeps the ongoing advances in computation power, it introduces challenges in portability, utilization, and efficient execution.

Harnessing meaningful performance from a wide range of available architectures has become necessary and requires rigorous optimization effort and an in-depth understanding of the underlying architecture. For this reason, architecture-specific tuning of complex kernels (such as FFTs) has been an active field of research. Automated code generation frameworks, such as SPIRAL [5], aim to deliver the promise of providing optimized code for a given architecture. Since optimizations are architecture-specific, portability to different kinds of heterogeneous systems becomes a concern. Therefore, an additional layer that ensures portability to such code synthesis and generation system is a must to alleviate the burden on the users. In an ideal case, there should be an abstraction on top of the code generation framework that hides the processor detail, and at runtime, code is generated and executed for the processor being used.

FFTX is one such abstraction layer. FFTX is a modern FFT library providing common optimized FFT transforms on a wide variety of hardware platforms. As a user-facing library, FFTX provides familiar interfaces for different transforms that can be easily integrated into large applications. These interfaces then translate to the SPIRAL code generation system, which provides optimized kernels for specific hardware architectures. This entire process is done transparently to the user. While FFTX provides an abstraction on top of SPIRAL to ease the code generation process, there are challenges ensuring portability to diverse architectures.

Another dimension of the challenges of heterogeneous systems stems from the utilization perspective. It is challenging for application developers to manage the utilization of all available compute devices, especially if those devices change between compute systems. This results in extra control logic and static partitioning of the application for specific system configurations. This distracts from the actual computation the application is trying to solve. State-of-the-art runtime systems for heterogeneous systems such as IRIS runtime [6] provide orchestration capabilities to ensure utilization by introducing a task-based programming model and providing seamless data transfers between different compute units. In the task-based programming paradigm, a higher level of abstraction of the computation is created by introducing tasks that can be executed in different heterogeneous compute units following given scheduling logic. While task-level abstraction provides means for portability and utilization of multi-device heterogeneity, expressing the computing need in a task graph is application dependent and requires effort from the users.

SC-W 2023, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0785-8/23/11.
https://doi.org/10.1145/3624062.3624242

To address these challenges, we propose FFTX-IRIS, a dynamic system for next-generation systems. FFTX-IRIS strives to tie two worlds: 1) architecture-optimized code generation using SPIRAL, and 2) ensuring portability and multi-device heterogeneity using IRIS runtime. FFTX-IRIS handles the performance and portability of modern applications in the backend while providing developers with an architecture-agnostic library-style API resembling standard APIs. Using SPIRAL and IRIS internally, FFTX-IRIS combines code generation, automatic task representation of the computation, Just-In-Time compilation, and dynamic runtime scheduling to use all available computing platforms efficiently. This paper focuses on showcasing the recent capabilities of FFTX-IRIS toward portability and heterogeneity.

Contributions. This paper makes the following contributions:

- Combining code generation and task-based execution by integrating FFTX and IRIS runtime to create a new system, FFTX-IRIS.
- Automated task level abstraction for generated kernels by SPIRAL.
- Demonstration of performance portability of benchmarks written against FFTX-IRIS on different architectures, CPUs, and GPUs from different vendors.
- Utilization of multi-device Heterogeneity of the benchmarks written against FFTX-IRIS; different hardware architectures are used at the same time to solve a single problem.

2 BACKGROUND

FFTX-IRIS consists of two key software frameworks, FFTX and IRIS, with FFTX using a code generation system called SPIRAL as its backend. We provide a brief overview of each framework individually before showing by example, the new FFTX-IRIS design.

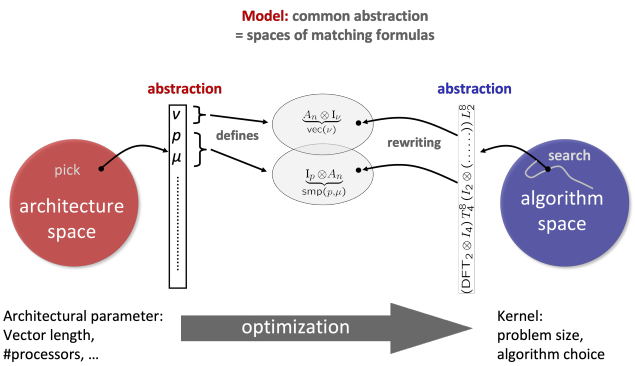

Figure 1: SPIRAL code generation system[5].

2.1 SPIRAL

SPIRAL is a code generation system that produces optimized C/C++ code for different types of multi dimensional linear transforms like the discrete Fourier transform (DFT) and sine/cosine transforms that are mainly used in the area of signal processing [5]. It uses a mathematical declarative language called the Operator language (OL) [4] that helps to capture the semantics of the given problem specification. This is then broken down into a loop based structure called the Σ-OL, which helps in generating an optimized C/C++ code for that given problem specification. Figure 1 shows the basic task flow that is followed in SPIRAL to generate optimized code for a given problem specification. In recent years, the scope of SPIRAL has expanded to areas of scientific applications other than signal processing, like graph algorithms, stencil algorithms used in numerical partial differential equations (PDE), and cryptography.

2.2 FFTX

The FFTX project is a high-performance library designed to provide common FFT transforms for new hardware architectures. Unlike traditional libraries, FFTX uses a combination of a code generator, SPIRAL[5], and Just-In-Time compilation as its library backend. In this model, optimized kernels are generated using runtime parameters and compiled into running application. This is in contrast to handwritten routines implementing library function calls. This approach enables increased optimization and portability of applications across different hardware platforms.

2.3 IRIS Runtime

IRIS [2, 6] (see Figure 2) is a task-based programming model and runtime for heterogeneous computing systems consisting of multicore CPUs, GPUs (NVIDIA, AMD), DSPs (Qualcomm Hexagon), and FPGAs (Xilinx, Intel). It accepts the kernels written in OpenMP, OpenCL, CUDA, HIP, XilinxCL, IntelCL, Hexagon C++, and OpenACC programming languages. However, mapping the programming language to the device compute unit is constrained. For example, NVIDIA GPU compute devices can be used through kernels written only in OpenCL and CUDA. The IRIS runtime has a task scheduler that maps the application tasks to a compute device, and the task kernels are executed using the compute unit-specific runtime. It orchestrates multiple programming platforms in a system into a single execution/programming environment. IRIS supports OpenMP, CUDA, HIP, OpenCL, XilinxCL, IntelCL, and Hexagon DSP runtimes while abstracting all underlying heterogeneous runtimes, providing task mapping, memory management, and scheduling at runtime.

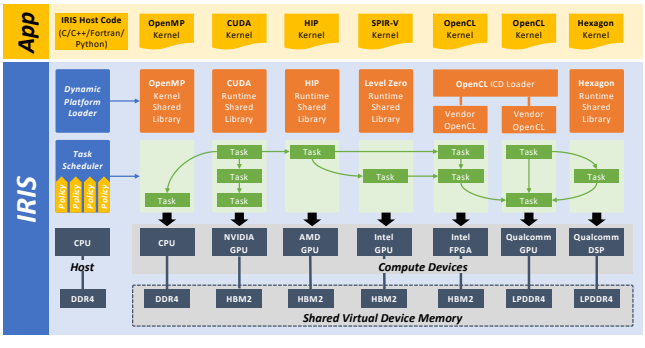

Figure 2: IRIS run-time system for heterogeneous architectures [2, 6].

3 RELATED WORK

There are runtime systems that provide functionalities in terms of heterogeneity, such as StarPU [1], OpenMP [11], OpenACC [10], etc. These runtime systems support a set of architectures and often lack the capabilities of executing a single problem on diverse heterogeneity (such as executing on multi-vendor GPUs at the same time). Portable abstractions such as Kokkos [3] provide a solution for portability but lack multi-device heterogeneity. So, a runtime system that supports both portability through task abstraction and multi-device heterogeneity is desired for seamless execution in contemporary heterogeneous systems. IRIS runtime [6] mitigates these challenges; hence, this work's runtime is of interest.

There are DSL compilers that combine code generation and task-based runtime systems, such as DISTAL[13] and spDISTAL[14]. These compilers expose language constructs to describe domain specific computations and their mappings onto distributed machines. Using these constructs, optimized distributed code is generated by the DSL compiler efficiently using machine resources. FFTX-IRIS takes a different approach to integrate two ideas: 1) code generation and 2) runtime scheduling, focusing on single-node systems. Kernel generation is exposed through a user-facing library that translates to the SPIRAL DSL. These kernels are generated at runtime for the specific architectures present on the machine. IRIS then leverages these kernels at runtime and automatically manages their execution on the available devices. This process is completely transparent to the end user.

4 FFTX-IRIS DESIGN

We describe the major objectives of the FFTX-IRIS system and also walk through all major components and their interactions.

4.1 Objective

FFTX-IRIS is a framework that provides users with automatic parallelization and task-level abstraction for efficient use of modern heterogeneous systems. This is done through a common frontend interface that is hardware architecture agnostic and is sequential/single-threaded by inspection of the source code. Under the hood, however, is a translation to device-optimized code that can be seamlessly executed across all available devices. This is all done dynamically at runtime, enabling hardware portability.

4.2 Architecture Agnostic Frontend

Figure 3 shows a user-written Multi-Dimensional Discrete Fourier Transform (MDDFT) application using a traditional C++ syntax. The user declares the sizes for each dimension as well as create some buffers for the input and output of the FFT. They then create and instantiate the `FFTXProblem`[12] object (the library API of FFTX), passing the memory object sizes and kernel name to the constructor. Then, there is a call to the member function `transform` to perform the computation. Finally, the result is printed to verify correctness.

`FFTXProblem` is an abstract class that must be derived for the specific transformation the user requires. These derived classes hold different computations of interest to the application developer. This example shows a derived `MDDFTProblem`.

```
#include <stdio.h>
#include <iostream>
#include <vector>
#include "iris.hpp"
#include "fftx.hpp"

int main(int argc, char** argv) {
  int n,m,k;
  n = 8;
  m = 8;
  k = 8;
  std::vector<int> sizes{n,m,k};
  double *Y, *X, *sym;
  X = new double[n*m*k*2];
  Y = new double[n*m*k*2];
  sym = new double[n*m*k*2];
  generateInputBuffer(X, sizes);
  std::vector<void*> args{Y,X,sym};
  MDDFTProblem mdp(args,sizes,"mddft");
  mdp.transform();
  for(int i = 0; i < n*m*k; i++) {
    std::cout << Y[i] << std::endl;
  }
  return 0;
}
```

Figure 3: MDDFT Application using the FFTX-IRIS system

```
for(int i = 0; i < kernel_names.size(); i++) {
    iris_task task;
    iris_task_create(&task);

    if((getIRISARCH().find("cuda") != std::string::npos
    || getIRISARCH().find("hip") != std::string::npos)
    && !findOpenMP()) {
        std::vector<size_t> grid{
        (size_t)kernel_params[i*6]*kernel_params[i*6+3],
        (size_t)kernel_params[i*6+1]*kernel_params[i*6+4],
        (size_t)kernel_params[i*6+2]*kernel_params[i*6+5]};

        std::vector<size_t> block{
        (size_t)kernel_params[i*6+3],
        (size_t)kernel_params[i*6+4],
        (size_t)kernel_params[i*6+5]};

        iris_task_kernel(task, kernel_names.at(i).c_str(),
        3, NULL, grid.data(), block.data(),
        sig_types.size()+pointers,
        params.data(), params_info.data());
    } else{
        iris_task_kernel(task, kernel_names.at(i).c_str(), 1,
        NULL, &size, NULL, 3+pointers, params.data(),
        params_info.data());
    }

    if(i == kernel_names.size() -1)
    iris_task_dmem_flush_out(task, *(iris_mem*)params.at(0));

    iris_task_submit(task, iris_any, NULL, 1);
}
```

Figure 4: IRIS task creation and execution for FFTX generated kernels

4.3 Code Generation and Task Abstraction

FFTX-IRIS utilizes a combination of runtime code generation and scheduling in order to transition from single-threaded code to parallel code with multi-device execution. This process occurs when the member function `transform` is executed and is illustrated as a block diagram in Figure 5.

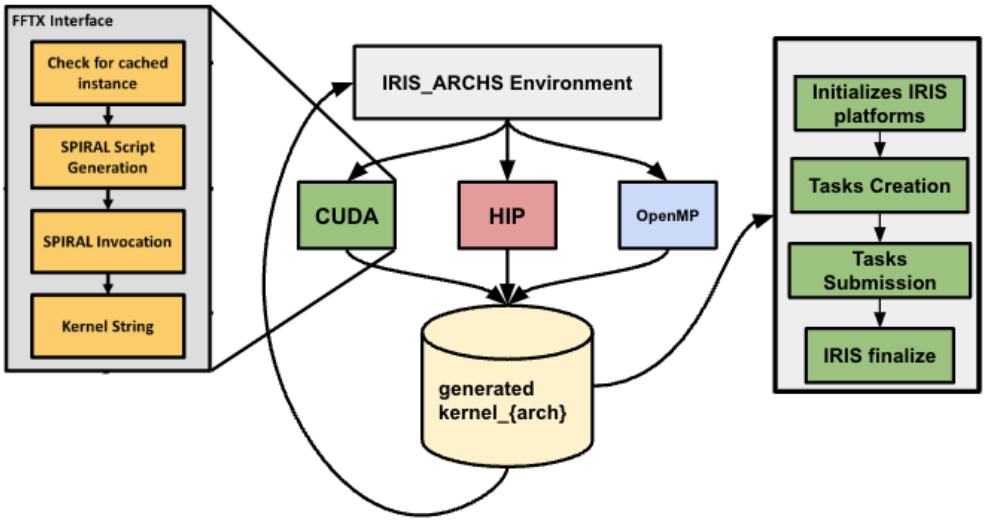

Figure 5: FFTX-IRIS Design. FFTX provides the optimized architecture-specific kernels, and IRIS executes them on the target device.

Code Generation. The `transform` member function houses a sequence of steps that translates a computation written in FFTX to an optimized kernel. This first step is a call to `semantics`, a user-provided function within the `FFTXProblem`. There, the user describes the computation they are trying to perform using the FFTX internal API. This internal API is then translated to the domain-specific language of the SPIRAL code generation system. This translation uses runtime information such as user-provided sizes to aid in code optimization. Once fully translated, a child process calls SPIRAL, which reads the translated user computation as input and generates an optimized kernel, which is written to disk in the appropriate format for IRIS.

Task Abstraction. FFTX uses a Just-In-Time compilation system to compile and link SPIRAL generated code to a running application. This would then execute the SPIRAL code by calling the three hook functions: the `spiral_init` function for kernel setup, the `spiral_execute` function for kernel invocation, and the `spiral_destory` function for cleanup. With FFTX-IRIS, this process is handled by the IRIS runtime system automatically, but the system needs to be configured in the proper way. To do this, the generated code is augmented with additional metadata. This metadata houses information such as global memory arrays, kernel signature arguments, number of device kernels, and kernel launch parameters in the case of GPUs. This information removes the need to generate the SPIRAL hook functions, so they are omitted.

Before the IRIS system is initialized, the metadata generated by SPIRAL is parsed and stored in buffers such that IRIS can properly schedule and execute kernels. The most important operation is the setup of the IRIS memory objects that act as mirrors to the original memory objects passed to the `FFTXProblem` constructor, as well as intermediate memory objects required by SPIRAL. These IRIS memory objects will perform any memory transfers between various devices without user involvement. Once complete, IRIS iterates over all generated kernels, adding them as tasks before submitting them to be executed, and writes the result back to the output memory object provided by the user. Figure 4 shows IRIS task generation and kernel invocation.

4.4 Enabling Portability and Heterogeneity

The design of FFTX-IRIS as a dynamic system allows execution on a wide variety of systems and architectures. Code generation through FFTX is done dynamically by parsing the IRIS_ARCHS environment variable. This means that SPIRAL will only generate kernels for the architectures that the system has available. Additionally, providing necessary metadata at code generation time means that setup buffers are populated with only what is required for the execution of that specific kernel. This allows an application binary compiled with a standard C++ compiler to execute on many compute platforms, providing full portability.

FFTX-IRIS enables heterogeneity by generating kernels for all available architectures in an IRIS-compatible format. SPIRAL generates kernels with only the required input and output pointers. This enables IRIS to determine memory flow and create dependencies between tasks intelligently. IRIS can then draw upon the appropriate hardware kernel to execute by following the scheduling algorithm.

4.5 Memory Management

Generating IRIS-compatible kernels in SPIRAL required modifications of the traditional SPIRAL code generation flow, specifically with regard to memory objects. In most SPIRAL generated kernels intermediate, device local pointers are created that reside statically on a device and are initialized using the `init_spiral` hook function. These intermediates are used in place of the user-provided pointers, which act as the initial input and final output pointers only.

In order for IRIS to expose complete heterogeneity, IRIS must control all classes of memory objects: read-only, write-only, and read/write. IRIS cannot assign tasks to arbitrary devices if generated intermediates are local and device-specific. Therefore, the SPIRAL-generated code is modified to remove this restriction. FFTX-IRIS omits the generation of these arrays as device arrays and moves

Table 1: Heterogeneous systems used in this research.

System	Equinox	Explorer	Zenith
GPUs	Total 4 GPUs 4× NVIDIA V100	Total 2 GPUs 2× AMD MI50	Total 2 GPUs 1 NVIDIA GeForce RTX 3090 1 AMD Radeon RX 6900
CPU	Intel Xeon E5-1698, 20 cores	AMD EPYC 7702, 128 cores	AMD Ryzen Threadripper, 32 cores
Compiler	GNU-11.4.0	GNU-8.5.0	GNU-8.5.0
CUDA and ROCm versions	CUDA-11.7	ROCm-5.1.0	CUDA-11.7 and ROCm-5.1.2

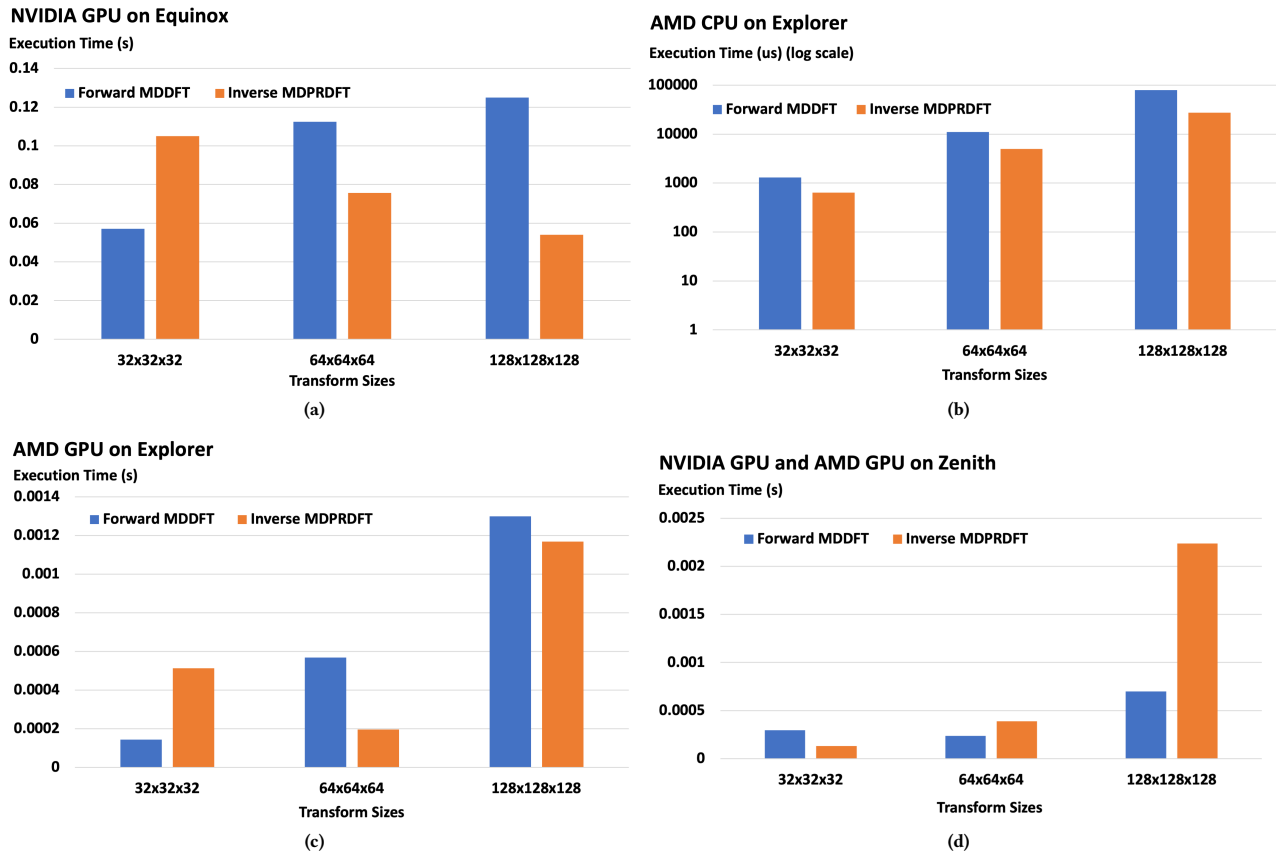

Figure 6: Portability of single transform.

the pointers into the relevant kernel signatures as needed. The metadata is further augmented to direct iris on the type of these pointers and their size.

At runtime, IRIS executes tasks based on a given scheduler. IRIS has different schedulers (such as *iris_any*, *iris_all*). When a kernel is scheduled, IRIS looks at the heterogeneous memory objects associated with that kernel, locates where the latest copy is, and then launches host-to-device or device-to-device data transfers. IRIS runtime performs all memory management without any interaction from the users.

5 EXPERIMENTS

This section evaluates FFTX-IRIS in different scenarios of portability and multi-device heterogeneity. First, we introduce systems and applications/benchmarks. Next, two aspects of FFTX-IRIS are evaluated: portability to various processors/systems and multi-device heterogeneity using a complex application.

5.1 Systems

Table 1 shows the heterogeneous systems used in this research, which are from the ExCL cluster of Oak Ridge National Laboratory. The *Equinox* node is an NVIDIA V100 DGX, whereas the *Explorer* node has two AMD MI50 GPUs. The most interesting node is *Zenith*, which has two GPUs, one from AMD and the other from NVIDIA, representing future heterogeneous systems for diverse heterogeneity.

5.2 Applications and Benchmarks

Single Transform. FFTX provides a number of single transform APIs for common FFTs that developers use in their applications.

These single transforms perform just the associated computation of that class of FFT. In general, FFT algorithms come in stages where the input signal is modified for a certain stage and stored back to be modified in a later stage. The computation within a stage and the number of stages vary depending on the input size and dimensionality, determining which FFT algorithm should be utilized. For this reason, it is difficult to have a general call per transform class, so FFT libraries have a specific planning phase to capture parameters and choose an algorithm, and execution phase to operate on the provided data.

A Partial Differential Equation Based Model. A number of scientific applications are based on different types of PDE models. These models usually follow complex algorithms that provide us with a scope to explore different high-performance techniques that, in turn, can be used to reduce their computational cost. Thus, they work as a good example to test the FFTX-IRIS system. For this work, we have chosen the example of the 2D Euler equations that are used in gas dynamics [7]. Our implementation in this work is based on an algorithm developed in [9]. It is part of the ongoing work of generating stencil codes for PDE applications using SPIRAL as a backend for a domain-specific language frontend called Proto. The new domain specific language is called Protox. Some of the initial results of this work can be found in [8]. An auto-generated DAG for the algorithm used to solve the 2D Euler equation numerically can be seen in Figure 7. We can observe that this algorithm provides the chance to test the heterogeneity as forks in DAG can occur simultaneously.

5.3 Portability of Single Transform Benchmarks

We demonstrate portability of FFTX-IRIS by taking two well-known single transform benchmarks, forward MDDFT and inverse MDPRDFT, and executing them on a few target systems with different available architectures. Each transform has three execution kernels that are sequentially dependent, with two intermediate pointers swapping between input and output. It is important to highlight that both the source code and executable binary did not change as we moved between machines; only the environment variable `IRIS_ARCHs` was modified to expose the available devices on each system. The binary was compiled with a standard *gcc* compiler on *Equinox*.

The performance shown in Figure 6 highlights how different machine configurations can influence the performance of a single application. We see that for the forward MDDFT benchmark, the best performance is on *Zenith*, while the inverse MDPRDFT benchmark performs best on *Explorer*. This is due to FFTX-optimized kernels being tuned for AMD architectures, which both systems have. On *Zenith* in particular, IRIS is able to utilize both an NVIDIA GPU and an AMD GPU for its computation, allocating memory on the appropriate device automatically. This removes the burden from the developer while still maintaining performance. This portability enables exploration into determining better machine configurations and opportunities to further optimize kernels for underperforming machines.

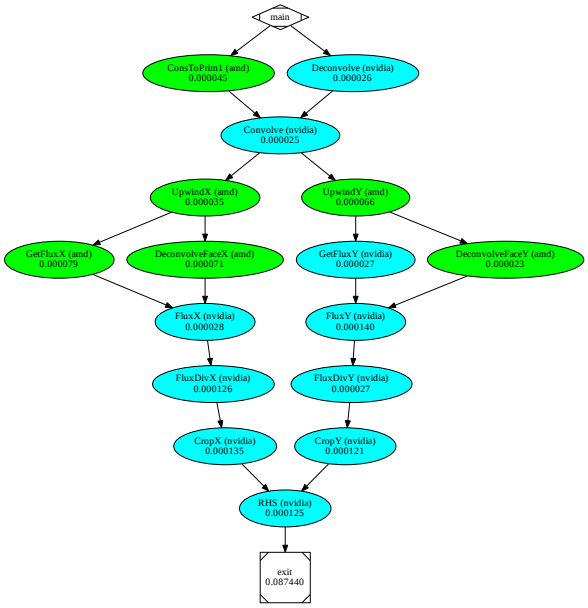

Figure 7: Heterogeneous execution of ProtoX on Zenith. GREEN ellipses represent execution on AMD GPUs and CYAN ellipses represent execution on NVIDIA GPUs.

5.4 Multi-Device Heterogeneity for ProtoX

Figure 7 shows the FFTX-IRIS generated DAG for the ProtoX application. Even though FFTX-IRIS strives to generate a complete task graph automatically, some application-specific instructions were included to construct a correct graph. Figure 7 shows the available task-level parallelism where multiple processors can be used at the same time to accelerate the execution. We ran ProtoX on *Zenith* to demonstrate that FFTX-IRIS can schedule tasks from a single application to different heterogeneous devices — in this case, NVIDIA and AMD GPUs. The *GREEN* and *CYAN* ellipses represent execution on AMD and NVIDIA GPUs, respectively. Although the DAG in Figure 7 shows execution on different GPUs, there is still room for optimization. This is due to the random scheduler from IRIS. However, this study demonstrates multi-device heterogeneity and opens the door for efficient scheduling through performance models. FFTX-IRIS aims to look into scheduling in the future.

5.5 FFTX-IRIS Overhead

Introducing a dynamic system like FFTX-IRIS comes at the cost of additional overhead. Code generation and task scheduling occur at runtime, consuming additional cycles to execute application computation kernels automatically. From a code generation standpoint, there is significant overhead when generating the kernels for the first time. This overhead is generally a one-time cost, as the kernel can be cached and called upon for all subsequent runs. Because of memory management, the runtime system overheads are slightly higher than vendor-specific runtime systems. IRIS performs extra booking keeping to place and moves memory dynamically. This is an active area of exploration that is being optimized.

6 CONCLUSION

FFTX-IRIS is a novel software framework for accelerating applications on heterogeneous platforms. It takes two independent frameworks, FFTX and IRIS, and seamlessly integrates them to provide increased performance on many new hardware platforms. FFTX is the user-facing kernel framework, providing optimized computation kernels through the SPIRAL code generation system. IRIS is the backend heterogeneous runtime system executing SPIRAL generated kernels on any and all available target platforms. FFTX-IRIS is transparent to the user, obfuscating unnecessary and complicated application porting. Moving forward, FFTX-IRIS will be able to support increased heterogeneity by running on multiple platforms at the same time.

ACKNOWLEDGMENTS

This work is funded, in part, by Bluestone, a X-Stack project in the DOE Advanced Scientific Computing Office with program manager Hal Finkel. This research used resources of the Experimental Computing Laboratory (ExCL) at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

REFERENCES

[1] C. Augonnet, S. Thibault, R. Namyst, and P. A. Wacrenier. 2011. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 23, 2 (2011), 187–198.

[2] Anthony M. Cabrera, Seth Hitefield, Jungwon Kim, Seyong Lee, Narasinga Rao Miniskar, and Jeffrey S. Vetter. 2021. Toward Performance Portable Programming for Heterogeneous Systems on a Chip: A Case Study with Qualcomm Snapdragon SoC. In *2021 IEEE High Performance Extreme Computing Conference, HPEC 2021, Waltham, MA, USA, September 20-24, 2021*. IEEE, 1–7. https://doi.org/10.1109/HPEC49654.2021.9622794

[3] H Carter Edwards, Christian R Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of parallel and distributed computing* 74, 12 (2014), 3202–3216.

[4] Franz Franchetti, Frédéric de Mesmay, Daniel McFarlin, and Markus Püschel. 2009. Operator Language: A Program Generation Framework for Fast Kernels. In *Domain-Specific Languages*, Walid Mohamed Taha (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 385–409.

[5] Franz Franchetti, Tze-Meng Low, Thom Popovici, Richard Veras, Daniele G. Spampinato, Jeremy Johnson, Markus Püschel, James C. Hoe, and José M. F. Moura. 2018. SPIRAL: Extreme Performance Portability. *Proceedings of the IEEE, special issue on "From High Level Specification to High Performance Code"* 106, 11 (2018).

[6] Jungwon Kim, Seyong Lee, Beau Johnston, and Jeffrey S. Vetter. 2021. IRIS: A Portable Runtime System Exploiting Multiple Heterogeneous Programming Systems. In *2021 IEEE High Performance Extreme Computing Conference, HPEC 2021, Waltham, MA, USA, September 20-24, 2021*. IEEE, 1–8. https://doi.org/10.1109/HPEC49654.2021.9622873

[7] Randall J. LeVeque. 2002. *Finite Volume Methods for Hyperbolic Problems*. Cambridge University Press. https://doi.org/10.1017/CBO9780511791253

[8] H. Mankad, S. Rao, B. Van Straalen, P. Colella, and F. Franchetti. 2023. ProtoX: A First Look. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*.

[9] Peter McCorquodale and Phillip Colella. 2011. A high-order finite-volume method for conservation laws on locally refined grids. *Communications in Applied Mathematics and Computational Science* 6, 1 (2011), 1–25.

[10] OpenACC. 2015. OpenACC: Directives for Accelerators.

[11] OpenMP. 1999. OpenMP Reference.

[12] S. Rao, A. Kutuluru, P. Brouwer, S. McMillan, and F. Franchetti. 2020. GBTLX: A First Look. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC43674.2020.9286231

[13] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. 2022. DISTAL: The Distributed Tensor Algebra Compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 286–300. https://doi.org/10.1145/3519939.3523437

[14] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. 2022. SpDISTAL: Compiling Distributed Sparse Tensor Computations. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 59, 15 pages.